WCH 2018
II Workshop em Computação Heterogênea

Conteúdo

WCH Artigos Completos

WCH S1

WCH S2

WCH Artigos Curtos

WCH Paslestras

Escalonamento de Tarefas usando Implementações Híbridas GPU/CPU das Heurísticas Min-min e Max-min no Cloudsim

Rafael F. Schmid[1] **, Edson N. Cáceres**[2]

[1] Instituto Federal de Mato Grosso do Sul
Rua José Tadao Arima, 222 – 79200-000 – Aquidauana – MS – Brasil

[2]Faculdade de Computação – Universidade Federal de Mato Grosso do Sul
Caixa Postal 549 – 79070-900 – Campo Grande – MS – Brasil

rafael.schmid@ifms.edu.br, edson@facom.ufms.br

Abstract. *Scheduling tasks is one of the key points to performance in heterogeneous computing. As NP-hardness problem, many heuristics have been proposed to solve it. In this work, we tried a hybrid implementation GPU/CPU of the Min-min and Max-min heuristics, and their simulation to compare with the standard scheduling of the Cloudsim framework. Min-min and Max-min heuristics presented better makespans than standard Cloudsim scheduling. Besides that, they are little affected when the number of machines increases, even if the computational power stay constant, which might help in the choice of the cheaper contract of a cloud computing environment.*

Resumo. *O escalonamento de tarefas é um dos pontos chave do desempenho em computação heterogênea. Como é um problema NP-Completo, várias heurísticas já foram utilizadas para resolvê-lo. Nesse trabalho, foi efetuada a implementação híbrida GPU/CPU das heurísticas Min-min e Max-min, e a simulação dessas heurísticas para comparação dos resultados com o escalonamento padrão do Cloudsim. As heurísticas Min-min e Max-min apresentaram* makespan *semelhantes e muito melhores, além disso, são pouco afetadas quando o número de máquinas aumenta, mesmo que o poder computacional se mantenha constante, o que pode auxiliar na redução do custo ao contratar um ambiente de computação em nuvem.*

1. Introdução

Ambientes de computação distribuídos tornaram-se populares nas últimas décadas como um meio de prover o poder de computação necessário para resolver problemas complexos. Geralmente, ambientes de computação distribuídos são compostos por muitos computadores heterogêneos capazes de trabalhar cooperativamente. Um exemplo de um ambiente distribuído é a computação em nuvem. Computação em nuvem é um sistema distribuído e paralelo no qual vários tipos de recursos são dinamicamente alocados aos usuários seguindo o acordo de nível de serviços (SLA) estabelecido entre o provedor do serviço e o consumidor. Em computação em nuvem, software, plataforma e infraestrutura são oferecidos aos consumidores como um serviço [Armbrust et al. 2010].

O uso de simulação é necessário em um ambiente de computação em nuvem, pois os serviços nesses ambientes possuem diferentes composições, configurações e requisitos

de implantação, o que torna extremamente desafiador quantificar o desempenho em um ambiente real. Por isso, é comum o uso de simuladores como o CloudSim para realizar essa tarefa [Calheiros et al. 2011].

Em geral, uma grande aplicação pode ser decomposta em um conjunto de tarefas menores e executadas em múltiplos processadores [Xu et al. 2014]. O escalonamento eficiente dessas tarefas nos recursos disponíveis é um do pontos chave para atingir alto desempenho [Topcuoglu et al. 2002]. Esse problema é mais conhecido como o problema do escalonamento em computação heterogênea (HCSP - Heterogeneous Computing Scheduling Problem). Como trata-se de um problema NP-difícil, muitas técnicas de heurísticas têm sido aplicadas para resolvê-lo, como Min-min, Max-min, GA (*Genetic Algorithm*), GSA (*Genetic Simulated Annealing*), *Sufferage*, etc [Braun et al. 2001]. Em função de suas características, a heurística Min-min é a mais rápida para esse problema e entrega soluções quase tão boas quanto GA, que é uma heurística mais demorada [Wu et al. 2000].

Neste trabalho, realizamos testes em um ambiente simulado de computação em nuvem, utilizando o Cloudsim, onde comparamos as heurísticas Min-min e Max-min com o algoritmo de escalonamento nativo da ferramenta. Com isso, mostramos a importância da utilização de um algoritmo de escalonamento eficiente e também que se o *framework* apresentasse outras heurísticas implementadas nativamente, poderia trazer benefícios aos usuários que desejassem melhores escalonamentos. Além disso, utilizamos dois cenários para mostrar que os *makespans* das heurísticas Min-min e Max-min, quase não são afetados pela quantidade de máquinas, o que pode trazer ganhos em relação ao custo de contratação de um ambiente em nuvem.

2. Formalização do HCSP

Um sistema de computação heterogênea (HC) é composto por vários computadores, também chamados processadores ou máquinas, e um conjunto de tarefas a serem executadas no sistema. O tempo de execução de uma tarefa varia de uma máquina para outra, então haverá competição entre as tarefas para utilizar a máquina capaz de executá-las no menor período de tempo [Nesmachnow and Canabé 2011]. A esse problema damos o nome de Problema de Escalonamento em Ambientes de Computação Heterogênea (HCSP - Heterogeneous Computing Scheduling Problem).

Apesar de muitas outras métricas de desempenho terem sido consideradas no problema de escalonamento [Leung 2004], a métrica mais comum do HCSP é o *makespan*. O *makespan* é definido como o tempo gasto a partir do momento em que a primeira tarefa começa até o momento em que a última tarefa é completada [Nesmachnow and Canabé 2011]. Logo, esse tempo é determinado pela máquina que irá terminar de processar suas tarefas por último.

O escalonamento dessas tarefas pode ser realizado de dois modos: estático e dinâmico. Esses modos definem em que momento as decisões de escalonamento serão feitas. No caso de escalonamento estático, as informações sobre todos os recursos no HC bem como de todas as tarefas estão disponíveis no instante em que a aplicação está sendo escalonada. No caso do escalonamento dinâmico, a ideia básica é alocar tarefas *on the fly* a medida que elas chegam para executar. Isso é útil quando é impossível determinar o tempo de execução, direções de desvios, número de iterações de um loop e nos casos de

tarefas estarem chegando em tempo real.

No modelo de tarefas independentes, todas as tarefas podem ser executadas independentemente, sem se preocupar com a ordem de execução, e as aplicações a serem executadas são compostas por uma coleção de tarefas indivisíveis que não têm dependência entre elas. A formalização abaixo apresenta o modelo matemático para o HCSP com o intuito de minimizar o *makespan*:

- Dado um sistema de computação heterogênea composto de um conjunto de M máquinas $P = m_1, m_2, ..., m_M$ e uma coleção de N tarefas $T = t_1, t_2, .., t_N$ a serem executadas.
- Considere uma função de tempo de execução $ET : T \times P \rightarrow R^+$, onde $ET(t_i, m_j)$ é o tempo necessário para executar a tarefa t_i na máquina m_j.
- O objetivo do HCSP é encontrar uma atribuição de tarefas às máquinas (uma função $f : T \rightarrow P$) que minimize o *makespan*, conforme definido na Equação 1.

$$\max_{m_j \in P} \sum_{\substack{t_i \in T \\ f(t_i)=m_j}} ET(t_i, m_j) \tag{1}$$

Um grande número de heurísticas foram propostas para obter soluções semi-ótimas do problema de escalonamento de tarefas em ambientes de computação heterogênea, já que trata-se de um problema NP-difícil. Nesta seção, apresentamos as heurísticas estáticas Min-min e Max-min, que abordaremos nesse trabalho, e que são comumente utilizadas na solução desse problema.

Min-min: A heurística Min-min começa com o conjunto U de todas as tarefas não mapeadas. Então, o conjunto de menor tempo de conclusão M para cada tarefa em U é encontrado analisando os tempos das tarefas já atribuídas às máquinas até aquele instante e o tempo das tarefas em U. A tarefa cujo tempo de conclusão seja o menor é escolhida e atribuída à máquina correspondente. A nova tarefa mapeada é removida do conjunto U, e o processo se repete até que todas as tarefas sejam mapeadas [Braun et al. 2001].

Max-min: A heurística Max-min também começa com o conjunto U de tarefas não mapeadas, encontra o conjunto de menor tempo de conclusão M para cada tarefa em U, analisando os tempos das tarefas já atribuídas às máquinas até aquele instante e o tempo das tarefas em U. Posteriormente, a tarefa cujo tempo de conclusão seja máximo é selecionada e atribuída à máquina que proporciona o menor tempo de conclusão. A nova tarefa mapeada é removida do conjunto U, e o processo se repete até que todas as tarefas sejam mapeadas [Kumar and Verma 2012].

3. Trabalhos Relacionados

Em 2001, foi realizado um trabalho por Braun et al. onde fizeram a comparação entre onze heurísticas diferentes. Em seus testes, utilizaram cenários com características de heterogeneidade consistente e inconsistente, bem como heterogeneidade de máquina alta e baixa. Os resultados mostraram que essas características afetam diretamente a qualidade da solução gerada por essas heurísticas. Além disso, pode-se evidenciar também que os algoritmos genéticos (GA) tiveram os melhores resultados na maioria dos cenários, mas que o Min-min pode ser resolvido muito rapidamente com resultados próximos ao GA [Braun et al. 2001].

Etminani et al. desenvolveram um algoritmo que seleciona a melhor solução entre as heurísticas Min-min e Max-min em cada iteração do algoritmo, baseado na utilização das máquinas e no tamanho das tarefas que ainda precisam ser escalonadas. Os resultados mostraram que essa heurística é melhor que as heurística Min-min e Max-min, quando aplicadas isoladamente. O novo algoritmo reduziu o *makespan*, aumentou a utilização das máquinas e o balanceamento de carga [Etminani and Naghibzadeh 2007].

Uma adaptação da heurística Min-min foi proposta por Kokilavani et al. para melhorar o balanceamento de carga. Após aplicar o Min-min, o algoritmo percorre as máquinas analisando a carga de cada uma e fazendo o reescalonamento quando necessário. Para isso, encontra a máquina com o maior *makespan* e a tarefa dentro dela que possui o menor tempo esperado. Depois procura qual máquina irá gerar o maior tempo de conclusão para essa tarefa. Se o maior tempo de conclusão for menor que o *makespan*, reescalona essa tarefa na nova máquina e calcula o novo tempo. Esses passos são repetidos até que todas as máquinas tenham sido consideradas [Kokilavani et al. 2011].

Em seu trabalho, Kurmar et al., fizeram a comparação entre um algoritmo genético padrão, que utiliza uma população inicial gerada aleatoriamente, e um algoritmo genético modificado, que utiliza como população inicial as heurísticas Max-min e Min-min. Eles utilizaram o CloudSim para validar o algoritmo e concluíram que um algoritmo genético consegue resultados melhores quando a sua população inicial já é uma boa solução para o problema sendo resolvido [Kumar and Verma 2012].

Em 2013, Bhoi et al. fizeram uma alteração no algoritmo *Improved Max-min* [Elzeki et al. 2012] para no lugar de selecionar as tarefas maiores para serem escalonadas primeiro, escolhe as tarefas que possuem a média do tempo de execução das tarefas que faltam escalonar ou que estão um pouco acima da média. A tarefa selecionada pelos algoritmos em cada passo é atribuída à máquina que possui o menor tempo de conclusão. A esse algoritmo deram o nome de *Enhanced Max-min* e conseguiram mostrar, através de testes no simulador CloudSim, que essa variação no algoritmo proporciona um *makespan* melhor [Bhoi et al. 2013].

Em 2014, Tsai et al., criaram o Hyper Heuristic Scheduling Algorithm (HHSA). Um algoritmo que seleciona aleatoriamente uma heurística a partir do conjunto de heurísticas disponíveis, e executa o número de iterações que satisfaz seus critérios. A cada execução são verificados se critérios de modificação e melhorias estão sendo satisfeitos para aquela heurística. Caso contrário, seleciona aleatoriamente uma nova. Os resultados mostraram que esse método consegue convergir mais rapidamente que os outros testados e apresentou os melhores resultados para a maioria dos casos.

4. Propostas de Escalonamento e Experimentos

Nesta seção apresentamos as propostas de escalonamento que podem ser incorporadas ao Cloudsim e como os experimentos foram realizados. Como as heurísticas Min-min e Max-min podem obter bons resultados com pouco custo de processamento (na escala de milissegundos), efetuamos a implementação híbrida GPU/CPU de ambas heurísticas, usando a mesma estratégia da Min-min apresentada no trabalho de Schmid et al. [Schmid and Cáceres 2018].

Para testar as estratégias de escalonamento no CloudSim abordamos três características:

1. Características das tarefas,
2. Características das máquinas e
3. Características do algoritmo de escalonamento.

Para analisar as características das tarefas, foram usadas 128, 256, 512, 1024 e 2048 tarefas, onde foram gerados 10 arquivos para cada uma dessas dimensões. Esses arquivos são compostos pelo números de instruções de cada tarefa em cada linha do arquivo. O número de instruções de cada tarefa foi gerado de forma aleatória no intervalo de 100 até 10000 instruções. A Tabela 1 mostra a média e o desvio padrão do número de instruções gerados para cada uma das 10 entradas para as dimensões 1024 e 2048.

	1024 Tarefas		2048 Tarefas	
	Média	Desvio Padrão	Média	Desvio Padrão
Entrada 1	5004,7227	2825,8179	5153,4551	2927,0715
Entrada 2	4999,2402	2859,6238	5259,6758	2831,0745
Entrada 3	5085,7295	2918,1625	4942,0654	2837,4149
Entrada 4	5089,9541	2857,8014	4973,4033	2796,6533
Entrada 5	4908,5400	2848,9555	5033,8291	2867,6947
Entrada 6	5004,3115	2888,7416	5132,0957	2860,0561
Entrada 7	5024,6641	2881,7100	4817,8721	2794,7192
Entrada 8	5072,2148	2848,7863	4965,8984	2854,9785
Entrada 9	4908,9268	2741,1339	4908,6230	2876,5185
Entrada 10	4975,2432	2888,5286	5133,0479	2902,4015

Tabela 1. Média e o Desvio Padrão do Número de Instruções Gerados nos Arquivos de Entrada.

Para avaliar as características das máquinas, foram criados 2 cenários. No primeiro, a quantidade de máquinas aumenta junto com a quantidade de tarefas, dessa forma, a relação tarefas/máquinas é sempre a mesma em todos os testes realizados. No segundo cenário a quantidade de tarefas é fixa e a quantidade de máquinas varia, porém, tomamos o cuidado de manter o mesmo poder computacional em todos os testes, ou seja, a medida que o número de máquinas aumenta, o poder computacional de cada uma diminui. Em ambos os cenários, os hosts recebem VMs de acordo com sua capacidade computacional em MIPS, ou seja, enquanto o somatório dos MIPS das VMs alocadas em um host não superar o MIPS do host, ele recebe mais VMs.

Foram utilizadas a heurística Min-min, que procura a cada iteração o par tarefa/máquina que gera o menor *makespan*, a heurística Max-min, que procura a tarefa com maior *makespan* e escalona na máquina que executa ela mais rápido, e o algoritmo padrão de escalonamento do Cloudsim, que distribui as tarefas igualmente entre as máquinas.

4.1. Cenário 1: Relação entre Tarefas e Máquinas Fixa

Nesse cenário, o número de tarefas aumenta na mesma proporção que a quantidade de máquinas. Por exemplo, no teste com 4 máquinas virtuais, existem 128 tarefas a serem escalonadas, já quando temos 8 máquinas virtuais, são 256 tarefas a serem escalonadas. Ao dividir o número de tarefas pelo número de máquinas a proporção é sempre 32 tarefas para cada máquina em todos os casos de testes.

Foram criados 6 *datacenters*, cada um com 6 *hosts* e poder computacional igual a 1000 MIPS, e as VMs foram alocadas conforme cabiam nos hosts. Dessa forma, pode haver mais de uma VM alocada em um mesmo *host*, e também hosts sem VMs alocadas. Foram utilizadas a partir de 4 máquinas virtuais com poder computacional (em MIPS) de: 1000, 500, 250 e 125. De um teste para o outro o número de máquinas virtuais com cada uma dessas velocidades dobra, conforme descrito na Tabela 2.

Quantidade de Máquinas Virtuais						
MIPS 1000	MIPS 500	MIPS 250	MIPS 125	Total de Máquinas Virtuais	Total de Tarefas	Total de Datacenter
1	1	1	1	4	128	1
2	2	2	2	8	256	1
4	4	4	4	16	512	2
8	8	8	8	32	1024	3
16	16	16	16	64	2048	6

Tabela 2. Cenário 1: Relação entre Tarefas/Máquinas igual em todos os testes.

4.2. Cenário 2: Poder Computacional do Ambiente e Quantidade de Tarefas Fixos

Nesse cenário, o somatório dos MIPS das máquinas virtuais em todos os testes é sempre igual a 2000, independente do número de máquinas virtuais. Logo, conforme o número de máquinas virtuais aumenta, o MIPS de cada uma diminui.

A Tabela 3 descreve o poder computacional em MIPS de cada VM e dos hosts de cada datacenter. Foram criados 5 *datacenters*, cada um com 3 hosts com poder computacional igual ao da VM mais rápida. Por exemplo, os hosts do teste com 4 VMs, foram criados com MIPS igual a 550 e as VMs foram alocadas conforme cabiam em cada host, dessa forma, pode haver mais de uma VM alocada em um mesmo *host*, e também hosts sem VMs alocadas.

Quantidade de Datacenter	Quantidade de VMs	MIPS de cada VM	MIPS de cada Host	Total de Tarefas
1	2	1100, 900	1100	1024
2	4	550, 525, 475, 450	550	1024
3	8	265, 262, 257, 253, 248, 244, 239, 232	265	1024
5	16	160, 158, 153, 148, 144, 137, 132, 129, 126, 117, 113, 107, 101, 94, 93, 88	160	1024

Tabela 3. Cenário 2: Poder computacional do ambiente e quantidade de tarefas iguais em todos os testes.

Os experimentos foram realizados em uma máquina com processador Intel(R) Xeon(R) CPU E5-1620 v3 3.50GHz, 32 GB de RAM, placa de processamento gráfico Quadro M4000 com 8 GB de memória global, versão do CUDA 8.0 e versão 5.4 do

compilador gcc. Os códigos dos programas e arquivos de entrada estão disponíveis no GitHub[1].

5. Resultados e Análises

Nesta seção apresentamos os resultados obtidos e a análise do comportamento das heurísticas em cada um dos experimentos.

5.1. Cenário 1: Relação entre Tarefas e Máquinas Fixa

A Tabela 4 apresenta a média e o desvio padrão do *makespan* das 10 entradas em cada dimensão do problema. O desvio padrão das heurísticas Min-min e Max-min vai diminuindo a medida que a dimensão do problema aumenta, o que indica que a diferença entre as soluções vai diminuindo. Isso já não ocorre com o algoritmo padrão de escalonamento do Cloudsim, pois ele não leva em conta o tempo de processamento das tarefas para escalonar.

	Min-min		Max-min		Padrão do Cloudsim	
Dimensão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão
4x128	326,3922	28,7015	329,4135	22,1948	1249,0391	212,6547
8x256	351,5895	11,1755	350,2154	10,0362	1300,7986	186,8099
16x512	347,0348	10,9200	344,7769	10,0727	1320,261	110,5326
32x1024	342,9106	8,3135	342,9451	4,3920	1246,9382	99,6824
64x2048	347,7862	8,3482	347,657	5,9988	1296,5012	112,7203

Tabela 4. Média e Desvio Padrão dos *Makespans* para cada Dimensão.

O *speedup* das heurísticas Min-min e Max-min quando comparadas com o escalonamento padrão do Cloudsim é apresentado na Tabela 5. Através dela podemos ver que ao usar qualquer uma dessas heurísticas o desempenho fica pelo menos 3,6 vezes melhor em qualquer dimensão do problema.

Na Tabela 6 estão descritos os tempos, em segundos, que as heurísticas Max-min e Min-min levam para definir o escalonamento das tarefas nas máquinas disponíveis. Como as dimensões do problema não são grandes e estamos usando um algoritmo híbrido GPU/CPU, o *overhead* da chamada da função em GPU toma praticamente todo o tempo do escalonamento, por isso o tempo de escalonamento é quase igual em todas as dimensões.

A média do *makespan* obtido em cada dimensão, já somado o tempo do escalonamento, é apresentada na Figura 1. Através dela podemos ver que as heurísticas Min-min e Max-min possuem *makespan* muito próximos. Isso ocorre porque o ambiente de teste utilizado é consistente. Um ambiente é consistente quando uma máquina m_j executa qualquer tarefa t_i mais rápido que outra máquina m_k, já um cenário inconsistente é considerado quando uma máquina m_j pode ser mais rápida que uma máquina m_k ao executar algumas tarefas, porém mais lento ao executar outras. Ou seja, no ambiente testado, a máquina mais rápida, será a mais rápida para executar qualquer tarefa. Portanto, a

[1]https://github.com/rafaelfschmid/cloudsim-scheduling.git

	Speedup	
Dimensão	Min-min	Max-min
4x128	3,8268	3,7917
8x256	3,6998	3,7143
16x512	3,8044	3,8293
32x1024	3,6363	3,6360
64x2048	3,7279	3,7293

Tabela 5. Speedup em relação ao algoritmo padrão do Cloudsim.

	Tempo de Escalonamento (s)	
Dimensão	Min-min	Max-min
4x128	0,5683	0,5551
8x256	0,5646	0,5543
16x512	0,5695	0,5608
32x1024	0,5912	0,5784
64x2048	0,5913	0,5932

Tabela 6. Tempo de Escalonamento das Heurísticas para as Dimensões do Problema

diferença entre os algoritmos Min-min e Max-min é a ordem de escolha das tarefas. Enquanto o Max-min começará o escalonamento pelas tarefas maiores, o Min-min começará pelas tarefas menores.

Outro ponto que chama a atenção na Figura 1 é que os *makespans* das heurísticas Min-min e Max-min quase não são afetados quando aumentamos a dimensão do problema. Como nesse cenário, quando dobramos o número de tarefas, dobramos também o número de máquinas, foi criado o Cenário 5.2 para verificar o comportamento dessas heurísticas ao se manter o poder computacional e a quantidade de tarefas, enquanto aumentamos a quantidade de máquinas.

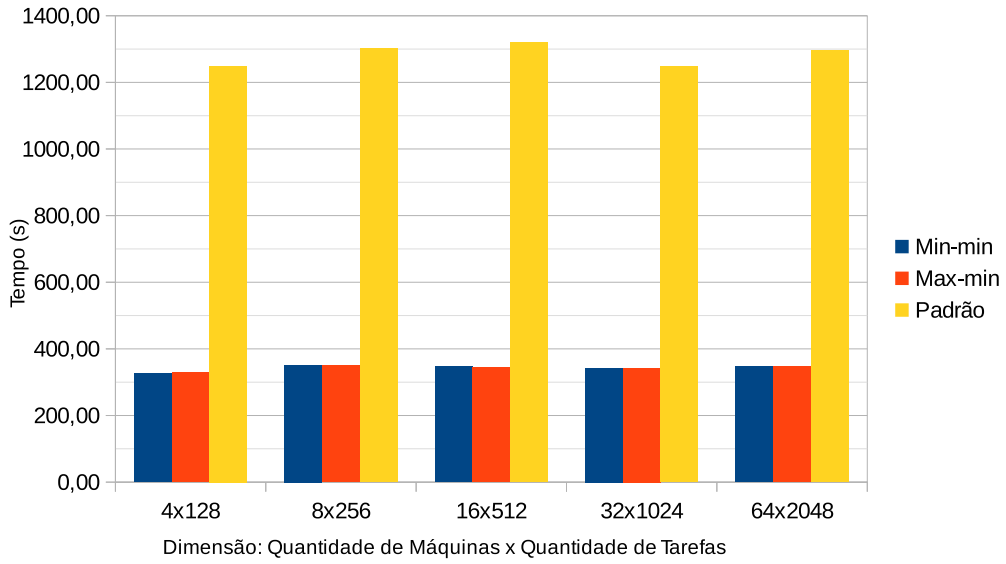


Figura 1. Média do Tempo de Execução de Cada Heurística para as Dimensões do Problema.

5.2. Cenário 2: Poder Computacional e Quantidade de Tarefas Fixos

Neste cenário, o objetivo era identificar o comportamento dos algoritmos com o aumento da quantidade de máquinas, mantendo o poder computacional e o número de tarefas.

Identificamos que, quando se mantém o poder computacional, o tempo para resolver o mesmo problema com mais máquinas usando Min-min ou Max-min, são muito

próximos. Ao analisar a Tabela 7 verifica-se que para todas as dimensões o Max-min conseguiu finalizar as tarefas com uma média de 2564 segundos, alterando apenas os números depois da vírgula. Já o Min-min, apesar de apresentar diferença nos *makespans* de uma dimensão para outra, as diferenças são muito pequenas, sendo que o maior tempo foi de 2575,0316 e o menor foi 2559,0802, que equivale a menos de 1% de diferença.

	Min-min		Max-min		Padrão do Cloudsim	
Dimensão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão
2x1024	2563,7544	34,3939	2564,9393	33,2900	2826,1775	71,0769
4x1024	2565,1586	33,5178	2564,7140	33,1896	2806,5358	82,5526
8x1024	2559,0802	32,3172	2564,5398	33,2552	2756,4284	119,8196
16x1024	2575,0316	32,8726	2564,5415	33,3325	3429,6016	114,7096

Tabela 7. Média e Desvio Padrão dos *Makespans* para cada Dimensão.

Outra análise efetuada, foi comparar a quantidade de tarefas atribuídas à cada VM. Como o algoritmo do Cloudsim, atribui a mesma quantidade de tarefas para todas as máquinas, ele não é apresentado nessa análise. Para comparar as heurísticas Min-min e Max-min, foi criada a Figura 2 que apresenta, para uma das entradas com 16 máquinas virtuais e 1024 tarefas, a quantidade de tarefas em cada VM, lembrando que o poder computacional da VM 0 é maior que o poder computacional da VM 1 e assim por diante. Logo, espera-se que VMs com IDs menores recebam mais tarefas que VMs com IDs maiores, e foi exatamente o que aconteceu em ambas as heurísticas. A diferença entre o Min-min e o Max-min é que a quantidade de tarefas das máquinas mais lentas é um pouco maior no Max-min do que no Min-min, e, consequentemente, o número de tarefas das máquinas mais rápidas é um pouco menor. Isso ocorre porque o Max-min é melhor para fazer o balanceamento de carga, já que resolve as tarefas maiores primeiro.

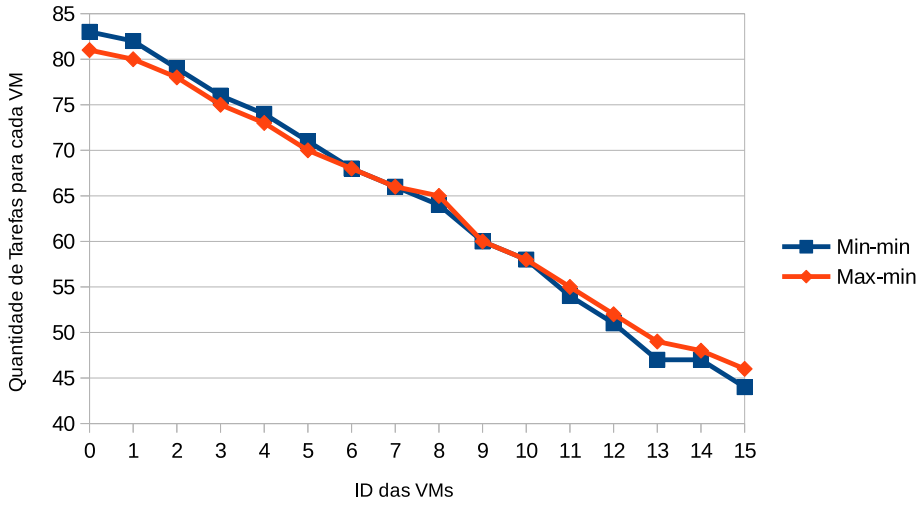


Figura 2. Quantidade de Tarefas em Cada VM.

Apesar de as máquinas mais rápidas receberem mais tarefas, os métodos Min-min e Max-min distribuem as tarefas de acordo com o *makespan* de cada máquina, ou seja,

eles atribuem a tarefa sendo escalonada à máquina que vai terminá-la primeiro, levando-se em conta as tarefas que já foram atribuídas a ela. A Figura 3 apresenta o *makespan* de cada VM utilizando uma das entradas com 16 máquinas virtuais e 1024 tarefas. Nela é possível perceber que o algoritmo Max-min efetua melhor o balanceamento de carga, por começar pelas tarefas maiores, aquelas que resultarão em maior desbalanceamento se deixadas para o final, que é o que ocorre com a heurística Min-min. O algoritmo padrão do Cloudsim distribui o mesmo número de tarefas para cada máquina, independente de seu tamanho, logo, é esperado que as máquinas mais lentas (com ID maiores) levem mais tempo para finalizarem seus trabalhos.

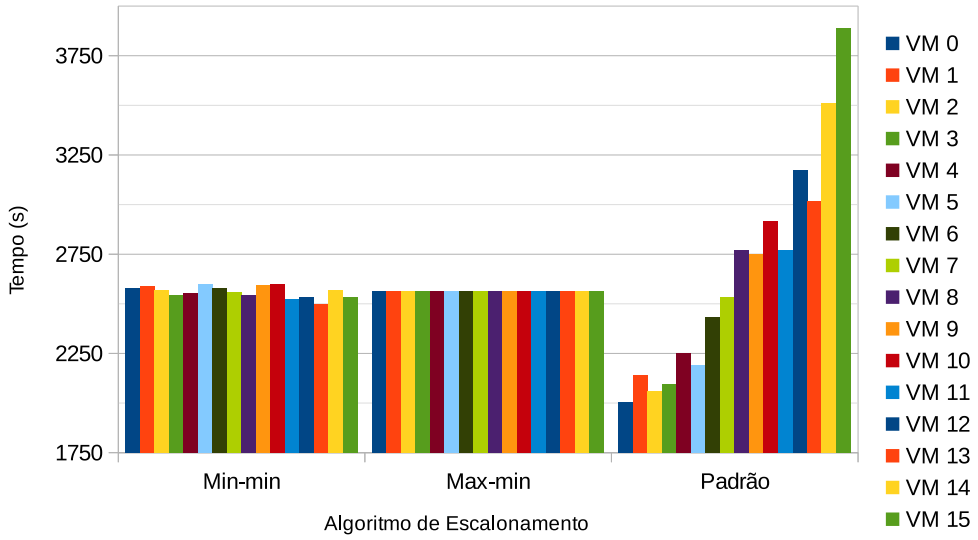


Figura 3. Tempo de Execução de Cada VM para uma das Entradas de Dimensão 16 máquinas e 1024 tarefas.

A Tabela 8 apresenta o percentual de utilização dos Datacenters. Para a obtenção dessa medida efetuamos a soma do MIPS de todas as VMs alocadas em um *Datacenter* e dividimos pelo total de MIPS dos hosts existentes nele. Por exemplo, no caso de teste com 8 VMs, foram alocados 3 *datacenters*, cada um com 3 hosts com poder computacional de 265 MIPS. No primeiro *datacenter* foram alocadas 3 VMs, com MIPS igual a: 265, 262 e 257, sendo assim, a utilização desse datacenter foi o somatório dos MIPS das 3 VMs dividido pelo somatório dos MIPS dos 3 Hosts: $\frac{265+262+257}{265+265+265} = 0,9862 = 98,62\%$. Essa tabela se aplica a todos os algoritmos, pois o algoritmo de alocação de VMs a um Host utilizado foi o padrão do Cloudsim, que aloca em ordem conforme as VMs vão cabendo em um Host.

Na Tabela 8 também é apresentado um cenário hipotético onde o cálculo do preço de um *datacenter* leva-se em conta o percentual de utilização do seu potencial de processamento, ou seja, se tiverem MIPS não utilizados, eles não são cobrados no preço final. Para simplificar as análises, consideramos o custo de 1 *datacenter* como sendo 100. Dessa forma, para que todos os ambientes custem o mesmo, os preços de cada *datacenter* devem seguir os preços sugeridos na tabela. Esses valores foram obtidos dividindo a utilização de 1 datacenter pelo somatório das utilizações dos outros datacenters. Por exemplo, para calcular o custo de cada datacenter ao utilizarmos 5 *datacenters* com baixo poder compu-

tacional, dividimos 60,61 (que é a utilização de 1 datacenter sozinho) por 416,67 (que é o somatório das utilizações dos *datacenters*): $\frac{60,61}{416,67} = 14,55$.

	1 Datacenter	2 Datacenters	3 Datacenters	5 Datacenters
Datacenter 0	60,61%	93,94%	98,62%	98,13%
Datacenter 1		27,27%	93,71%	89,38%
Datacenter 2			59,25%	80,63%
Datacenter 3				88,54%
Datacenter 4				60,00%
Utilização Total	60,61%	121,21%	251,57%	416,67%
Preço Sugerido de cada Datacenter	100,00	50,00	24,09	14,55

Tabela 8. Percentual de Utilização dos Datacenters e Preço Hipotético Sugerido

6. Conclusões

Considerando os tempos obtidos, fazer um escalonamento prévio das tarefas utilizando uma das heurísticas Max-min ou Min-min traz ganhos consideráveis para os resultados no Cloudsim. Visto que tratam-se de heurísticas simples, que se resolvem muito rapidamente, podemos inclusive sugerir a adição das mesmas como um escalonador nativo da ferramenta, seja resolvido todo em CPU ou de forma híbrida GPU/CPU, como fizemos nesse trabalho.

Além disso, identificamos que em um ambiente consistente as heurísticas apresentam tempos de conclusão semelhantes sempre que a relação tarefas/máquinas é a mesma, e também quando o poder computacional é o mesmo para um determinado número de tarefas, mesmo que a quantidade de máquinas seja diferente. Isso é importante porque sugere ao usuário que, ao contratar um ambiente de computação em nuvem, possa escolher um ambiente com mais ou menos máquinas e máquinas mais complexas ou mais simples, sem afetar de forma significativa o resultado final, e por um preço mais baixo.

6.1. Trabalhos Futuros

Como trabalho futuro pretendemos entender a precificação dos ambientes de computação em nuvem e gerar diversos cenários que identifiquem os ambientes mais indicados para cada dimensão do problema, além de efetuar testes com tarefas e máquinas com grande divergência entre si.

Agradecimentos

Os autores gostariam de agradecer o apoio financeiro da CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior do Ministério da Educação do Brasil.

Referências

[Armbrust et al. 2010] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

[Bhoi et al. 2013] Bhoi, U., Ramanuj, P. N., et al. (2013). Enhanced max-min task scheduling algorithm in cloud computing. *International Journal of Application or Innovation in Engineering and Management (IJAIEM)*, 2(4):259–264.

[Braun et al. 2001] Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810 – 837.

[Calheiros et al. 2011] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50.

[Elzeki et al. 2012] Elzeki, O., Reshad, M., and Elsoud, M. (2012). Improved max-min algorithm in cloud computing. *International Journal of Computer Applications*, 50(12).

[Etminani and Naghibzadeh 2007] Etminani, K. and Naghibzadeh, M. (2007). A min-min max-min selective algorihtm for grid task scheduling. In *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*, pages 1–7.

[Kokilavani et al. 2011] Kokilavani, T., Amalarethinam, D. G., et al. (2011). Load balanced min-min algorithm for static meta-task scheduling in grid computing. *International Journal of Computer Applications*, 20(2):43–49.

[Kumar and Verma 2012] Kumar, P. and Verma, A. (2012). Independent task scheduling in cloud computing by improved genetic algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(5).

[Leung 2004] Leung, J. Y. (2004). *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press.

[Nesmachnow and Canabé 2011] Nesmachnow, S. and Canabé, M. (2011). Gpu implementations of scheduling heuristics for heterogeneous computing environments. In *XVII Congreso Argentino de Ciencias de la Computación*.

[Schmid and Cáceres 2018] Schmid, R. and Cáceres, E. (2018). Implementações eficientes da heurística
min-min para o hcsp em gpu. In *III Encontro de Teoria da Computação (ETC 2018); XXXVIII Congresso da Sociedade Brasileira de Computação (CSBC)*. SBC.

[Topcuoglu et al. 2002] Topcuoglu, H., Hariri, S., and Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

[Wu et al. 2000] Wu, M.-Y., Shu, W., and Zhang, H. (2000). Segmented min-min: A static mapping algorithm for meta-tasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*, pages 375–385. IEEE.

[Xu et al. 2014] Xu, Y., Li, K., Hu, J., and Li, K. (2014). A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences*, 270:255–287.

πFlowMR – um protótipo dataflow escalável, implementado em um cluster de FPGAs de baixo custo

Silva Junior, J. T.[1]; Matias P.[2]; Ruggiero, C. A.[1]

[1]Instituto de Física de São Carlos – Universidade de São Paulo (IFSC / USP)
São Carlos – SP – Brazil

[2]Departamento de Computação – Universidade Federal de São Carlos
São Carlos – SP – Brazil

jtsjunior@usp.br, matias@ufscar.br, toto@ifsc.usp.br

Abstract. *This paper describes the current development and the main results of the πFlowMR Dataflow architecture prototype, inspired by the Manchester Dataflow Machine, implemented in a low cost FPGA cluster. Currently being developed in the Sao Carlos Institute of Physics, at the University of Sao Paulo, this prototype has presented interesting results, showing close to ideal speedups in some experiments, with a structure which can scale to several processing units.*

Resumo. *Este artigo descreve o atual desenvolvimento e os principais resultados do πFlowMR, o protótipo de uma arquitetura baseada em fluxo de dados, inspirado pela Máquina Dataflow de Manchester, implementado em um cluster de FPGAs de baixo custo. Em desenvolvimento no Instituto de Física de São Carlos, da Universidade de São Paulo, este protótipo vem apresentando resultados interessantes, obtendo ganho em desempenho muito próximo do ideal em alguns experimentos, com uma estrutura escalável que permite adicionar várias unidades de processamento.*

1. Introdução

Desde os anos 60 existe uma grande discussão acerca do ganho em desempenho computacional. Ainda nesta época, essa discussão já pairava sobre duas principais vertentes: de um lado, acreditava-se que seria possível o aumento indefinido na frequência do relógio dos processadores, de forma que as operações dos programas seriam executados a uma taxa de frequência mais alta e isso garantiria ganho em desempenho computacional, conforme previsto empiricamente pela Lei de Moore [Kish 2002]; o outro lado da discussão argumentava que haveria limitações físicas para o aumento da frequência do relógio e que para obter ganhos significativos em desempenho, seria necessária a realização de operações concorrentes onde fosse possível [Dennis 1980]. Pôde-se verificar nas últimas décadas que realmente existem limitações físicas que não permitem o aumento indefinido dessa frequência [Matzke 1997], como o princípio da incerteza ou a dissipação da potência em calor [Markov 2014]. Dessa forma, a outra vertente da discussão ganhou grande notoriedade e atualmente o ganho em desempenho computacional é obtido principalmente a partir da execução concorrente de operações [Sharp 1992, Flynn et al. 2012, Markov 2014].

Na arquitetura convencional de Von Neumann, a computação ocorre a partir da manipulação consecutiva das informações presentes em uma memória. Nesse formato, os

programas podem ser decompostos em processos e cada ação que eles devem executar é realizada a partir de uma sequência de operações. Se houver recursos paralelos, esses processos podem se comunicar entre si, seja a partir da leitura de uma memória compartilhada ou por meio da passagem de mensagens explícita entre as unidades de processamento que compõem o processador. Sendo assim, é possível que as operações sequenciais de um programa possam ser executadas paralelamente, em um cenário onde se deve haver bastante cuidado sobre conflitos de recursos em ações concorrentes [Barahona and Gurd 1986]. Perceba portanto, que é necessário grande esforço intelectual do programador, a fim de conseguir extrair desempenho computacional a partir de processadores convencionais em uma abordagem de programação concorrente.

A fim de reduzir a dependência do programador em extrair a capacidade de paralelismo do processador, diversos paradigmas de computação paralela foram explorados ao longo deste período [Magna 1997]. Dentre estas abordagens, uma que se destaca por sua simplicidade e capacidade de exploração de paralelismo, é o paradigma da computação dirigida por dados. Nesse paradigma, as instruções presentes no programa que se deseja executar são ativadas a partir da disponibilidade dos dados necessários para sua execução [Arvind and Culler 1986]. Com isso, o paralelismo torna-se intrínseco à arquitetura e não mais dependente, exclusivamente, da estrutura do programa que se deseja computar. Uma implementação desse paradigma, que se destacou em sua época, foi a Máquina Dataflow de Manchester (MDFM), um poderoso processador baseado no modelo dinâmico a fluxo de dados [Gurd 1985], e que foi tomado como base para o presente estudo.

Uma das principais características da MDFM é a fina granularidade de seu conjunto de instruções. Essa característica permite explorar uma quantidade significativa de paralelismo. Entretanto, a estrutura da arquitetura necessita de uma grande quantidade de recursos para lidar com tamanho paralelismo [Sargeant and Ruggiero 1987]. Dessa forma, o principal objetivo deste projeto é a distribuição das instruções dos programas entre múltiplas instâncias da MDFM. Essa estratégia é suficiente para o aumento sensível na granularidade da arquitetura [Magna 1997], garantindo a divisão dos recursos necessários para a exploração de todo o paralelismo que essa estrutura é capaz de suportar. Além disso, a partir deste sistema paralelo, espera-se obter uma quantidade ilimitada, a priori, na quantidade de unidades de processamento capazes de serem exploradas por esta estrutura. Com isso, busca-se a exploração massiva do conceito de escalabilidade, a partir de um protótipo baseado na MDFM.

Para a descrição do *hardware*, foi utilizado o Bluespec SystemVerilog, uma linguagem de descrição de *hardware* de alto nível, fortemente tipada, onde o comportamento do *hardware* é definido a partir de um conjunto de regras [Nikhil and Arvind 2009]. Diferente das linguagens convencionais, o Bluespec faz uso de um sistema de reescrita de termos (TRS, do inglês *Term Rewriting System*), a partir do qual é capaz de determinar toda a sequência de ações possível, dentro de um ciclo do relógio, livre de conflitos e em elaboração estática. Essa característica torna a linguagem mais simples para a descrição do *hardware* e é o que garante o ganho em produtividade proposto por ela. A análise em TRS, realizada pelo compilador do Bluespec, é equivalente àquela que o *designer* de uma descrição deveria fazer, a fim de obter sua estrutura em elaboração estática. Para grandes projetos, o TRS tende a ser mais confiável e eficiente que uma análise manual. O Bluespec ainda conta com o Bluesim, uma poderosa ferramenta para simulação, nativa

da linguagem, que garante grande produtividade para a descrição de *hardware*. Uma das principais características dessa linguagem é a garantia de que todo o código compilado por ela será sintetizável em FPGA [Nikhil 2004].

A seguir, na seção 2, será introduzido o modelo a fluxo de dados e a MDFM, além de uma descrição sumária sobre as implementações e principais resultados das versões anteriores do presente projeto. Na seção 3 serão descritas as principais implementações da πFlowMR (Multianel, do inglês *Multiring*). Em seguida, na seção 4, serão descritas as principais métricas utilizadas para avaliação dos resultados e os programas utilizados para a validação da estrutura, seguido pela introdução e discussão dos principais resultados obtidos até o presente momento. E, finalmente, na seção 5, serão descritas as principais conclusões avaliadas nestes resultados.

2. πFlow – Primeira versão

O fluxo de dados é um paradigma de computação paralela, em que programas são descritos no formato de um grafo bidimensional [Arvind and Culler 1986]. Nessa representação, instruções que podem ser executadas paralelamente são dispostas lado a lado e instruções que dependem de resultados anteriores são dispostas uma abaixo da outra [Dennis 1980, Gurd et al. 1985]. Nesse paradigma, os dados transitam pela arquitetura através de fichas, um conjunto de dados estruturados que carregam todas as informações necessárias para a execução de instruções. O fluxo de dados suporta duas formas de implementação: estática e dinâmica. No fluxo de dados estático, todas as operações que o programa precisa executar devem ser expressadas explicitamente no grafo. Já no fluxo de dados dinâmico, existem marcadores nas fichas que permitem a reutilização de código, para a execução de funções, laços e recursividade, por exemplo [Veen 1986].

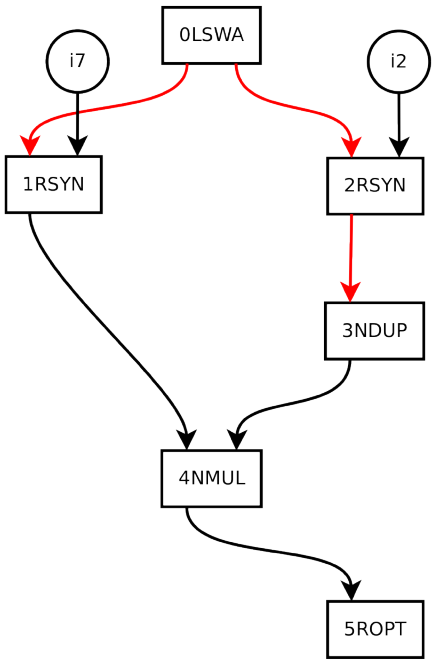


Figura 1. Representação de um programa para o modelo à fluxo de dados.

Um exemplo básico de programa característico em fluxo de dados é mostrado

na Figura 1. Note que é possível analisar as propriedades paralelas do programa apenas visualizando sua representação gráfica. Esse programa – bastante simples, e utilizado apenas para fins didáticos – realiza a multiplicação entre os números inteiros 7 e 2, a partir das informações adicionadas explicitamente em seu código. No programa, a instrução 0 sincroniza as chamadas às instruções subsequentes. As instruções 1 e 2 sincronizam as fichas de entrada, com os valores estáticos definidos em suas entradas à direita. A instrução 3 é utilizada apenas para fins didáticos[1]: ela poderia estar conectada a outro trecho de código a partir de sua saída direita mas, no exemplo, apenas repassa sua ficha de entrada à saída esquerda desta. A instrução 4 realiza a multiplicação entre os dados de suas fichas de entrada. E, finalmente, a instrução 5 encaminha sua ficha de entrada como uma ficha de saída do programa.

2.1. Máquina Dataflow de Manchester

A MDFM foi um protótipo baseado no modelo a fluxo de dados dinâmico, estruturado em um formato cíclico, composto por unidades por onde as fichas transitavam, a fim de extrair as informações necessárias para a execução das instruções que lhes cabiam [Gurd et al. 1985]. Na MDFM, as fichas possuíam o formato descrito na Expressão 1. O identificador "dado" era uma cadeia de 37 bits, em que 5 bits determinavam o tipo de dado que essa ficha carregava, e uma cadeia de 32 bits descrevia a informação carregada por ela. O rótulo era responsável por determinar as características dinâmicas do modelo a fluxo de dados e, portanto, era utilizado para a realização de operações como laços e recursividade, sem a necessidade de replicação de código. O destino identificava o endereço na memória da Unidade de Instruções, que armazenava a operação do conjunto de instruções da arquitetura que essa ficha deveria executar. Na MDFM, o marcador era utilizado para identificar se a ficha corrente era uma ficha de instruções [Gurd et al. 1985].

$$\langle \text{dado}\,(37\,\text{bits}),\ \text{rótulo}\,(36\,\text{bits}),\ \text{destino}\,(22\,\text{bits}),\ \text{marcador}\,(1\,\text{bit})\rangle \tag{1}$$

A visão geral da MDFM é mostrada na Figura 2. A Fila de Fichas era composta por uma FIFO, que comportava até 4k fichas de armazenamento interno, tendo a função de fornecer uma nova ficha a cada ciclo do relógio a unidade vizinha. Ela atuava como um *buffer* da estrutura. A Unidade de Pareamento tinha o objetivo de unir fichas destinadas à mesma instrução. Essa unidade era formada por uma memória RAM, para o armazenamento de fichas que aguardavam por suas fichas parceiras. A união de fichas destinadas à mesma instrução era realizada por meio de uma função *hash* aplicada entre o rótulo e o destino da ficha recém chegada à unidade. Se houvesse uma ficha armazenada no endereço referente ao resultado da função *hash*, destinada à mesma instrução, então ambas as fichas formavam um pacote que era encaminhado à Unidade de Instruções. Como podem ocorrer colisões no endereço gerado pela função *hash*, havia a possibilidade de existir uma ficha no endereço que não era destinada à mesma instrução. Neste caso, a ficha recém chegada à unidade era encaminhada à Unidade de Sobrecarga, formada por uma lista encadeada, que era analisada na saída dos pacotes da Unidade de Pareamento, em busca de fichas que podiam ser encaminhadas em retorno a essa unidade [da Silva and Watson 1983].

[1]Reforçando que a ficha de saída da instrução 1 aguarda pareamento, até a instrução 3 ser executada.

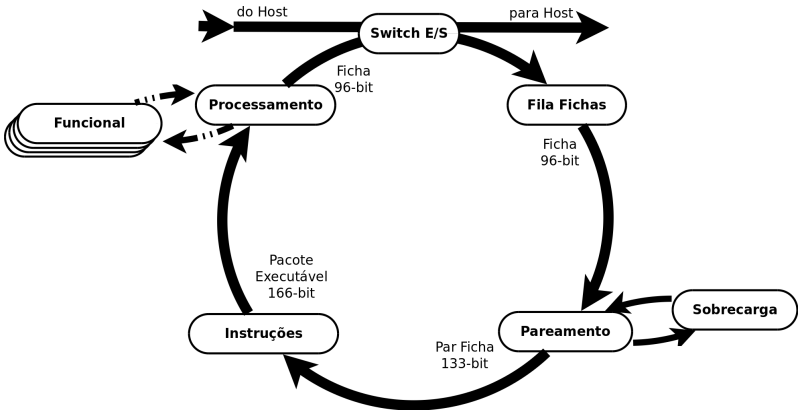


Figura 2. Visão geral da estrutura da Máquina Dataflow de Manchester.

A Unidade de Instruções era formada por uma memória RAM, que armazenava todas as instruções do programa que estava sendo executado pela máquina em linguagem montadora (grafo). A informação "destino", presente em todas as fichas, refere-se ao endereço (na memória RAM desta unidade) da instrução que a ficha deve executar. A unidade obtinha as informações da instrução, formando pacotes executáveis completos e encaminhando-os à Unidade de Processamento. A Unidade de Processamento distribuía os pacotes de entrada à primeira Unidade Funcional livre para execução, a partir de um duto de comunicação compartilhado entre elas. A execução das instruções gerava novas fichas subsequentes, que eram encaminhadas à Fila de Fichas para continuar a execução do programa. Caso a ficha resultante da execução fosse uma ficha de saída da máquina, esta era encaminhada ao *Switch*, que a enviava ao *Host* conectado à máquina [Gurd et al. 1985].

2.2. πFlow

A primeira versão da πFlow tinha o objetivo principal de implementar a MDFM com o uso de tecnologias modernas. O diagrama ilustrativo dessa versão é apresentado na Figura 3. Note que a principal modificação da arquitetura, com relação à máquina original, é a dupla ligação entre a Unidade de Processamento e a Fila de Fichas. Essa implementação foi proposta pois o compilador da MDFM era otimizado o suficiente para gerar a quantidade máxima possível de fichas subsequentes, que nessa arquitetura era de duas fichas por instrução [Magna 1997]. Essa característica buscava o aproveitamento desse paralelismo intrínseco da arquitetura, na geração das fichas subsequentes, encaminhando-as paralelamente à Fila de Fichas. Essa implementação forneceu um singelo ganho em desempenho, porém uma falha sistemática foi encontrada durante a análise dos seus resultados.

A estrutura proposta foi implementada com sucesso, porém a escalabilidade obtida estava aquém da que era observada na máquina original [Silva Junior 2016]. Isso ocorria pois, diferentemente da MDFM, na πFlow as unidades funcionais eram sincronizadas com a mesma frequência de relógio que o restante da estrutura da máquina. Apesar do desempenho individual das unidades obtido assim ter sido muito maior que o da MDFM, essa característica fez com que a estrutura da arquitetura passasse a ser um gargalo, impedindo que o desempenho aumentasse com a adição de mais unidades funcionais. Essas unidades precisavam de apenas 4 ciclos para a geração de novas fichas subsequentes, mas a estrutura da máquina consumia um total de 6 ciclos, no melhor caso, entre a geração

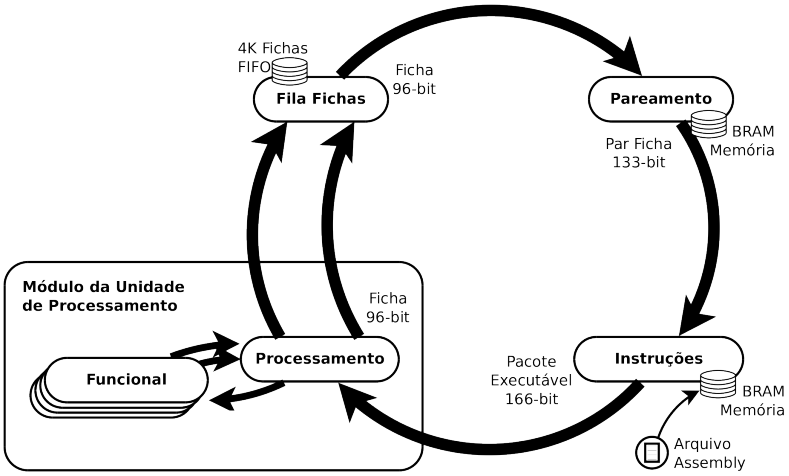


Figura 3. Visão geral da estrutura da versão inicial da πFlow.

de uma ficha subsequente e sua execução efetiva. Dessa forma, a máquina não era capaz de fornecer uma quantidade significativa de fichas para manter as unidades funcionais ocupadas o suficiente, a fim de atingir um desempenho apreciável. Essa característica motivou a implementação da πFlowBW (Largura de Banda, do inglês *Bandwidth*).

2.3. πFlowBW – *Bandwidth*

Com base no ganho em desempenho observado com a introdução dos dutos de saída na Unidade de Processamento, foi introduzida a hipótese de que o aumento no *throughput*[2] das unidades que compõe a arquitetura poderia implicar na obtenção de um desempenho equivalente àquele que era atingido na arquitetura original. Afinal, mais fichas estariam disponíveis às unidades funcionais por unidade de tempo. Assim, foi introduzido o conceito de *dutos de largura de banda* na estrutura da πFlow, ilustrado na Figura 4. Foram adicionadas estruturas parametrizáveis, que permitiam variar facilmente, em tempo de síntese, o *throughput* da máquina.

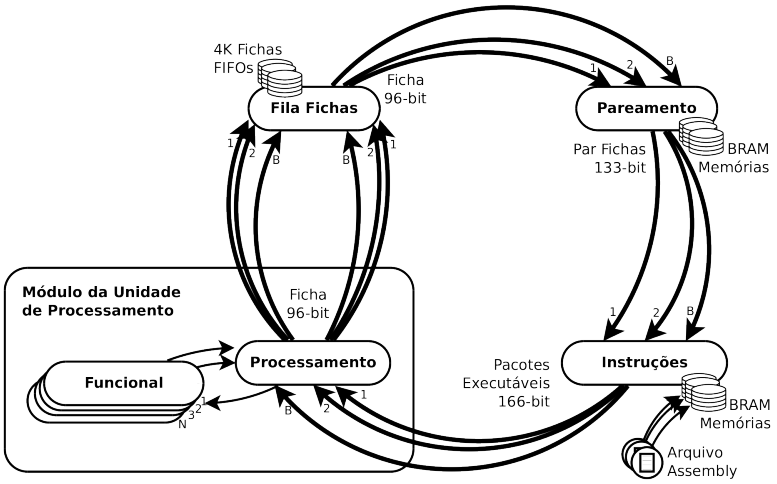


Figura 4. Visão geral da estrutura da πFlowBW.

[2]Quantidade de fichas e pacotes que transitam entre as unidades da estrutura por unidade de tempo.

Com essa estrutura, para programas suficientemente paralelos, passou a ser possível fornecer uma quantidade muito maior de fichas que transitavam entre as unidades da arquitetura, limitada por um valor constante, definido em tempo de compilação no código Bluespec. Tanto os recursos internos de cada unidade, como por exemplo a FIFO de armazenamento da Fila de Fichas ou a memória RAM da Unidade de Pareamento, quanto as FIFOs de comunicação entre as unidades da estrutura, passaram a ser divididas entre a quantidade de dutos que compunham a solução. Essa implementação garantiu uma escalabilidade similar à que podia ser obtida na arquitetura original.

Uma das principais limitações observadas nessa implementação era a baixa quantidade de unidades funcionais possíveis de serem sintetizadas em FPGA. Essa característica se dava devido à complexidade dos circuitos necessários para distribuir as fichas entre os dutos de largura de banda e as unidades funcionais. Com isso, a quantidade de recursos necessários para abrigar soluções mais robustas – com mais unidades funcionais – acabou ficando incompatível com a proposta inicial de usar placas de FPGA de baixo custo.

3. πFlowMR – *Multiring*

O principal objetivo da versão atualmente em desenvolvimento do projeto é implementar um conceito conhecido como Multianel. Nessa abordagem, busca-se explorar uma quantidade muito maior de unidades funcionais – em comparação com a primeira fase do projeto – mas ainda utilizando placas de FPGAs de baixo custo. Na primeira fase do desenvolvimento, foi construída uma estrutura cíclica para alimentação de uma quantidade fixa de unidades de processamento. O Multianel consiste em utilizar múltiplas unidades destes anéis, cada qual com a mesma quantidade de elementos de processamento, que estarão conectados a seus vizinhos por meio de canais de intercomunicação [Barahona 1984], como destacado na Figura 5.

Para descobrir o anel ao qual pertence determinada ficha, foi utilizado apenas o rótulo *Nome de Ativação*[3]. Outras abordagens foram experimentadas, como compor o Nome de Ativação com o endereço da instrução que deve ser executada pela ficha (solução equivalente àquela utilizada pela Unidade de Pareamento), mas estas implementações acabaram aumentando consideravelmente a comunicação entre os anéis e, com isso, adicionando um novo gargalo à arquitetura. Quanto menos comunicação for necessária, melhor será o desempenho da máquina: no paradigma da computação dirigida por dados, busca-se executar uma instrução o quanto antes, assim que todos os dados estiverem disponíveis [Magna 1997, Barahona and Gurd 1986].

Ao utilizar apenas o Nome de Ativação para determinar o anel de uma ficha particular, promove-se um aumento na granularidade da máquina[4], fazendo com que um trecho inteiro do programa (um nível da recursividade) seja executado em cada anel. Com isso, um anel é capaz de gerar fichas para si próprio ou para qualquer outro anel da estrutura. Fichas que dependem de comunicação ocorrem somente na execução de instruções que geram novos Nomes de Ativação (instrução GAN, do inglês *Generate Activation Name*).

[3]O Nome de Ativação é utilizado para permitir a chamada de funções durante a execução de programas. Algumas instruções específicas do conjunto de instruções da arquitetura são utilizadas para manipular um registrador compartilhado entre as unidades funcionais que armazena um número inteiro, sequencial e único, durante a execução de um programa.

[4]A partir da aglutinação de instruções.

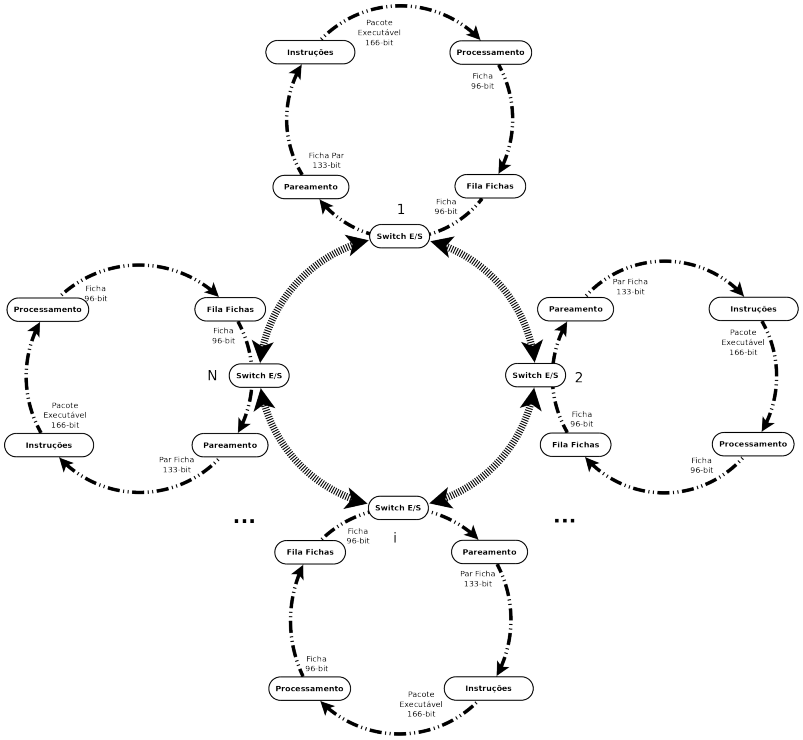


Figura 5. Visão geral da topologia da rede de comunicação entre os anéis na estrutura proposta pelo presente trabalho.

4. Resultados e Discussões

Uma das principais métricas utilizadas para analisar o desempenho de estruturas paralelas é o *speedup*. Esse indicador compara o desempenho do programa executado em uma única unidade de processamento com o desempenho do mesmo programa executado em N unidades. O *speedup* é ideal quando ocorre um crescimento linear com N, o que indica que o ganho em desempenho com a adição de unidades de processamento é exatamente proporcional a essa quantidade. Devido a fatores de sobrecarga em qualquer algoritmo ou arquitetura paralela, seu desempenho sempre ficará abaixo da curva ideal [Nussbaum and Agarwal 1991]. No presente projeto, essa sobrecarga se dá principalmente devido à estrutura do anel, enquanto as fichas e pacotes passam por entre as unidades que compõem a arquitetura, até a execução efetiva das instruções.

Outra métrica muito importante para avaliar o desempenho de arquiteturas paralelas, é o paralelismo médio do programa – "π", como definido em máquinas clássicas baseadas em fluxo de dados [Gurd et al. 1985]. Essa métrica fornece uma indicação do *speedup* máximo que um algoritmo pode obter em uma execução paralela. Ele é obtido a partir da razão entre duas grandezas: $\pi = S_1/S_\infty$. Na expressão, S_1 representa a quantidade de instruções do algoritmo, sem qualquer paralelismo, ou seja, a quantidade de nós da representação do programa no grafo em fluxo de dados. O S_∞ é o caminho crítico do grafo em fluxo de dados: supondo que se tenha uma quantidade infinita de unidades de processamento, esse indicador apresenta a quantidade de passos até finalizar a execução do programa.

Para validar a estrutura da máquina, foram utilizados dois *micro benchmarks* que apresentam grande potencial de paralelismo. Ambos foram gerados usando o compilador

da Máquina Dataflow de Manchester [Böhm and Sargeant 1989, Bohm and Gurd 1990]. O primeiro programa faz o cálculo do fatorial de determinado valor N, mas como o resultado dessa operação "cresce" muito rapidamente com N, o programa foi alterado para realizar a operação de soma no lugar da multiplicação, resultando no cálculo de uma progressão aritmética, de 1 a N, com razão 1. Essa operação foi chamada de "somatorial" e apresenta grande potencial paralelo, pois foi implementada de forma duplamente recursiva, ou seja, cada função é chamadas duas vezes em cada nível da recursividade. O outro programa faz o cálculo da integração da função $f(x) = x^2 + 6x + 10$, nos limites definidos como parâmetros de entrada. A integração é realizada utilizando o método do trapézio, implementado também de maneira duplamente recursiva. Ambos os programas possuem características que fazem crescer seu paralelismo proporcionalmente a um valor bem definido. No somatorial, N determina o tamanho do programa[5]. Por exemplo, o somatorial de 10 será expandido na forma de uma árvore binária, com 20 funções paralelas. No programa de integração, define-se o parâmetro de profundidade, que determina a quantidade de divisões na curva da função de referência e, consequentemente, o tamanho do programa durante sua execução. Por ter sido implementado de forma duplamente recursiva, esse tamanho também cresce como uma potência de dois.

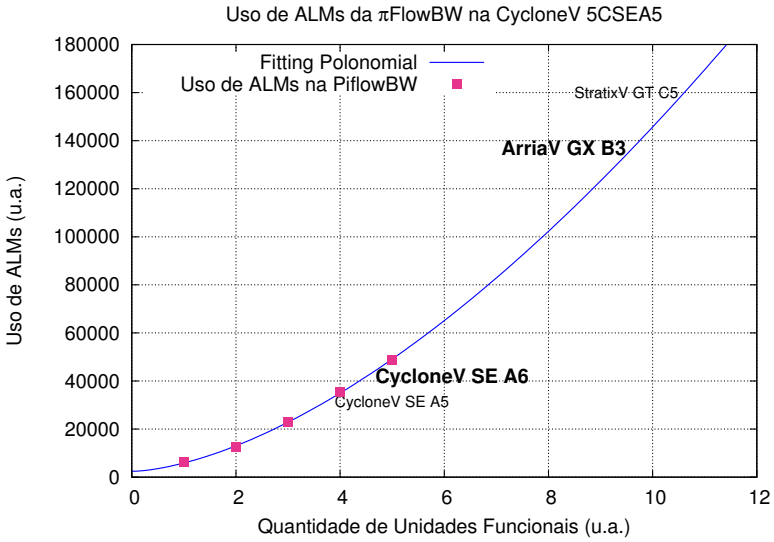
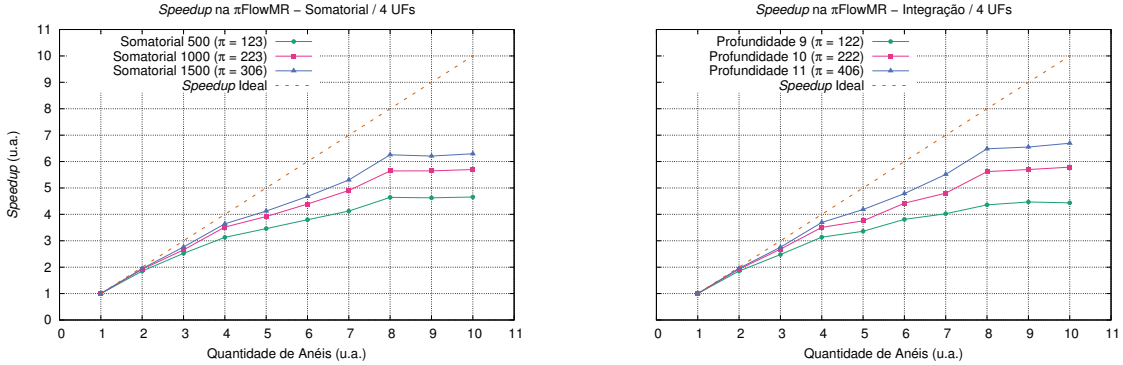


Figura 6. Ajuste teórico da quantidade de elementos lógicos utilizados para a síntese da arquitetura no chip de FPGA Cyclone V SoC 5CSEMA5F31C6.

Todos os resultados apresentados neste artigo foram obtidos a partir de simulação. Essa abordagem foi motivada pela filosofia do Bluespec, que gera apenas o subconjunto sintetizável do Verilog e, portanto, toda solução obtida a partir de seu compilador é sintetizável e o mais próximo possível da solução definitiva do circuito proposto pelo *design* descrito nessa linguagem [Nikhil 2004]. Apesar dessa estratégia ter sido adotada, foi realizada a análise da utilização de recursos da πFlowBW por meio de síntese para a placa Terasic DE1-SoC, que possui uma FPGA Cyclone V A5. A Figura 6 mostra a quantidade de elementos lógicos necessários para sintetizar a estrutura com até 5 unidades funcionais. A partir desses pontos, foi realizado o ajuste de uma função polinomial e, por

[5]O tamanho do grafo em fluxo de dados em tempo de execução, expandido pelas características dinâmicas da arquitetura.

meio de sua extrapolação, foi possível prever a utilização de elementos lógicos necessários para a síntese dessa solução em outras placas de FPGA comerciais, que atendem às necessidades deste projeto. Como pretende-se construir o Multianel com placas DE10-Nano, será possível sintetizar um máximo de 4 unidades funcionais. A solução pôde ser sintetizada para suportar uma frequência de operação de 50 MHz no *chip* de FPGA proposto. Perceba que, apesar de neste projeto focarmos na utilização de placas de baixo custo, a πFlowMR pode perfeitamente ser utilizada em FPGAs mais densas, permitindo a exploração de soluções mais robustas.



(a) Somatorial na πFlowMR. **(b)** Integração na πFlowMR.

Figura 7. Gráfico do desempenho das últimas versões da πFlowMR, como função da quantidade de anéis na estrutura da máquina.

Na Figura 7 é apresentado o ganho em desempenho da máquina com a adição de novos anéis em sua estrutura. Com base nesses resultados, é possível observar uma escalabilidade promissora, suportando a hipótese inicial deste projeto. Apesar do ganho em desempenho ter sido avaliado sob a perspectiva de uma granularidade ligeiramente mais grossa, a partir de uma aproximação destes resultados, é possível estimar o ganho em desempenho sob a ótica da fina granularidade do conjunto de instruções da arquitetura, mantida em cada anel de sua estrutura. Nessa perspectiva, o *speedup* chega próximo a 30 com o uso de 40 unidades funcionais, resultado que é consistente com os que eram obtidos pela MDFM [Bohm and Gurd 1990] e com aproximações previstas nas versões anteriores do presente projeto [Silva Junior 2016].

Para validar a aproximação do ganho em desempenho sob a perspectiva da fina granularidade do conjunto de instruções de cada anel, foi analisado ainda o ganho em desempenho da máquina como função da quantidade de unidades funcionais da solução. No gráfico da Figura 8, foram tomadas as melhores medidas obtidas em simulação para diferentes quantidades de anéis. Note que o ganho máximo em desempenho encontra-se ao redor de 30, assim como a aproximação obtida nos resultados anteriores. Adicionando-se mais de 40 unidades funcionais, obtém-se pouco ganho em desempenho, mas esse resultado é esperado devido à relação entre o tamanho do programa em execução e os gargalos naturais da arquitetura [Nussbaum and Agarwal 1991]. Assim, conclui-se que estamos próximos dos limites para ganho em desempenho para este programa específico. No entanto, a estrutura proposta é capaz de suportar até 100 unidades funcionais, 5 vezes mais que o limite de 20 unidades da MDFM original. Esse fato reforça a viabilidade do método para a exploração massiva de escalabilidade.

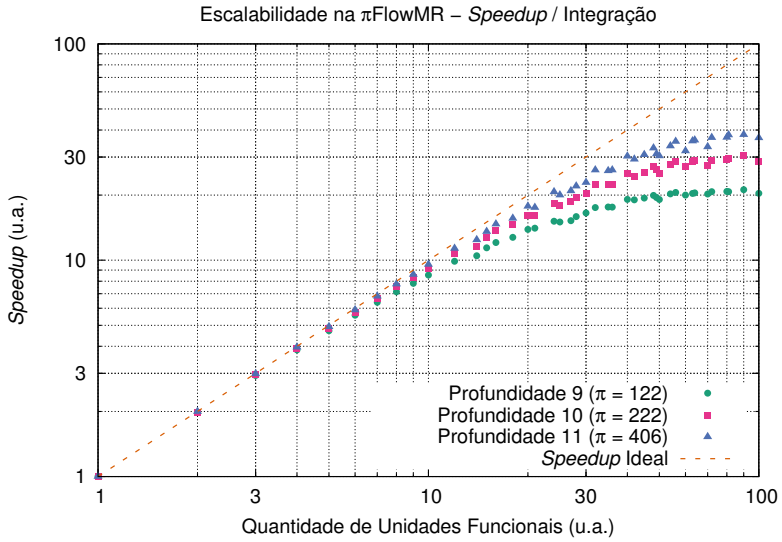


Figura 8. Gráfico do desempenho da πFlowMR sobre a perspectiva da fina granularidade do conjunto de instruções, como função da quantidade de unidades funcionais na estrutura da máquina.

5. Conclusões

Os resultados obtidos até o momento suportam a viabilidade do Multianel para explorar o conceito de escalabilidade sobre a arquitetura da MDFM. Aparentemente, a proposta apresenta-se realmente capaz de atingir um ganho real com relação a arquiteturas baseadas no paradigma do fluxo de dados publicadas anteriormente na literatura. Uma análise inicial mostrou que a πFlowMR apresenta um *speedup* bastante superior ao de arquiteturas convencionais com execução paralela. Seguimos bastante motivados para verificar todo o potencial que o paradigma do fluxo de dados é capaz de oferecer, mesmo após tantos anos de sua proposta inicial. A implementação dessa arquitetura em placas de FPGA amplamente acessíveis reforça sua utilidade como acelerador em projetos de computação heterogênea.

6. Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001. Agradecemos ainda ao grupo CIERMag, do Instituto de Física de São Carlos, que oferece recursos para o desenvolvimento deste projeto.

Referências

Arvind and Culler, D. E. (1986). Annual review of computer science vol. 1, 1986. chapter Dataflow Architectures, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA.

Barahona, P. M. C. C. (1984). Performance evaluation of a multi-ring dataflow machine. Master's thesis, University of Manchester, Manchester.

Barahona, P. M. C. C. and Gurd, J. R. (1986). Processor allocation in a multi-ring dataflow machine. *Journal of Parallel and Distributed Computing*, 3:305–327.

Bohm, A. P. W. and Gurd, J. R. (1990). Iterative instructions in the Manchester dataflow computer. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):129–139.

Böhm, A. P. W. and Sargeant, J. (1989). Code optimisation for tagged-token dataflow machines. *IEEE Transactions on Computers*, 38(1):4–14.

da Silva, J. G. D. and Watson, I. (1983). Pseudo-associative store with hardware hashing. *IEE Proceedings E - Computers and Digital Techniques*, 130(1):19–24.

Dennis, J. B. (1980). Data flow supercomputers. *IEEE Computer*, 13(11):48–56.

Flynn, M. J., Pell, O., and Mencer, O. (2012). Dataflow supercomputing. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–3.

Gurd, J. R. (1985). The Manchester dataflow machine. *Computer Physics Communications*, 37(1):49–62.

Gurd, J. R., Kirkham, C. C., and Watson, I. (1985). The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52.

Kish, L. B. (2002). End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3-4):144–149.

Magna, P. (1997). *Proposta e Simulação de uma Arquitetura a Fluxo de Dados de Segunda Geração*. PhD thesis, Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos.

Markov, I. (2014). Limits on fundamental limits to computation. *Nature*, 512:147–54.

Matzke, D. (1997). Will physical scalability sabotage performance gains? *Computer*, 30(9):37–39.

Nikhil, R. (2004). Bluespec SystemVerilog: efficient, correct RTL from high level specifications. Waltham: IEEE Computer Society.

Nikhil, R. S. and Arvind (2009). What is Bluespec? *SIGDA Newsletter*, 39(1):1–1.

Nussbaum, D. and Agarwal, A. (1991). Scalability of parallel machines. *Commun. ACM*, 34(3):57–61.

Sargeant, J. and Ruggiero, C. A. (1987). Control of parallelism in the Manchester dataflow machine. *Lecture Notes in Computer Science*, 274:1–15.

Sharp, J. A. (1992). *Dataflow computing*. Ablex Publishing Company, New York.

Silva Junior, J. T. (2016). PiFlow - projeto, simulação e implementação de um protótipo dataflow em FPGA. Master's thesis, Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos.

Veen, A. H. (1986). Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396.

Avaliação de Desempenho do Montador DALIGNER em Arquiteturas Manycore

Evaldo B. Costa[1], **Gabriel P. Silva**[1], **Marcello G. Teixeira**[1]

[1]Departamento de Ciência de Computação – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brazil

{ebcosta,gabriel,marcellogt}@dcc.ufrj.br

Abstract. *In bioinformatics, the DNA sequence assembly refers to the reconstruction of an original DNA sequence by the alignment and merging of fragments that can be obtained from several sequencing methods. The main sequencing methods process thousands or even millions of these fragments, called read sequences, which can be short (hundreds of base pairs) or long (thousands of base pairs). This is a highly compute-intensive task, which requires the use of programs and parallel algorithms so they can be performed within adequate limits of time and accuracy. In this work, we evaluate the performance of the parallel assembler program DALIGNER that assembles DNA from long read sequences. The data used for the performance evaluation were obtained from the sequencing of Escherichia coli bacteria (str. K-12 substr MG1655). The assembler performance was evaluated in an architecture with an Intel Xeon E5-2680 v2 multicore processor and also using the Intel Xeon Phi 7210 accelerator.*

Resumo. *Em bioinformática, a montagem de sequências de DNA refere-se à reconstrução de uma sequência original a partir do alinhamento e fusão de fragmentos que podem ser obtidos a partir de diversos métodos de sequenciamento. Os principais métodos de sequenciamento processam milhares ou até milhões desses fragmentos, chamados de sequências de leitura, que podem ser curtas (com centenas de pares de base) ou longas (com milhares de pares bases). Isso é uma tarefa com alta demanda computacional, que requer o uso de programas e algoritmos paralelos para ser concluída dentro de limites adequados de tempo e acurácia. Neste trabalho, avaliamos o desempenho do programa montador paralelo DALIGNER que faz a montagem de DNA a partir de sequências de leituras longas. Os dados utilizados para a avaliação de desempenho foram obtidos do sequenciamento da bactéria Escherichia coli (str. K-12 substr. MG1655). O desempenho do montador foi avaliado em uma arquitetura com o processador multicore Intel Xeon E5-2680 v2 e também com o uso do acelerador Intel Xeon Phi 7210.*

1. Introdução

Alguns dos grandes desafios relacionados aos montadores de sequências de DNA são o tempo e a quantidade de recursos computacionais necessários para a execução do processo de montagem desses genomas. Com os avanços nos sistemas computacionais, que passaram a ter maior poder de processamento, memória e armazenamento de dados, os programas de montagem passaram a utilizar de forma mais eficiente esses recursos. Sistemas paralelos com alto poder computacional, aliados a montadores de sequência fazendo uso da

programação paralela, são cada vez mais usados para o processamento de grandes quantidades de dados obtidos como saída dos sequenciadores de DNA [Cantacessi et al. 2012] [Costa et al. 2015].

Quando o processamento de sequenciamento de DNA teve início, o volume de dados obtidos nos processos de sequenciamento era bastante reduzido e apresentava um crescimento lento, em função das tecnologias então empregadas. Recentemente porém, surgiu uma nova tecnologia para o sequenciamento de DNA com leituras longas, que é muito promissora em relação à qualidade da montagem obtida, superando àquela obtida com a tecnologia dominante até o momento, de sequências de leitura curtas. Essa tecnologia contudo, deve processar arquivos que contém leituras com um percentual elevado de erros (até 15%) [Sovic et al. 2015], exigindo uma grande quantidade de repetições dessas leituras e, consequentemente, um aumento proporcional no volume de dados e no seu tempo de processamento.

Este trabalho tem como objetivo realizar um estudo do desempenho do montador de leituras longas DALIGNER [Myers 2014]. Para a execução dos testes foi utilizado um conjunto de sequências Escherichia coli str. K-12 substr. MG1655.

2. Trabalhos Relacionados

O processo de alinhamento e montagem de grandes genomas tem crescido nos últimos anos com o desenvolvimento de novos programas e tecnologias de sequenciamentos [Mardis 2008]. Esses programas conseguem executar de forma mais eficiente os processos de alinhamento e montagem, consumindo menos recursos computacionais.

O BLASR (Basic Local Alignment with Successive Refinement) é um programa utilizado para fazer o alinhamento de arquivos de sequências de de SMS (Single Molecule Sequencing) que são de leituras longas [Chaisson and Tesler 2012]. Foi originalmente projetado como uma ferramenta para mapear leituras longas para um genoma de referência.

O BLASR combina as estruturas de dados usadas no mapeamento de leitura curta com métodos de alinhamento usado em alinhamento de genomas inteiros. A estratégia utilizada para o mapeamento de leituras de SMS é localizar um número relativamente pequeno de intervalos, onde a leitura pode ser mapeada e, em seguida, usa alinhamentos detalhados para determinar qual o melhor intervalo.

Outro programa é o MECAT [Xiao et al. 2016]. É um método de alinhamento baseado em uma pontuação diferente de alinhamento global. Para grandes dados de SMS humanos, este método é 7 vezes mais rápido do que o MHAP para alinhamento em pares e 15 vezes mais rápido do que BLASR para mapeamento de referência.

Ele é capaz de produzir montagem *de novo* com alta qualidade de grandes genoma, a partir de leituras de SMS (single molecular sequencing) com baixo custo computacional, utilizando uma quantidade menor de memória e processamento.

O montador DALIGNER de sequências de leitura longas encontra sobreposições e alinhamentos locais nos conjuntos de dados sequenciados de maneira rápida e eficiente [Myers 2014]. O processo de execução do DALIGNER é dividido em duas etapas. Durante a primeira parte ele procura por *kmers* comuns presentes na sequência de referência e destino.

Durante a execução da segunda parte do processo o alinhamento é melhorado entre a referência e o destino, através do uso de um algoritmo $O(nd)$. Na Seção 3 apresentaremos mais detalhes sobre o programa.

Em termos de avaliação de montadores de sequências de DNA, diversos estudos foram publicados, alguns com foco na qualidade e outros também abordando aspectos de desempenho.

Essas ferramentas especializadas em bioinformática diferem não apenas em seus projetos e metodologia algorítmica, mas também em sua robustez de desempenho em uma variedade de conjuntos de dados, tempo e eficiência do uso de memória e escalabilidade. Alguns genomas que consumiam grande quantidade de recursos computacionais, com o uso das novas técnicas de sequenciamento e dos novos programas desenvolvidos, passaram a utilizar uma menor quantidade de recursos computacionais.

Neste estudo avaliamos especificamente o montador DALIGNER, utilizando como dados de entrada o sequenciamento da bactéria Escherichia coli, str. K-12, substr. MG1655. Nosso objetivo é analisar e comparar o desempenho do montador DALIGNER tanto em uma arquitetura *multicore* convencional (Intel Xeon E5-2680 v2) como também com o uso de aceleradores *manycore* (Intel Xeon Phi 7210).

3. Montador DALIGNER

O montador DALIGNER de sequências de leitura longas encontra sobreposições e alinhamentos locais nos conjuntos de dados sequenciados de maneira rápida e eficiente [Myers 2014]. O DALIGNER foi o primeiro programa projetado para o alinhamento de leitura versus leitura para sequências do Pacbio.

Todos os alinhamentos locais significativos entre os arquivos de sequências de leituras longas são localizados. O DALIGNER também pode ser usado como um mapeador de leitura geral, e uma ferramenta de comparação de entre as sequências, uma vez que uma ”leitura” pode agora ser uma sequência de DNA.

Para determinar a similaridade entre duas seqüências, o algoritmo encontra um conjunto de k-mers em cada leitura, ordena cada conjunto de acordo com esses k-mers e mescla os k-mers comuns em um novo conjunto. Esses k-mers são posteriormente estendidos para encontrar o alinhamento local usando um algoritmo de ”ondas” progressivas de pontos de alcance mais distante. Este algoritmo é baseado em um algoritmo $O(nd)$ descrito em [Myers 2014], de modo que na prática um alinhamento é detectado em tempo linear com o número de colunas existentes no alinhamento.

Assim como o BLASR, se baseia no mesmo conceito de filtragem, porém utilizando o algoritmo *radix sort* [Cormen 2009] otimizado onde cada número é considerado como um vetor e tem um parâmetro livre para ser definido empiricamente durante a execução. A segunda parte melhora o alinhamento entre a referência e o destino.

O DALIGNER implementa muitas técnicas para melhorar sua eficiência. Ele se utiliza da hierarquia de memória cache de 3 níveis dos computadores modernos para obter operações de busca de memória mais rápidas, e a paralelização no nível do *threads* é implementada para a melhorar a eficiência no tempo de execução do programa.

4. Ambiente de Testes

Os testes foram realizados em dois servidores. Um servidor com dois processadores *multicore* Intel Xeon E5-2680 v2 (2,8 GHz, 10 núcleos cada, 25 MB de cache), com 128 GB de memória, doravante denominado Sistema XEON. O segundo servidor utilizando o acelerador Intel Xeon Phi 7210 com 96 GB de memória, doravante denominado Sistema PHI. Em ambos servidores o arquivo usado nos testes foi armazenado em disco local do tipo SSD (Solid-State Drive). O sistema operacional utilizado nos servidores foi o GNU/Linux CentOS de 64 bits versão 7.2.

4.1. Sistema XEON

O processador *multicore* da Intel E5-2680 é um processador com 10 núcleos que permite melhorar bastante o desempenho de um sistema computacional, além disso possui tecnologias de controle de velocidade e consumo para cada núcleo individualmente. Possui aplicação em sistemas para uso como servidor de aplicações, banco de dados e servidor de arquivo. Também é utilizado em soluções de computação de alto desempenho.

Na Figura 1 e Tabela 1 são mostradas a arquitetura e características técnicas do Sistema XEON utilizado nos testes.

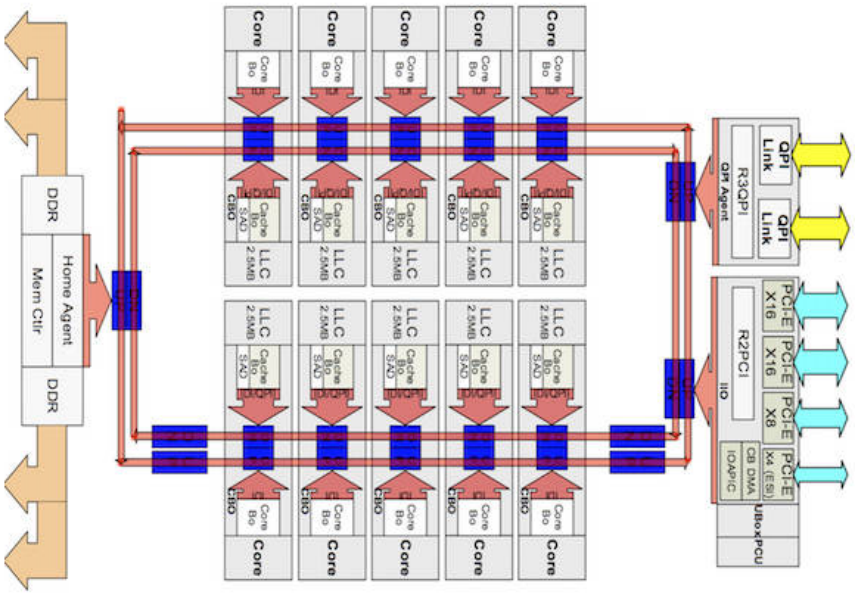


Figura 1. Arquitetura Intel Xeon E5-2680

Tabela 1. Especificações técnicas do Sistema XEON

Especificações	
Produto	Intel Xeon E5-2680 v2
Frequência	2,80 GHz
Número de núcleos	2 x 10
Cache	25 MB SmartCache
Memória	128 GB
Disco SSD	480 GB

4.2. Sistema PHI

Arquitetura do Intel MIC (Many Integrated Core) é voltada para a computação de alto desempenho (HPC), que utiliza grandes demandas de dados para processamento paralelo em uma variedade de áreas, tais como a Física Computacional, Química, Biologia e ainda Finanças [Rahman 2013]. O coprocessador é suportado por um ambiente de desenvolvimento que inclui diversos produtos, como compiladores, diversas bibliotecas, ferramentas de *tuning* e depuradores. Na Figura 2 e Tabela 2 são mostradas a arquitetura e características técnicas do Sistema PHI.

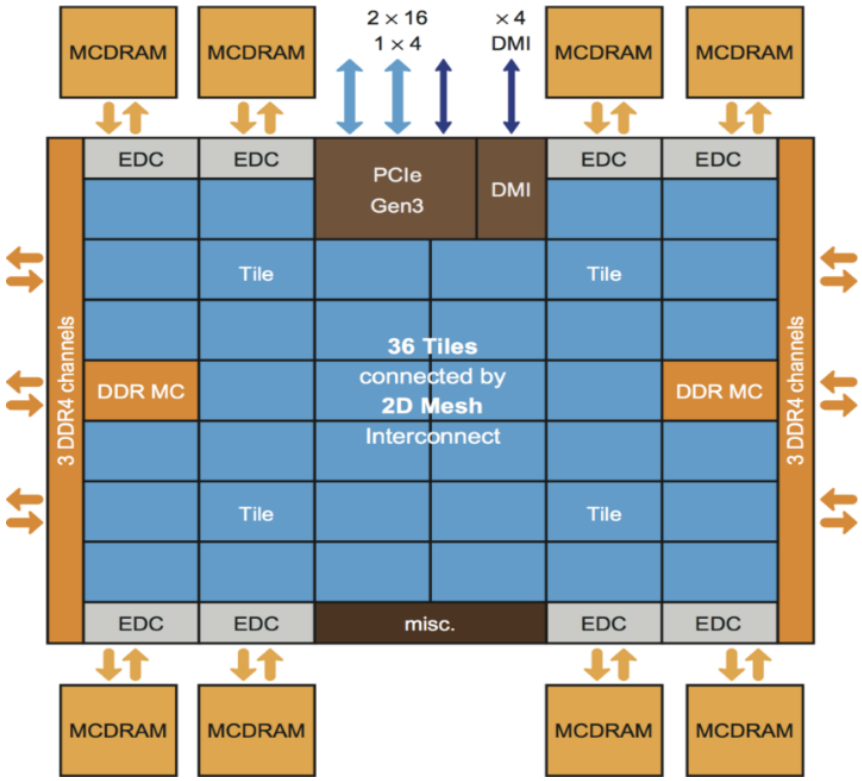


Figura 2. Arquitetura Intel Xeon Phi 7210

Tabela 2. Especificações técnicas do Sistema PHI

Especificações	
Produto	Intel Xeon Phi 7210 ”Knights Landing”
Frequência	1,30 GHz
Número de núcleos	64
Cache	32 MB L2
Memória	96 GB
Disco SSD	480 GB

4.3. Dados Utilizados

Para a execução dos testes foram utilizadas sequências de leituras longas obtidas a partir do sequenciamento do *Escherichia coli* mais conhecida por E. coli, é uma bactéria bacilar Gram-negativa que se encontra normalmente vivem nos intestinos de pessoas e

animais. A maioria das estirpes de E. coli são inofensivas, no entanto, algumas E. coli são patogênicas, o que significa que podem causar doenças. Os tipos de E. coli que podem causar doenças podem ser transmitidos através de água ou alimentos contaminados, ou através do contato com animais ou pessoas.

O arquivo utilizado para a execução dos testes foi baixado do banco de genomas do NCBI (National Center for Biotechnology Information) através do endereço https://www.ncbi.nlm.nih.gov/nuccore/U00096.3

O sequenciamento do E. Coli str. K-12 substr. MG1655 é composto por arquivos fastq com o tamanho total de 700 GB, os arquivos foram sequenciados a partir de um PacBio RS II, o sequenciador PacBio RS II é o primeiro sequênciador de DNA de leitura longas [Eid et al. 2009].

Como o montador DALIGNER não utiliza arquivos no formato fastq foi necessário utilizar o programa seqtk para converter os arquivos do formato fastq para fasta. O arquivo fastq é um formato baseado em texto para armazenar uma sequência de nucleotídeos e seus índices de qualidade correspondentes. Esse formato foi desenvolvido pelo Wellcome Trust Sanger Institute. Já o formato fasta é baseado em texto para representar uma sequência de nucleotídeos, utilizando-se uma sequência de letras. Os arquivos no formato fasta são mais fáceis de manipular e de analisar as sequências contidas neles usando-se ferramentas de processamento de texto e linguagens de script. Com isso os arquivos sequenciados puderam ser montados.

5. Resultados

Para os resultados apresentados neste estudo, foram executadas três séries de testes, variando-se a quantidade de *threads* usadas. Após cada execução, o tempo médio das séries foi calculado para definir o *speedup* e a eficiência obtida em cada caso.

5.1. Montador DALIGNER

Para a execução dos testes o montador DALIGNER foi compilado no Sistema XEON usando o compilador `GNU GCC` versão 5.4.0, com as configurações padrão conforme especificadas no arquivo `Makefile` do montador.

No Sistema PHI foi utilizado o compilador `Intel icc` versão 17.0.4. Para executar a compilação do montador, o arquivo `Makefile` foi alterado com os parâmetros necessários para uso do compilador Intel.

Os parâmetros do montador que foram alterados durante a execução dos testes foram: -M que especifica a quantidade de memória utilizada e -T que determina a quantidade de *threads*. Para os demais parâmetros do DALIGNER foram mantidos com os valores padrão.

Para identificar as rotinas que consomem mais recursos computacionais na execução do DALIGNER foi utilizada a ferramenta `Intel VTune Amplifier XE 2017`. A partir disso verificou-se que a função *Local_Alignment* consome em média 83% (veja a Tabela 3) do tempo de execução do DALIGNER.

Durante o processo de execução o DALIGNER tem duas fases principais: a primeira parte ele procura por *kmers* comuns presentes na sequência de referência e destino; a segunda

Tabela 3. Utilização de CPU durante o processo de execução do DALIGNER

Função	Uso de CPU (%)	Módulo
merge_thread	4%	daligner
lex_thread	13%	daligner
Local_Alignment	83%	daligner

parte melhora o alinhamento entre a referência e o destino, através do uso de um algoritmo $O(nd)$.

A rotina Local_Alignment procura os alinhamentos locais segundo o algoritmo descrito na Seção 3. A qualidade do alinhamento é determinada por uma série de parâmetros, tais como a correlação média $(1 - 2 * taxa\_erro)$ para os alinhamentos procurados. Para o dados Pacbio esse valor é de 0,70 assumindo uma taxa média de erro de 15% em cada leitura; o intervalo de espaçamento para guardar os pontos de *trace* e as diferenças de segmento; as frequências de ocorrência de A, C, G e T, obtidas a partir do arquivo de dados; outros parâmetros para controlar o término do alinhamento.

A Figura 3 sumariza os resultados gerado pelo Intel VTune.

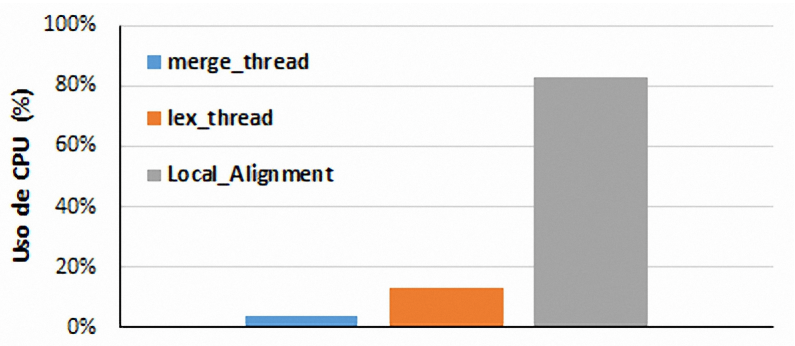


Figura 3. Utilização de CPU pelas funções do DALIGNER

Embora esta rotina ocupe uma grande parte do tempo de execução do montado, as oportunidades de exploração paralela são reduzidas, por este motivo foi utilizado a exploração do uso de *threads* como serão apresentadas nas seções seguintes.

5.2. Sistema XEON

No sistema utilizado para testes havia um total de 20 núcleos, distribuídos em 2 processadores. O gráfico da Figura 4 apresenta os valores de *speedup* e eficiência obtidos variando a quantidade de *threads* para a execução do montador.

Como pode ser observado os valores de *speedup* e eficiência tiveram uma média de valores de 7,65 e 0,81 respectivamente, que é um valor bastante interessante para este tipo de arquitetura.

A Tabela 4 mostra os tempos obtidos durante o processo de montagem entre 1 e 20 *threads*, além do *speedup* final de 13,96 e uma eficiência de 0,70 para a execução do programa com 20 *threads*.

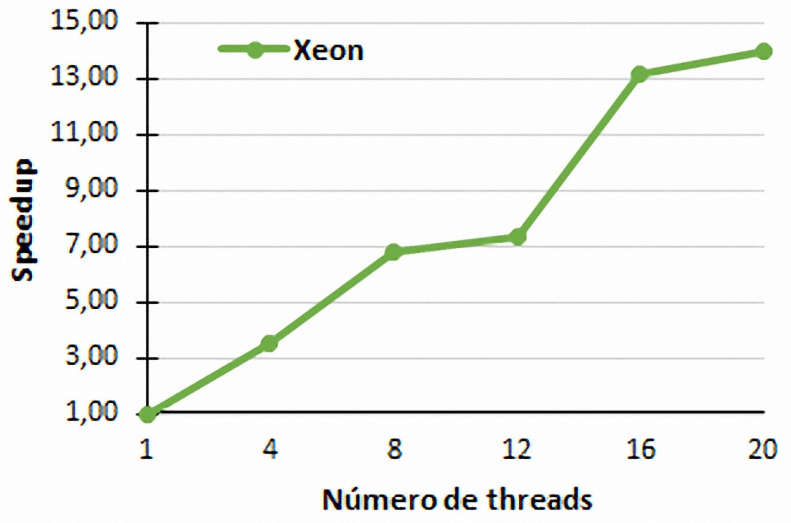


Figura 4. Speedup Sistema XEON

Tabela 4. Sistema XEON

Arquitetura	Threads	Tempo (seg)	Speedup	Eficiência
Xeon	1	670	1,00	1,00
	4	188	3,56	0,89
	8	98	6,84	0,85
	12	91	7,36	0,61
	16	51	13,14	0,82
	20	48	13,96	0,70

5.3. Sistema PHI

O Sistema PHI apresentou uma curva de *speedup* bastante consistente, com um eficiência de 0,80 com 32 *threads*. A partir daí a curva tem um mudança de inclinação até terminar com uma eficiência de 0,48 com 60 *threads*. Acreditamos que esta perda de desempenho seja dada pela limitação de memória cache desta arquitetura e pelo saturamento do seu barramento interno de comunicação.

Na Figura 5, o resumo do desempenho do Sistema PHI de acordo com o número de threads utilizadas no processamento.

Na Tabela 5, podemos ver o tempo de execução para o processamento dos dados do genoma entre 1 e 60 *threads* e os respectivos valores de *speedup* e eficiência. Como pode ser observado os valores de *speedup* e eficiência tiveram uma média de valores de 13,41 e 0,85 respectivamente.

A média de valores de *speedup* e eficiência obtidos usando o Sistema PHI foi melhor que os valores obtidos usando o Sistema XEON.

5.4. Sistema XEON versus Sistema PHI

O desempenho do montador no Sistema XEON e no Sistema PHI foi bastante similar quando o número de *threads* variou entre 1 e 8. Quando foram utilizadas mais de 8 *threads* o desempenho do Sistema PHI em termos de *speedup* foi consistentemente melhor que o

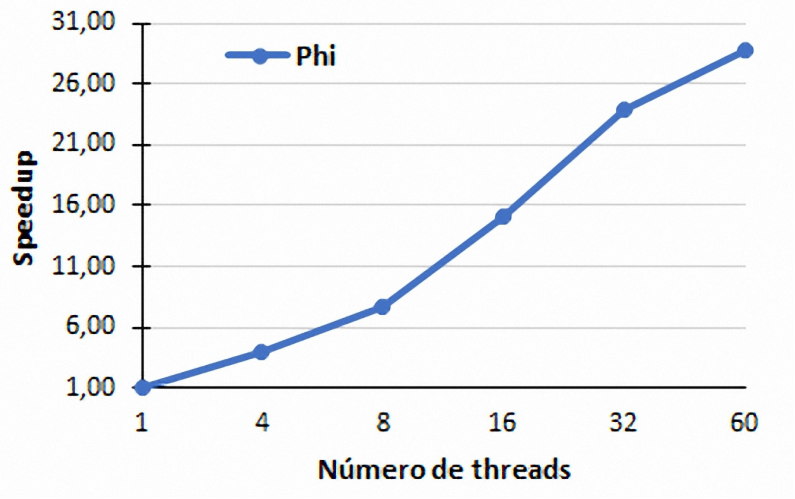


Figura 5. speedup Sistema PHI

Tabela 5. Sistema PHI

Arquitetura	Threads	Tempo (seg)	Speedup	Eficiência
Phi	1	2195	1,00	1,00
	4	550	3,99	1,00
	8	285	7,70	0,96
	16	146	15,03	0,94
	32	92	23,86	0,80
	60	76	28,88	0,48

Sistema XEON, embora inferior em tempos absolutos.

As Figura 6 e Figura 7 sumarização esta comparação entre ambos sistemas.

Como pode ser visto nas tabelas e gráficos, o Sistema XEON obteve melhores tempos de execução em relação ao Sistema PHI. Um dos motivos para o Sistema XEON ter melhores tempos estão relacionados a uma velocidade maior no relógio do processador (2,8 GHz x 1,2 GHz) e a memória cache por núcleo (1,25 MB x 0,5 MB) maior do que as do Sistema PHI.

Para melhorar o desempenho do programa DALIGNER usando o acelerador Intel Xeon Phi 7210 é necessário fazer alterações no código para adequar o programa as características de clock e cache do acelerador. As características do código do montador atualmente favorecem o melhor desempenho em sistemas *multicore*, como o Sistema XEON usado nos testes.

5.5. Uso de Memória

Outro fator importante analisado foi a quantidade de memória consumida pelo montador. Para analisar a utilização de memória durante o processo de montagem foi usado o comando do GNU/Linux ”smem”, o qual exibe o uso da memória física para cada processo em execução.

Na Figura 8 são observados os resultados do uso de memória pelo montador em ambos os servidores.

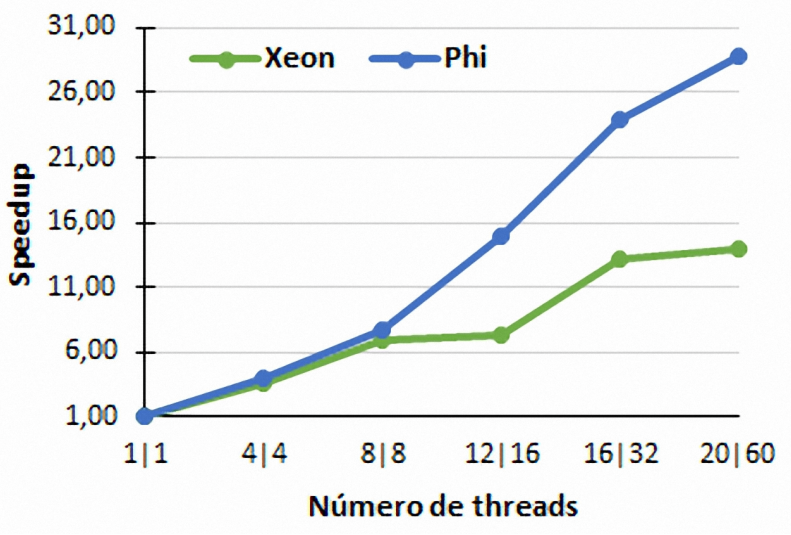


Figura 6. Speedup Sistema XEON e Sistema PHI

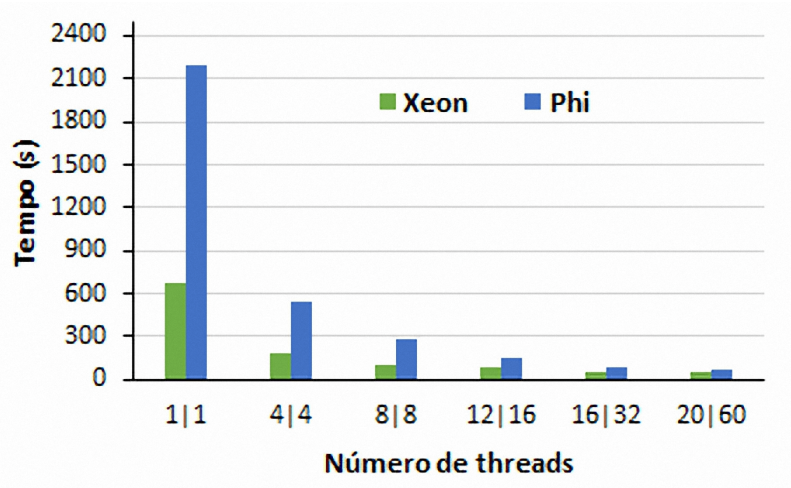


Figura 7. Tempo de execução Sistema XEON e Sistema PHI

O parâmetro *-M 96* de uso máximo de memória foi passado para o montador, o que equivale ao valor de 96 GB para ambos os servidores. Com isso o uso de memória ficou padronizado tanto no servidor *multicore* Intel Xeon E5-2680 v2, quanto no servidor com acelerador Intel Xeon Phi 7210.

Podemos observar no gráfico da Figura 8 que o total de memória consumida nas diversas etapas da computação foram as mesma em ambos os sistemas.

5.6. Qualidade Montagem

A qualidade da montagem não foi usada como parâmetro para avaliar o montador usado neste estudo. Existem outros artigos em que a qualidade de montagem e avaliada como um parâmetro do estudo. No artigo publicado por Berlin et al [Berlin et al. 2014] são comparados os resultados da qualidade dos montadores utilizados em seu estudo, entre eles o DALIGNER.

Essas ferramentas especializadas em bioinformática diferem não apenas em seus projetos

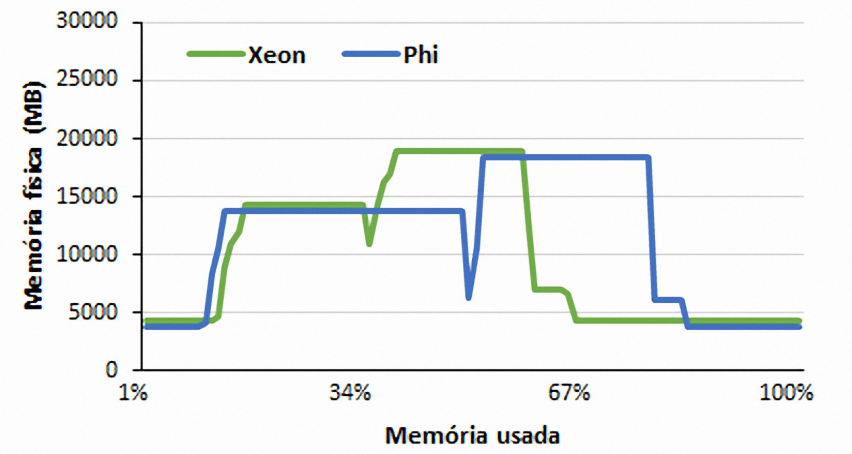


Figura 8. Uso de memória no Sistema XEON e Sistema PHI. O eixo x mostra o percentual de memória usada durante o processo de execução do DALIGNER

e metodologia algorítmica, mas também em sua robustez de desempenho utilizando uma variedade de conjuntos de dados, tempo, eficiência de memória e escalabilidade. Em seu estudo Chu et al [Chu et al. 2016] fez uma análise das atuais ferramentas de sobreposição de leitura para leitura para leituras longas, que incluem erros, incluindo BLASR, DALIGNER, MHAP, GraphMap e Minimap.

6. Conclusão e Trabalhos Futuros

Neste estudo apresentamos uma comparação do desempenho do programa montador paralelo DALIGNER, que faz a montagem de DNA a partir de sequências de leituras longas, em dois sistemas com arquiteturas diferentes. O desempenho do montador foi avaliado em uma arquitetura com o processador *multicore* Intel Xeon E5-2680 v2 e em outra com o uso do acelerador Intel Xeon Phi 7210. Para essa avaliação de desempenho foram utilizados os dados obtidos a partir do sequenciamento da bactéria Escherichia coli (str. K-12 substr. MG1655).

A contribuição desse trabalho é quantificar com mais precisão as diferenças de tempo de execução, *speedup* e eficiência de sistemas baseados no acelerador Intel Xeon Phi 7210 em relação aos servidores com o processador *multicore* Intel Xeon.

Em relação ao uso de memória, como a quantidade de memória utilizada para o processo de montagem foi definida com o mesmo valor, limitado dentro do programa montador, o gasto total de memória apresentou valores muito próximos tanto no Sistema XEON como no Sistema PHI.

Em termos de tempo de processamento, verificamos que inicialmente, para um pequeno número de *threads*, o Sistema XEON tem grande vantagem sobre o Sistema PHI (2,9 vezes mais rápido com 8 *threads*), mas a medida que o número de *threads* aumenta, esta vantagem cai para 2,7 vezes com 20 *threads* até chegar a 1,5 vezes com 60 *threads*.

Como trabalhos futuros, pretendemos modificar o código do DALIGNER, particularmente na rotina de alinhamento local Local_Alignment, de modo a acrescentar diretivas específicas para o acelerador Xeon Phi, buscando um *tuning* mais adequado a este tipo de arquitetura. Acreditamos que assim possamos diminuir o tempo total de execução do

montador, tornando esta arquitetura mais atrativa para os usuários do programa.

Não abordamos também neste estudo as questões relativas ao consumo de energia, que apresenta perfis melhores nas arquiteturas *manycore* do que nas arquiteturas *multicore* convencionais. Este é um tópico que também pretendemos endereçar futuramente.

Agradecimentos

Os autores agradecem ao Departamento de Ciência da Computação Universidade da Federal do Rio de Janeiro por fornecer os recursos computacionais utilizados para a realização dos experimentos apresentados neste trabalho.

Referências

Berlin, K., Koren, S., Chin, C.-S., Drake, J., Landolin, J. M., and Phillippy, A. M. (2014). Assembling large genomes with single-molecule sequencing and locality sensitive hashing. *bioRxiv*.

Cantacessi, C., Campbell, B. E., Jex, A. R., Young, N. D., Hall, R. S., Ranganathan, S., and Gasser, R. B. (2012). Bioinformatics meets parasitology. *Parasite Immunology*, 34(5):265–275.

Chaisson, M. J. and Tesler, G. (2012). Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC Bioinformatics*, 13(1):238.

Chu, J., Mohamadi, H., Warren, R., Yang, C., and Birol, I. (2016). Overlapping long sequence reads: Current innovations and challenges in developing sensitive, specific and scalable algorithms. *bioRxiv*.

Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

Costa, E. B., Silva, G. P., and Teixeira, M. G. (2015). Performance evaluation of parallel genome assemblers. In Saeed, F. and Haspel, N., editors, *Proc. of the 7th Int. Conf. on Bioinformatics and Computational Biology (BICOB 2015)*, volume 1, pages 31–38.

Eid, J., Fehr, A., Gray, J., Luong, K., Lyle, J., Otto, G., Peluso, P., Rank, D., Baybayan, P., Bettman, B., Bibillo, et al. (2009). *Science*, 323(5910):133–138.

Mardis, E. R. (2008). The impact of next-generation sequencing technology on genetics. *Trends Genet.*, 24(3):133–41.

Myers, G. (2014). *Efficient Local Alignment Discovery amongst Noisy Long Reads*, pages 52–67. Springer Berlin Heidelberg, Berlin, Heidelberg.

Rahman, R. (2013). *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition.

Sovic, I., Krizanovic, K., Skala, K., and Sikic, M. (2015). Evaluation of hybrid and non-hybrid methods for de novo assembly of nanopore reads. *bioRxiv*.

Xiao, C.-L., Chen, Y., Xie, S.-Q., Chen, K.-N., Wang, Y., Luo, F., and Xie, Z. (2016). Mecat: an ultra-fast mapping, error correction and de novo assembly tool for single-molecule sequencing reads. *bioRxiv*.

Uma Análise do Impacto das Transferências de Dados em Aplicações OpenMP 4.5 em uma GPU de Baixo Consumo

Rafael Gauna Trindade[1], **Bruno M. Muenchen**[1], **João V. F. Lima**[1]

[1] Universidade Federal de Santa Maria (UFSM) – Santa Maria – RS – Brasil

{rtrindade,bmokan,jvlima}@inf.ufsm.br

Abstract. *This paper presents an analysis of data transfers in OpenMP 4.5 applications over a low power GPU device. The methodology of this paper was to modify the OpenMP runtime from Clang compiler to support three memory management strategies: traditional, without copies with zero-copy, and without copies with unified copy. Our experimental results over a NVIDIA Jetson TX1 suggest that unified memory may reduce execution time up to 28%.*

Resumo. *Este artigo apresenta uma análise do impacto de transferências de dados em aplicações OpenMP 4.5 à uma GPU de baixo consumo. A metodologia utilizada consistiu em alterar a runtime OpenMP do compilador Clang a fim de suportar três formas de gerenciamento de memória da arquitetura CUDA: tradicional, sem cópias com zero-copy e sem cópias com memória unificada. Os resultados experimentais em uma NVIDIA Jetson TX1 demonstraram que memória unificada pode reduzir o tempo de execução em até 28%.*

1. Introdução

O desenvolvimento de tecnologias de hardware para computação de alto desempenho (HPC) segue um ritmo acelerado, e hardware energeticamente eficiente que antes era usado por consumidores têm demonstrado sucesso em HPC, como as Unidades de Processamento Gráfico (GPU), que introduziram o conceito de *General Purposes Computing on Graphics Processing Units*[1] (GP-GPU) [Nikolskiy et al. 2016].

Ao passo em que as tecnologias de hardware se desenvolvem criam-se necessidades a serem supridas através de novas tecnologias de software. Junto com a GPU surgiram tecnologias de desenvolvimento como CUDA e OpenCL. Estas tecnologias, devido a sua complexidade, aumentam o custo de desenvolvimento e manutenção, além de reduzirem a portabilidade dos softwares que as utilizam. Desta forma, iniciativas visam simplificar o desenvolvimento de software para estas plataformas. Exemplos disso são o OpenACC, uma especificação para diretivas de compiladores que possibilitam anotação em código de regiões paralelas e de dados que são carregados e executados em aceleradores [Hoshino et al. 2013], e o OpenMP, um dos padrões mais utilizados em computação paralela.

Como mencionado por [Martineau et al. 2016], APIs de baixo nível, como a CUDA, trazem consigo um alto grau de complexidade que pode se tornar desagradável para alguns desenvolvedores, criando a necessidade de um padrão portável com menos

[1]Computação de Propósito Geral em Unidades de Processamento Gráfico

barreiras. Devido a este fator, em suas versões mais recentes o OpenMP introduziu diversas diretivas para suportar computação heterogênea direcionada a dispositivos *many-core*, como as GPUs e o coprocessador Xeon Phi, por exemplo.

A utilização de GPUs para computação massivamente paralela não se resume apenas a supercomputadores, estando presente também em sistemas embarcados (MPSoC), como dispositivos da família Jetson da NVIDIA. Uma característica marcante destes sistemas é o fato de possuírem memória compartilhada entre a CPU e a GPU [Ukidave et al. 2015]. Esta característica pode ser explorada para o tratamento de uma deficiência existente em GPU, o acesso aos dados que estão na memória [Martineau et al. 2016]. Embora existam pesquisas que exploram a memória compartilhada dos MPSoCs, sendo restritas a aplicações CUDA, nenhuma delas explora essa funcionalidade em aplicações OpenMP.

Este artigo apresenta uma análise do impacto de transferências de dados em aplicações OpenMP 4.5 à uma GPU integrada de baixo consumo. A metodologia utilizada consistiu em alterar a *runtime* OpenMP do compilador Clang a fim de suportar três formas de gerenciamento de memória da arquitetura CUDA: tradicional, sem cópias com zero-copy e sem cópias com memória unificada.

As contribuições deste trabalho são:

- A análise de uma runtime OpenMP com diferentes estratégias de gerenciamento de memória: tradicional, *zero-copy* e memória unificada.
- A avaliação de aplicações científicas em OpenMP 4 em uma GPU integrada de baixo consumo: Lattice-Bolztmann D2Q9, LULESH e TeaLeaf 2D.
- Os resultados experimentais em uma NVIDIA Jetson TX1 demonstraram que memória unificada pode reduzir o tempo de execução em até 28%.

O restante deste artigo está organizado da seguinte forma. Na seção 2 são descritos os trabalhos relacionados. A seção 3 apresenta o contexto em GPUs de baixo consumo e OpenMP. A seção 4 descreve a metodologia de análise entre as diferentes estratégias e aplicações OpenMP avaliadas. A seção 5 apresenta os resultados experimentais obtidos, seguido da discussão na seção 6. Por fim, a seção 7 apresenta a conclusão e trabalhos futuros.

2. Trabalhos Relacionados

Existem diversos trabalhos na literatura a respeito da utilização da RTL do OpenMP para execução de código em aceleradores. Inicialmente, estes trabalhos eram principalmente voltados ao acelerador Xeon Phi, como apresentado por [Newburn et al. 2013]. Em seu trabalho ele apresenta uma infraestrutura de compilação para execução de *software* no acelerador Intel Xeon Phi, a qual inclui um compilador C/C++ e Fortran que capacita a execução de código neste co-processador.

No trabalho apresentado por [Bertolli et al. 2014], é proposto um método para coordenação de *threads* em uma GPU NVIDIA que seja tanto eficiente quanto facilmente integrável em um compilador. Também é discutida uma forma de integração da proposta na infraestrutura de compilação LLVM. Em um trabalho mais recente, [Bertolli et al. 2015] mostra como foi feita a integração da complexidade do controle de *loop*, descrito anteriormente, no LLVM, limitado a funcionalidades relativas ao OpenMP.

Uma análise de desempenho é apresentada por [Martineau et al. 2016], em que foi portada uma coleção de *benchmarks* compilada usando a versão 4.5 do RTL do OpenMP, do compilador Clang/LLVM. O desempenho apresentado é comparado a versão equivalente escrita em CUDA. Os testes foram executados em uma GPU NVIDIA Kepler. Um dos apontamentos do estudo é que otimizações relativas a memória compartilhada podem ser particularmente importantes para o desempenho.

Na pesquisa elaborada por [Ukidave et al. 2015] são explorados os casos de uso da Nvidia Jetson TK1 como um dispositivo de interface para computação em nuvem e como um dispositivo escalável para HPC energeticamente eficiente. É avaliado o desempenho da estrutura de memória unificada do dispositivo, o uso de energia e o desempenho. Em seu trabalho é apontado um desempenho muito superior no tempo de transferência de memória da Jetson TK1, que possui memória compartilhada entre GPU e CPU, em relação a uma GPU discreta NVIDIA Tesla K40. A explicação para estes resultados, conforme apontado no trabalho, é a constante movimentação de dados entre o dispositivo e o *host*, e entre o *host* e o dispositivo.

[Bercea et al. 2015] analisa a performance do OpenMP 4.0 em GPUs Nvidia utilizando o *benchmark* LULESH, disponibilizado pelo Departamento de Energia (DoE) dos Estados Unidos como parte do conjunto de *benchmarks* CORAL. O trabalho propõe um conjunto de construtores OpenMP para *offloading* de código em GPUs com o intuito de criar uma ferramenta que possibilite a escrita de um único código para execução em diversas plataformas, sem a necessidade de implementações específicas para cada uma.

[Otterness et al. 2017] realiza uma avaliação da eficácia da Nvidia TX1 no suporte de cargas de trabalhos de visão computacional em tempo real. Seu trabalho apresenta um estudo da TX1 como solução *multicore+GPU* para sistemas autônomos de segurança crítica em tempo real. Esta avaliação é uma parte de um projeto maior que tem o intuito de avaliar a eficácia de várias plataformas de hardware propostas para permitir a condução autônoma.

Embora diversos trabalhos abordem temas como o *offload* de aplicações para aceleradores, bem como a performance da arquitetura de memória unificada da plataforma CUDA, não há trabalhos que explorem a capacidade de utilizar tanto OpenMP quanto memória compartilhada em conjunto em sistemas em que a memória da CPU e da GPU é unificada.

3. Contexto

Esta seção apresenta conceitos relacionados ao gerenciamento de dados em CUDA para GPUs NVIDIA, detalhes sobre a placa de desenvolvimento NVIDIA Jetson TX1 e breve descrição sobre OpenMP 4.5 versão.

3.1. Gerenciamento de dados em CUDA

CUDA é uma plataforma de programação, assim como modelo de programação, para GPUs NVIDIA. A execução de código em GPU é baseada na estratégia *fork/join* com o paralelismo exposto em trechos de código de executam na GPU definidos como *kernel* e com linguagem um subconjunto de C.

A GPU é um co-processador que, em geral, possui memória própria e separada da DRAM. A estrutura básica de um programa CUDA é: alocar memória na GPU; uso

de transferências de memória da memória DRAM para GPU; carregar o programa, ou kernel, para rodar nos cores CUDA; transferir dados de saída da GPU para a DRAM; liberar memória da GPU.

Em GPUs integradas, CUDA possibilita memória *zero-copy* onde códigos CUDA usam o mesmo ponteiro, ou seja, não há transferências de dados. CUDA também suporta *Unified memory* (UVA), ou memória unificada, para GPUs de co-processador e integradas. Memória unificada é similar a zero-copy onde um único ponteiro de memória é usado na CPU e GPU. A diferença está na execução de um kernel CUDA onde o *driver* GPU transfere os dados sob demanda da memória local da CPU para a GPU. Além disso, a memória zero-copy não suporta a cache local de dados da CPU e da GPU.

3.2. NVIDIA Jetson TX1

A placa NVIDIA Jetson TX1 possui uma CPU 64-bit ARM Cortex-A57 de quatro cores, uma GPU integrada Maxwell GM20B de 256 CUDA cores e 4GB de memória RAM LPDDR4. A TX1 é uma plataforma "big.little" com um core adicional de baixo consumo ARM Cortex-A53 não acessível diretamente por software e só ativa em modo de baixo consumo.

A TX1 possui uma GPU integrada que compartilha a memória DRAM com os cores CPU, normalmente consome de 5 a 15 Watts, e requer pouca ventilação. A figura 1 ilustra os componentes da arquitetura do processador X1.

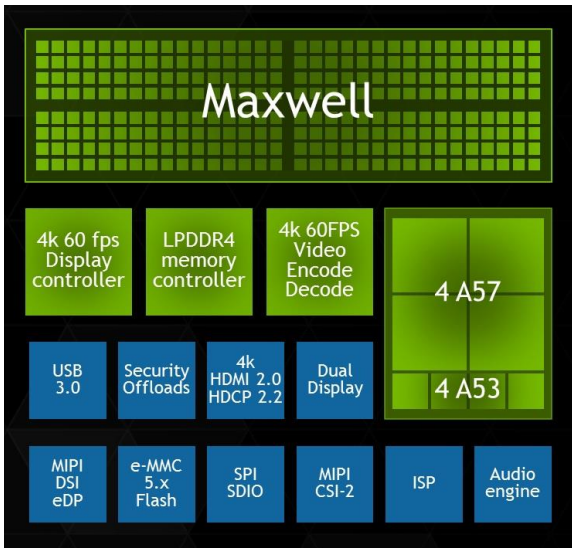


Figura 1: Arquitetura X1 com 4 cores A75 e 256 CUDA cores.

3.3. OpenMP para aceleradores

A versão OpenMP 4.0 introduziu diversas funcionalidades a fim de suportar aceleradores por meio de um modelo de programação *offloadling*. A versão OpenMP 4.5 adicionou novas funcionalidades tais como: todas as variáveis são `firstprivate` por padrão e diretivas `target data enter` e `target data exit` que possibilitam descrever um escopo de dados.

A figura 2 ilustra o exemplo de uma multiplicação $y < -\alpha x + y$ em OpenMP 4.5. A diretiva `parallel for` é semelhante a utilizada para CPUs, com a adição da diretiva de *offload* `target` e cláusula `map`. A diretiva `target` possibilita executar uma região de código em acelerador e a cláusula `map` especifica explicitamente os dados a serem transferidos ou mapeados no dispositivo.

```
#pragma omp target map(to: A[:N]) map(tofrom: B[:N])
#pragma omp parallel for
for(i = 0; i < N; i++) {
  B[i] = B[i] + A[i] * alpha;
}
```

Figura 2: Exemplo de código OpenMP para aceleradores.

4. Metodologia

Esta seção descreve a metodologia deste trabalho: alterações da runtime OpenMP para aceleradores de baixo consumo, e as aplicações dos experimentos.

4.1. Modificações da Runtime OpenMP

A runtime OpenMP do Clang foi modificada para suportar diferentes estratégias de transferências de dados baseadas nas funcionalidades do modelo CUDA. As três versões desenvolvidas foram:

Tradicional Aplicações CUDA tradicionais devem copiar dados explicitamente da CPU para GPU, além de alocar memória na GPU para tanto. Nesta versão, diretivas `target` com cláusulas `map` resultam em alocações de memória GPU e transferencias de dados conforme o tipo de acesso.

Zero-copy Em modo zero-copy, a CPU e a GPU acessam a mesma área de memória, o que evita alocações de memória e transferências de dados. Na prática, a única desvantagem é que memória zero-copy acessada pela GPU ignora os níveis de cache da GPU. Apresentações antigas e postagens em fóruns da NVIDIA mencionam o problema de cache devido ao modelo de consistência e problemas com coerência.

Unified memory (UVA) Memória unificada é similar a zero-copy onde a CPU e GPU compartilham o mesmo ponteiro de memória. Na realidade, UVA é uma abordagem híbrida entre vantagens de zero-copy e benefícios de cache com o método tradicional. A implementação do *driver* GPU garante as transferências de dados de forma transparente sob demanda entre a CPU e GPU.

A estratégia tradicional já é suportada pela runtime OpenMP, enquanto que as outras duas foram alteradas para evitar alocações de memória na GPU e transferências de dados. Além disso, adicionou-se uma função de alocação de memória em CPU `omp_alloc_host` que corresponde a alocação de memória necessária para zero-copy (`cudaMallocHost`) e UVA (`cudaMallocManaged`).

4.2. Benchmarks Científicos

4.3. Lattice-Bolztmann

O *Lattice-Boltzmann* (LBM) é um método numérico iterativo para modelar e simular propriedades da dinâmica de fluídos [Chen and Doolen 2003]. O LBM usa uma abordagem macroscópica para representar e simular o fluxo de fluídos. Este método pode demandar uma grande quantidade de memória ou de processamento, de acordo com a dimensão do problema [Schepke et al. 2009].

A implementação utilizada é baseada na versão de [Schepke et al. 2009], em que é utilizada uma estratégia de particionamento de dados em bloco, ao invés da abordagem tradicional de distribuição simples dos dados.

4.3.1. LULESH

O LULESH (*Livermore Unstructured Lagrange Explicit Shock Hydrodynamics*) foi inicialmente desenvolvido como um dos cinco desafios do programa *DARPA Ubiquitous High Performance Computing* (UHPC). O algoritmo tem sido largamente estudado e portado para modelos de programação como OpenMP, CUDA e OpenACC [Karlin et al. 2013].

A implementação OpenMP da LULESH 2.0 foi convertida por [Bercea et al. 2015] para usar as diretivas *target*, que possibilitam o *offloading* para GPU.

4.3.2. TeaLeaf

TeaLeaf é uma mini-aplicação para resolver equações de condução térmica que pertence ao projeto Mantevo[2]. Em particular, esta aplicação tem funcionalidades suficientes para expor o perfil de desempenho e complexidade computacional de um código de produção [Martineau et al. 2017].

O algoritmo faz uso de três solucionadores de matrizes esparsas, que são: a) Gradiente Conjugada; b) Chebyshev; e c) Chebyshev polinomialmente condicionada [Martineau et al. 2017]. A implementação utilizada nos experimentos é distribuída em múltiplas versões criadas a partir de uma implementação base[3], sendo que nenhuma modificação foi necessária para a realização dos testes.

5. Resultados Experimentais

O objetivos dos experimentos são:

- Analisar o desempenho de programas OpenMP 4 em aceleradores de baixo consumo.
- Avaliar o impacto da eliminação de transferências de memória à GPU com duas técnicas: *Zero-copy* e *UVA*.

[2] `https://mantevo.org/`
[3] `https://github.com/UK-MAC/TeaLeaf_ref`

Os experimentos foram realizados em uma placa NVIDIA Jetson TX1 que possui uma CPU ARM Cortex-A57 de quatro cores, uma GPU Maxwell de 256 CUDA cores e 4GB de memória RAM LPDDR4. A placa executa um sistema Ubuntu GNU/Linux 16.04, compilador Clang OpenMP 4.5 baseado em uma versão desenvolvida pela IBM[4] com adaptações à TX1, CUDA versão 8.0.

A figura 3 apresenta os resultados obtidos com diferentes métodos de memória com as três aplicações OpenMP. Os pontos apresentados são uma média de no mínimo 10 execuções com intervalo de confiança de 99% representado por meio de uma linha vertical escura em cada média.

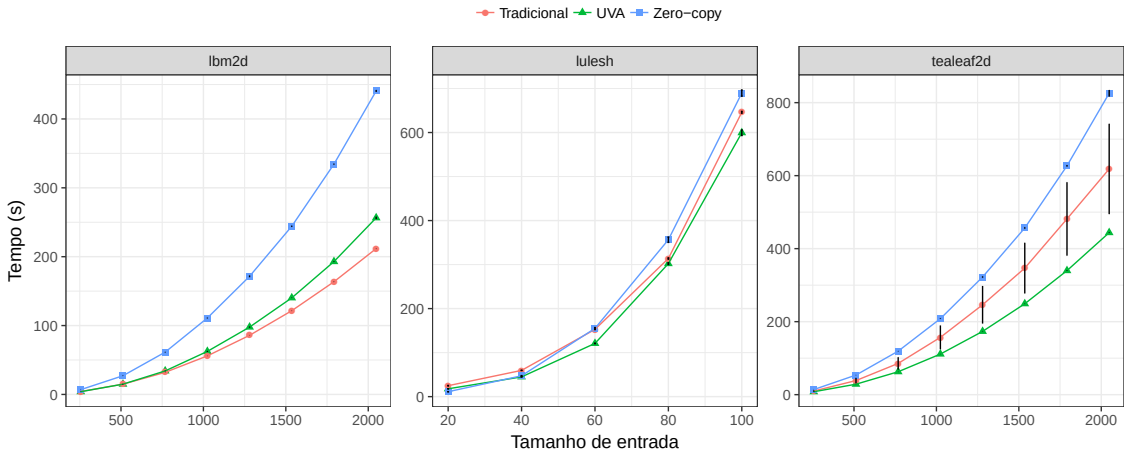


Figura 3: Resultados das aplicações OpenMP com diferentes tamanhos de entrada.

A utilização da memória UVA apresentou melhora no desempenho em dois dos três benchmarks testados. O ganho com relação a versão tradicional chegou a 7.18% com LULESH e 28.21% com o TeaLeaf. Por outro lado, o benchmark LBM apresentou redução de 21.09% ao utilizar UVA. Nota-se também a variação significativa dos tempos de execução com TeaLeaf com a versão tradicional. A versão com Zero-copy apresentou tempos de execução maiores em todos os experimentos. A redução chegou a 6.76% com LULESH, 33.53% com TeaLeaf e 108.08% com LBM.

A figura 4 mostra os resultados de rastros de execução da ferramenta *nvprof* em cada configuração de dados, onde as chamadas *target* foram agrupadas. Nota-se que entre as três aplicações, as cópias de memória são mais significativas no LULESH com 15.20% do tempo total de execução. Todavia, observou-se que as cópias nas aplicações LBM e TeaLeaf representaram menos que 1% do tempo total.

6. Discussão

As alterações na runtime do OpenMP para o compilador Clang podem beneficiar programas para aceleradores embarcados. Os resultados demonstraram que a utilização de memória unificada (UVA) pode reduzir o tempo de execução na plataforma TX1 em até 28% no caso da aplicação LULESH.

[4]`https://github.com/clang-ykt/openmp`

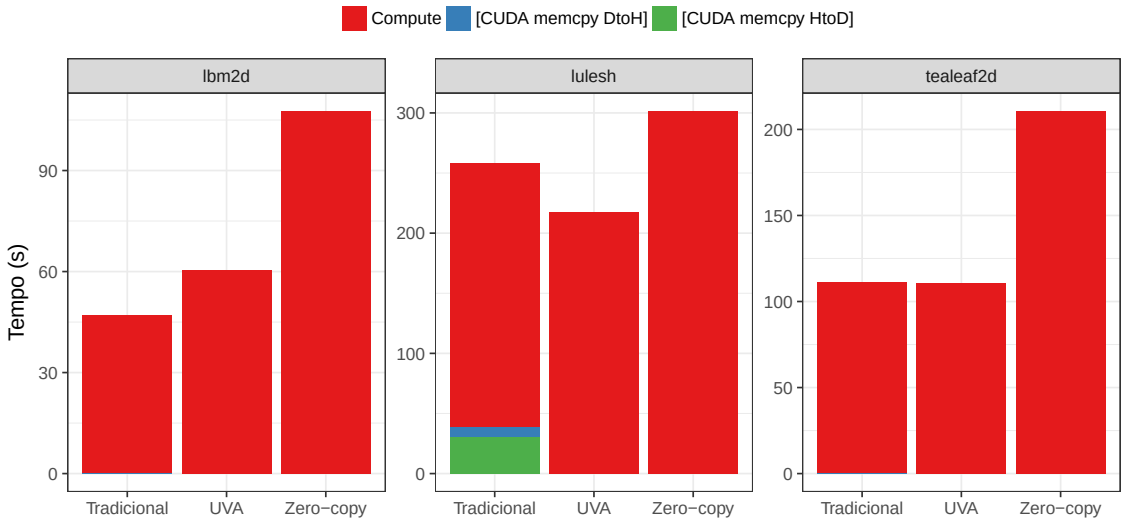


Figura 4: Tempos de computação e transferência de dados obtidos com o nvprof. Tamanho de entrada para LBM e TeaLeaf foi 1024, e LULESH 80.

Esse ganho não foi alcançado apenas na aplicação LBM. De acordo com os rastros da ferramenta *nvprof* (figura 4), observou-se que as transferencias de dados tem pouco impacto no desempenho da aplicação comparado com o custo de execução de cada kernel de computação.

Não obstante, a memória gerenciada por zero-copy aumentou o tempo de execução em todos os casos. O principal motivo desse comportamento se deve a restrição de não poder utilizar a cache da GPU com zero-copy. Além disso, esse modo de memória apresenta outras restrições quanto ao acesso de dados na GPU. Dessa forma, mesmo sem cópias de dados, memória zero-copy pode impactar o desempenho de forma negativa.

7. Conclusão

Este artigo apresentou uma análise do impacto de transferências de dados em aplicações OpenMP 4.5 à uma GPU de baixo consumo. A metodologia utilizada consistiu em alterar a *runtime* OpenMP do compilador Clang a fim de suportar três formas de gerenciamento de memória da arquitetura CUDA: tradicional, sem cópias com zero-copy e sem cópias com memória unificada.

Os resultados experimentais em uma NVIDIA Jetson TX1 demonstraram que memória unificada pode reduzir o tempo de execução em até 28%. Além disso, constatou-se que memória zero-copy afetou negativamente o desempenho das aplicações. Como trabalhos futuros, pretende-se avaliar otimizações de escalonamento e estratégias para redução de consumo de energia.

8. Agradecimentos

Agradecemos à CAPES pelo apoio recebido para o desenvolvimento deste trabalho, e o suporte da NVIDIA com a doação de uma GPU NVIDIA GTX Titan X utilizada nos experimentos preliminares deste trabalho.

Referências

Bercea, G.-T., Bertolli, C., Antao, S. F., Jacob, A. C., Eichenberger, A. E., Chen, T., Sura, Z., Sung, H., Rokos, G., Appelhans, D., and O'Brien, K. (2015). Performance analysis of OpenMP on a GPU using a CORAL proxy application. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 2:1–2:11, New York, NY, USA. ACM.

Bertolli, C., Antao, S. F., Eichenberger, A. E., Sura, K. O. B. Z., Jacob, A. C., Chen, T., and Sallenave, O. (2014). Coordinating GPU threads for OpenMP 4.0 in LLVM. *Proceedings of LLVM-HPC 2014: 2014 LLVM Compiler Infrastructure in HPC - Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 12–21.

Bertolli, C., Appelhans, D., O'Brien, K., Antao, S. F., Bercea, G.-T., Jacob, A. C., Eichenberger, A. E., Chen, T., Sura, Z., Sung, H., and Rokos, G. (2015). Integrating GPU support for OpenMP offloading directives into Clang. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, (May 2016):1–11.

Chen, S. and Doolen, G. D. (2003). Lattice Boltzmann Method for Fluid Flows. *Http://Dx.Doi.Org/10.1146/Annurev.Fluid.30.1.329*, (Kadanoff 1986).

Hoshino, T., Maruyama, N., Matsuoka, S., and Takaki, R. (2013). CUDA vs OpenACC: Performance case studies with Kernel benchmarks and a memory-bound CFD application. *Proceedings - 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013*, pages 136–143.

Karlin, I., Bhatele, A., Keasler, J., Chamberlain, B. L., Cohen, J., DeVito, Z., Haque, R., Laney, D., Luke, E., Wang, F., Richards, D., Schulz, M., and Still, C. (2013). Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA.

Martineau, M., McIntosh-Smith, S., and Gaudin, W. (2016). Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model. *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pages 338–347.

Martineau, M., McIntosh-Smith, S., and Gaudin, W. (2017). Assessing the performance portability of modern parallel programming models using tealeaf. *Concurrency and Computation: Practice and Experience*, 29(15):e4117–n/a. e4117 cpe.4117.

Newburn, C. J., Dmitriev, S., Narayanaswamy, R., Wiegert, J., Murty, R., Chinchilla, F., Deodhar, R., and McGuire, R. (2013). Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1213–1225.

Nikolskiy, V. P., Stegailov, V. V., and Vecher, V. S. (2016). Efficiency of the Tegra K1 and X1 Systems-on-Chip for Classical Molecular Dynamics. pages 682–689.

Otterness, N., Yang, M., Rust, S., Park, E., Anderson, J. H., Smith, F. D., Berg, A., and Wang, S. (2017). An evaluation of the nvidia tx1 for supporting real-time computer-

vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364.

Schepke, C., Maillard, N., and Navaux, P. O. A. (2009). Parallel lattice boltzmann method with blocked partitioning. *International Journal of Parallel Programming*, 37(6):593–611.

Ukidave, Y., Kaeli, D., Gupta, U., and Keville., K. (2015). Performance of the NVIDIA Jetson TK1 in HPC. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2015-October:533–534.

Um framework para agrupar funções com base no comportamento da comunicação de dados em plataformas multiprocessadas

Rafael R. Santos[1], **Vanderlei Bonato**[1]

[1]Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) – São Carlos – SP – Brazil

{rafarko,vbonato}@usp.br

Abstract. *The efficient clustering of an application is fundamental to use its mapping on heterogeneous platforms. The communication pattern of the application should be considered to allow the analysis of the existing bottlenecks and also the available bandwidth in the communication medium should be considered to analyze the application behavior when using an accelerator. This paper proposes an analysis of communication costs for clustering, taking into account the distribution of the data during the execution time associated with bandwidth restrictions and transmission rate. The use of this approach demonstrated an increase in performance ranging from 1,117x to 2,621x for the applications used in the experiments.*

Resumo. *O agrupamento eficiente de uma aplicação é fundamental para que seu mapeamento seja utilizado em plataformas heterogêneas. Deve-se considerar o padrão de comunicação da aplicação, para permitir a análise dos gargalos existentes e deve-se considerar a largura de banda disponível no meio de comunicação, para analisar como a aplicação se comportará ao utilizar um acelerador. Neste artigo é proposta uma análise sobre os custos de comunicação para um agrupamento, levando em conta a distribuição dos dados durante o tempo de execução associado a restrições de largura de banda e taxa de transmissão. O uso dessa abordagem demonstrou um aumento no desempenho variando de 1,117x a 2,621x para as aplicações usadas nos testes.*

1. Introdução

Muitas aplicações, denominadas de *supercomputing applications* ou *superapplications*, demandam elevado poder computacional com tendência de crescimento [Kirk and Hwu 2010]. Devido a isso, severas mudanças nas arquiteturas dos computadores têm ocorrido. Projetos passaram a utilizar paralelismo em larga escala, incluindo plataformas heterogêneas compostas por aceleradores customizados, visando o ganho de desempenho computacional [Borkar et al. 2005, Borkar and Chien 2011, Horowitz and Dally 2004].

Diferente das plataformas homogêneas, que possuem várias unidades de processamento idênticas, as plataformas heterogêneas buscam explorar as características de cada elemento de processamento de acordo com o perfil do código a ser executado. GPUs (*Graphics Processing Unit*) e FPGAs (*Field Programmable Gate Array*) são tradicionalmente utilizados como aceleradores e a escolha de qual parte do código é o mais adequado

não é uma tarefa trivial devida a complexidade associada à exploração do espaço de projeto.

Há uma série de trabalhos que evidenciam o potencial de aceleradores de acordo com o perfil do código. No trabalho de [Che et al. 2008], por exemplo, é traçado um perfil de aplicações que apresentam melhor desempenho em GPUs e em FPGAs baseado no custo de desenvolvimento, desempenho e restrições de *hardware*. Por exemplo, códigos com o perfil do método de eliminação de Gauss apresentam um desempenho melhor em GPUs pelo fato de não existir dependências no fluxo de dados, o que permite que a computação seja feita em paralelo. Por outro lado, no trabalho de [Bonato et al. 2008], por exemplo, é proposta uma arquitetura de *hardware* paralela em FPGA para o processamento de um algoritmo denominado SIFT (*Scale-Invariant Feature Transform*), que é utilizado para a extração de características de imagens. A arquitetura gerada atua como um acelerador no processamento de imagens.

Uma forma de explorar o mapeamento em plataformas multiprocessadas é por meio da análise da comunicação entre trechos de código. Esses trechos de código, usualmente associado em nível funções, podem ser agrupados de acordo com a demanda da comunicação entre eles para que posteriormente cada grupo seja mapeado ao elemento de processamento mais adequado para o perfil de código. Porém, esse agrupamento não é uma tarefa trivial, pois além de analisar o código *offline*, é preciso entender a dinâmica da movimentação dos dados durante a execução da aplicação e conhecer o potencial dos compiladores para seus processadores dedicados.

O agrupamento das funções de uma aplicação pode ser realizado manualmente ou automaticamente. Na forma manual, o usuário utiliza ferramentas de *profiling*, como **Gprof** [Graham et al. 1982] ou **Valgrind** [Nethercote and Seward 2007], para obter informações que auxiliem na identificação de gargalos da aplicação, por exemplo, quais as funções que possuem um elevado processamento, de modo que o mesmo possa agrupar as funções como desejar. Já na forma automática, o usuário insere as informações da aplicação obtidas de um *profiling* em ferramentas que retornam um agrupamento para as funções, por exemplo a ferramenta **PET**.

Entretanto, para realizar um bom agrupamento, deve-se analisar a capacidade da comunicação interna de uma plataforma heterogênea. Dependendo do meio de comunicação, o tempo de transmissão de dados pode variar, por exemplo, caso o meio de comunicação seja utilizado por muitos dispositivos, a comunicação de dados será mais lenta. A otimização desse tipo de comunicação beneficiaria a execução da aplicação agrupada, pois o meio de comunicação seria dedicado à essa aplicação.

A principal contribuição desse artigo é o desenvolvimento de um *framework* para agrupar funções de uma aplicação de acordo com o comportamento da comunicação de dados entre as mesmas durante sua execução, considerando para isso sistemas computacionais compostos por mais de uma unidade de processamento com características distintas entre si.

Este artigo está organizado da seguinte forma: na Seção 2 é apresentada uma breve contextualização sobre agrupamento e sobre o problema de pesquisa relacionado a este trabalho. Na Seção 3 são apresentados os principais trabalhos relacionados ao *framework* desenvolvido. Na Seção 4 é apresentado o *framework* desenvolvido. Na Seção 5 são

apresentados os resultados obtidos com os experimentos realizados para a validação do *framework* proposto. Por fim, na Seção 6 é apresentada a conclusão.

2. Contexto e Problema de Pesquisa

O termo agrupamento pode ser definido como a tarefa de aglomerar pontos semelhantes entre si em um mesmo grupo e separar em diferentes grupos pontos que são diferentes entre si [Berkhin 2006].

Segundo [I. Ashraf and V.M. Sima and K.L.M. Bertels 2015], uma aplicação deve ser particionada em pequenos grupos para que seja possível explorar todo o paralelismo existente em uma plataforma multiprocessada, pois cada grupo é mapeado para um núcleo disponível na plataforma, de modo a alcançar um aumento no desempenho da aplicação.

O agrupamento de uma aplicação diminui o volume de dados trocados entre suas funções, pois durante o agrupamento busca-se diminuir a comunicação entre os grupos e maximizar a comunicação interna de cada grupo, de modo que em um mesmo grupo estejam as funções que apresentam elevada troca de dados entre si.

Quando um agrupamento é realizado, deve-se verificar se o grupo encontrado é eficiente. Segundo [Ashraf et al. 2013], uma pequena parte das funções contribuem para o tempo total de execução da aplicação e entre essas funções, uma parte mais restrita de funções são responsáveis pela comunicação de dados na aplicação. Logo, é importante verificar se as funções corretas foram agrupadas.

Outra fragilidade que pode ser observada refere-se ao número de possibilidades de combinações para um agrupamento. Segundo [Zaki et al. 2014], há um total de k^n possíveis grupos, no qual *k* representa o número de grupos e *n* o número de funções da aplicação. Entretanto, o valor de k^n apresenta grupos com similaridades entre si, pois um grupo formado pelas funções A e B é similar a outro que apresenta as funções B e A. Logo, segundo [Zaki et al. 2014], o número de grupos distintos entre si é apresentado pelos *Números de Segundo Tipo de Stirling*, que é apresentado na equação 1.

$$S(n,k) = \frac{1}{k!}\sum_{t=0}^{k}(-1)^t kt(k-t)^n \tag{1}$$

Outro fator que devemos analisar é o meio de comunicação, pois se o mesmo for compartilhado, irá ocorrer uma concorrência pelo seu uso de modo a ocorrer uma variação na taxa de transferência de dados através do meio.

3. Trabalhos Relacionados

Serão destacados dois principais trabalhos que apresentam as ferramentas **MCProf** e **PET**, pois serviram como base para o desenvolvimento do *framework*.

MCProf *(Memory and Communication Profiler)* é uma ferramenta de *profile* dinâmico desenvolvida por [I. Ashraf and V.M. Sima and K.L.M. Bertels 2015] para a análise do volume de comunicação existente entre funções de uma aplicação e entre funções e a memória. É importante ressaltar que embora na literatura existam outras ferramentas de análise dinâmica. A ferramenta Valgrind [Seward and Nethercote 2005] é uma delas, porém tais ferramentas apresentam um grande *overhead* quando comparadas

às que fazem análise estática devido à carga extra de trabalho que necessária para a análise das aplicações.

PET *(Partition Evaluation Tool)* é uma ferramenta desenvolvida para a avaliação da qualidade de um agrupamento ótimo para uma aplicação, por meio da comparação de algoritmos de agrupamento [Ashraf et al. 2013]. Para encontrar o melhor agrupamento, a ferramenta utiliza cinco algoritmos, sendo que após a execução de cada um é apresentado qual o melhor agrupamento encontrado por cada algoritmo. Os algoritmos são: *Força Bruta, Busca Heurística, Busca Evolutiva, Recozimento Simulado e Busca Tabu*. No *framework* implementado foi utilizado somente o algoritmo de *Busca Evolutiva*, pois foi o único algoritmo que considerou todas as funções para realizar o agrupamento.

Através da análise dessas ferramentas e de outros trabalhos, é possível notar que nenhum trabalho considera a comunicação distribuída durante a execução de uma aplicação e o meio de comunicação existente na plataforma que irá executar a aplicação.

4. *Framework* Desenvolvido

O *framework* foi desenvolvido para agrupar funções de uma aplicação analisando o comportamento da comunicação de dados da mesma durante sua execução, considerando que serão executadas em sistemas computacionais compostos por mais de uma unidade de processamento. O *framework* foi desenvolvido no sistema operacional Ubuntu 15.04 na plataforma Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30GHz com 6.00GB de RAM, utilizando o terminal R na versão 3.1.2. Na Figura 1 é apresentado o modelo do *framework* desenvolvido.

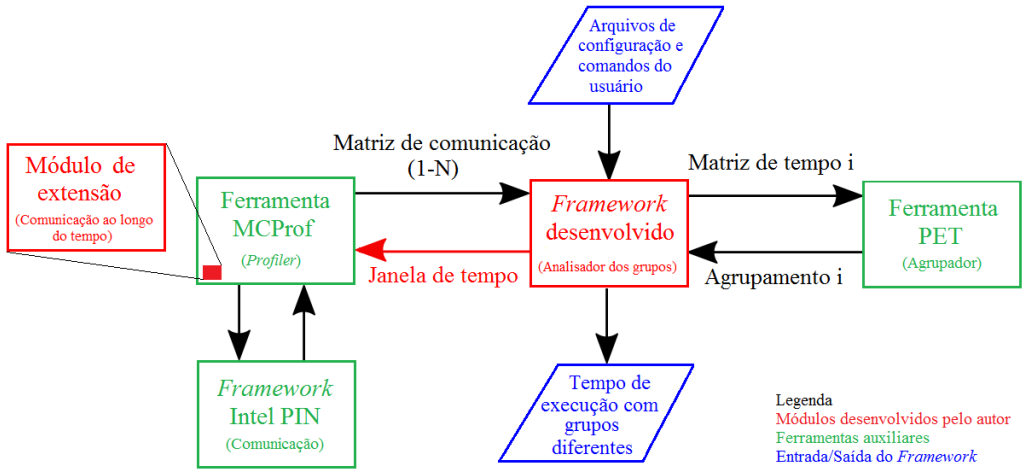


Figura 1. Esquema geral do *framework* desenvolvido.

Para inicializar o *framework*, o usuário deve criar no mesmo diretório do *framework*, dois arquivos de configuração denominados *Input.txt* e *Paths.txt*. O arquivo *Paths.txt* contém os caminhos absolutos dos executáveis das ferramentas MCProf e PET, e do diretório que está o *framework*. Já o arquivo *Input.txt* contém parâmetros referentes à aplicação a ser agrupada e ao meio de comunicação. Os parâmetros são:

- **Janela de tempo**: este parâmetro indica a janela de tempo em milissegundos em que cada matriz de comunicação será obtida através da ferramenta MCProf;

- **Número de funções**: indica o número de funções da aplicação;
- **Número de grupos**: indica o número de grupos que o usuário deseja usar para o agrupamento;
- **Largura de banda máxima**: taxa máxima de transferência permitida pelo meio de comunicação;
- **Largura de banda mínima**: taxa mínima de transferência desejável para o meio de comunicação;
- **Função de transferência**: taxa de transferência do meio de comunicação através do tempo.

Após a criação desses arquivos, basta inicializar o *framework* utilizando o terminal R, através do comando *source*, utilizado para executar um *script* na linguagem R.

Quando o *framework* é inicializado, ele lê os arquivos de configuração para em seguida realizar uma chamada de sistema para execução da ferramenta MCProf para se obter as matrizes de comunicação da aplicação. Uma matriz de comunicação representa a quantidade de dados trocados entre as funções da aplicação, na qual as funções existentes nas linhas da matriz trocam uma quantidade de dados com as funções existentes nas colunas da matriz. Na Figura 2 é apresentado um exemplo de uma matriz de comunicação com três funções F1, F2 e F3 trocando *bytes* entre si.

	F1	F2	F3
F1	0	11	8
F2	0	0	16
F3	0	0	0

Figura 2. Exemplo de matriz de comunicação de uma aplicação com 3 funções.

Para a obtenção das matrizes de comunicação através do tempo, foi desenvolvido um módulo de extensão para a ferramenta MCProf que utiliza a janela de tempo para obter as matrizes. Sempre que a aplicação é executada pelo tempo determinado no parâmetro *janela de tempo*, a MCProf retorna a matriz de comunicação daquele intervalo. Em seguida, a matriz de comunicação é zerada para acumular novamente a comunicação.

Após a MCProf retornar as matrizes de comunicação, o *framework* as converte em matrizes de tempo. Essas matrizes são similares as matrizes de comunicação, mas representam o tempo de comunicação entre funções em segundos. Para realizar essa conversão, o *framework* divide a matriz de comunicação pelo valor obtido através da função de transferência.

Antes de utilizar o valor obtido na taxa de transferência, o *framework* realizada uma checagem entre o valor e as larguras de banda máxima e mínima, para evitar que a matriz de comunicação seja dividida por um valor acima da largura de banda máxima ou por um valor menor que a largura de banda mínima. Assim, caso o valor da transferência ultrapasse os limites máximos e mínimos, o valor utilizado será o respectivo limite.

Após o *framework* adquirir as matrizes de tempo, é realizada uma chamada de sistema para a ferramenta PET, enviando uma matriz de tempo por vez, para que a ferramenta encontre um agrupamento para cada matriz. Após a PET encontrar um agrupamento para cada matriz de tempo, é enviada a matriz de comunicação acumulada para que seja encon-

trado um agrupamento utilizando apenas o volume total de comunicação, para comparar os resultados entre diferentes abordagens.

Em seguida, o *framework* analisa o custo de comunicação encontrado em cada agrupamento. Para isso, o primeiro agrupamento encontrado é aplicado em cada matriz de tempo para calcular o tempo de comunicação utilizando esse agrupamento. Essa verificação é realizada para cada agrupamento encontrado. Após isso, é testado o agrupamento encontrado utilizando o volume total de dados.

Após todos agrupamento serem analisados, o *framework* compara os tempos encontrados utilizando cada agrupamento. Ao final dessa comparação, é retornado para o usuário o tempo de comunicação do melhor agrupamento encontrado utilizando a comunicação distribuída através do tempo e o melhor tempo encontrado utilizando a comunicação acumulada.

5. Resultados

Para validar o *framework*, foram utilizadas cinco aplicações sequenciais, significando que duas ou mais funções da aplicação não acessam o meio de comunicação no mesmo intervalo de tempo. As aplicações utilizadas são **KLT** [Lucas et al. 1981], **Sparse 1.3a** [Kundert and Sangiovanni-Vincentelli 1988] e três aplicações existentes no *benchmark SPLASH-2* [Singh et al. 1992]: **Barnes** [Singh 1993], **FMM** [Greengard 1988] e **Ocean** [Woo et al. 1995].

Foi considerado que todos os grupos possuem comunicação entre si e um grupo pode enviar ou receber dados para qualquer outro grupo. Os grupos se comunicam dessa forma, pois as técnicas de agrupamento não consideram o meio de comunicação existente, somente o volume total de dados da comunicação.

Inicialmente, foi realizada uma análise considerando uma taxa de transferência constante para o meio de comunicação. Caso isso ocorra, o resultado da análise das matrizes de comunicação através do tempo será o mesmo se for utilizada a matriz de comunicação acumulada.

Nos testes foi considerada uma largura de banda máxima de 250 MB/s e uma largura de banda mínima de 50 MB/s e para emular diferentes taxas de transferências foram utilizadas três funções definidas em 2, 3 e 4. As diferentes funções de transferência foram escolhidas, pois no mundo real as transferências de dados dificilmente se comportam de maneira linear em um ambiente com recursos compartilhados.

$$f(t) = 250|sin(0,5t)|MB/s, \text{ sendo } t \geq 0, \text{ com t em radianos} \tag{2}$$

$$f(t) = 250log(t)MB/s, \text{ sendo } \{t \in \text{R} — t > 0\} \tag{3}$$

$$f(t) = 100/log(t)MB/s, \text{ sendo } \{t \in \text{R} — t > 1\} \tag{4}$$

As funções definidas para os testes simulam três tipos de transferência de dados, sendo que todas as funções, t corresponde ao tempo que o usuário define para a janela de tempo:

- Uma taxa de transferência periódica 2, que representa uma comunicação de baixa taxa em seu início, mas que cresce e em seguida decresce com o passar do tempo;
- Uma taxa crescente 3, que simula uma baixa taxa de transferência inicial que aumenta com o tempo;
- Uma taxa decrescente 4 que simula uma alta taxa de transferência inicial que decai com o tempo.

É importante destacar que a função 2 é uma senóide de meio ciclo devido ao módulo utilizado e seu período é aproximadamente T = 6 s. A função 3 não está definida para 0 devido ao fato de não existir log(0), nesse caso, o *framework* considera o valor da largura de banda mínima. Já a função 4 não está definida para 0 e 1, devido ao fato de não existir log(0) e de que log(1) = 0, impossibilitando a divisão por 0. Nesse caso, o *framework* considera o valor da largura de banda máxima.

Para cada aplicação, foram realizadas 30 execuções, considerando as funções de transferência apresentadas anteriormente, totalizando 180 resultados para cada aplicação. Além disso, para cada aplicação foi analisada tanto a comunicação acumulada quanto a comunicação distribuída através do tempo.

Para validar os resultados obtidos, foram realizados os testes de hipótese **Teste t de Student** [Haynes 2013] e **Teste de Wilcoxon** [Wilcoxon 1945] para analisar se os resultados obtidos são diferentes entre si. Esse testes de hipótese mostraram com um intervalo de mais de 95% de confiança de que os resultados são diferentes entre si.

O número de grupos utilizado para agrupar cada aplicação foi variado entre cada aplicação. Na Figura 3 é apresentado o volume total da comunicação da aplicação KLT. Como a comunicação apresenta três picos de comunicação, foi escolhido agrupar a aplicação em quatro grupos. Nas Figuras 4, 5, 6 e 7 são apresentados os volumes totais de comunicação das aplicações Sparse 1.3a, Barnes, FMM e Ocean respectivamente. Como essas aplicações possuem dois picos de comunicação, foi escolhido agrupar cada aplicação em três grupos.

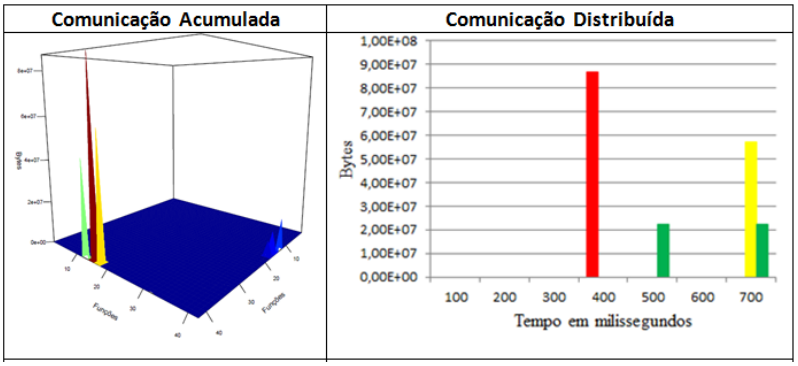


Figura 3. Distribuição da comunicação ao final da aplicação e durante sua execução.

Na Tabela 1 é apresentada a média e o desvio padrão dos tempos de comunicação encontrados utilizando uma taxa de transferência periódica para obtermos as matrizes de tempo. Na Tabela 2 é apresentada a média e o desvio padrão dos tempos de comunicação encontrados utilizando uma taxa de transferência crescente para obtermos as matrizes de

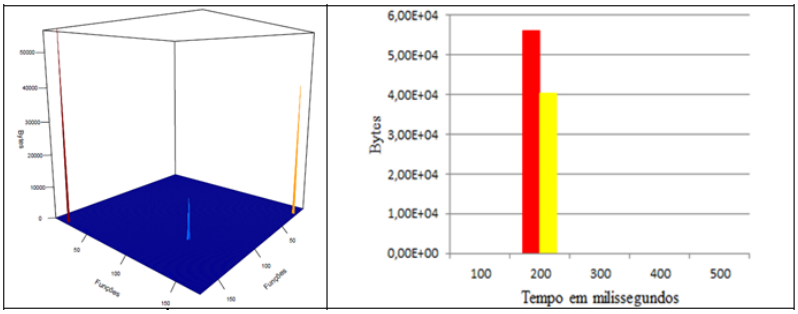


Figura 4. Distribuição da comunicação ao final da aplicação e durante sua execução.

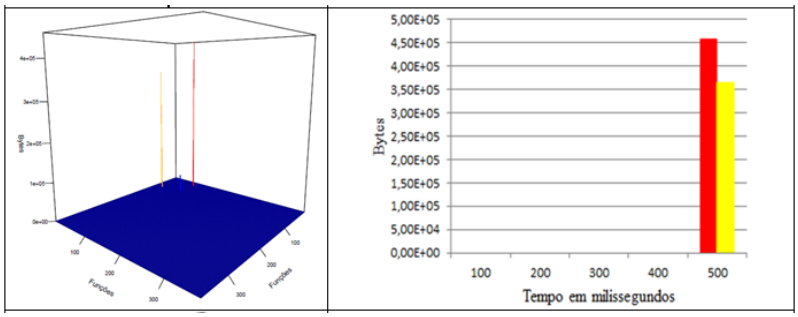


Figura 5. Distribuição da comunicação ao final da aplicação e durante sua execução.

tempo. Na Tabela 3 é apresentada a média e o desvio padrão dos tempos de comunicação encontrados utilizando uma taxa de transferência decrescente para obtermos as matrizes de tempo.

Tabela 1. Média dos tempos de comunicação obtidos utilizando a taxa de transferência periódica.

Aplicação	Comunicação Acumulada	Comunicação Distribuída	Ganho
KLT	1,401 ± 0,356	1,082 ± 0,390	1,294
Sparse 1.3a	0,737 ± 0,230	0,397 ± 0,136	1,858
Barnes	0,0070 ± 0,0014	0,0030 ± 0,0024	2,295
FMM	0,000009 ± 0,000001	0,000007 ± 0,000001	1,223
Ocean	0,0022 ± 0,0003	0,0017 ± 0,0003	1,265

É possível notar que a aplicação KLT apresenta um melhor resultado utilizando uma taxa de transferência crescente, embora não apresente um elevado ganho com essa taxa. Já a aplicação Sparse 1.3a apresenta um elevado ganho utilizando a taxa de transferência periódica, mas também apresenta pior tempo nessa taxa.

A aplicação Barnes apresenta um elevado ganho utilizando a taxa de transferência decrescente, mas também apresenta pior tempo nessa taxa. Já a aplicação FMM apresenta tempos similares utilizando a taxa de transferência periódica e crescente, embora apresente ganhos distintos com essas taxas. Por fim, a aplicação Ocean apresenta o mesmo ganho utilizando a taxa de transferência periódica e crescente, embora apresente tempos distintos com essas taxas.

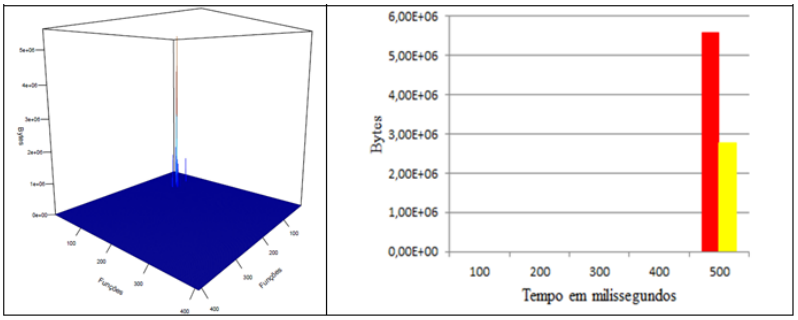


Figura 6. Distribuição da comunicação ao final da aplicação e durante sua execução.

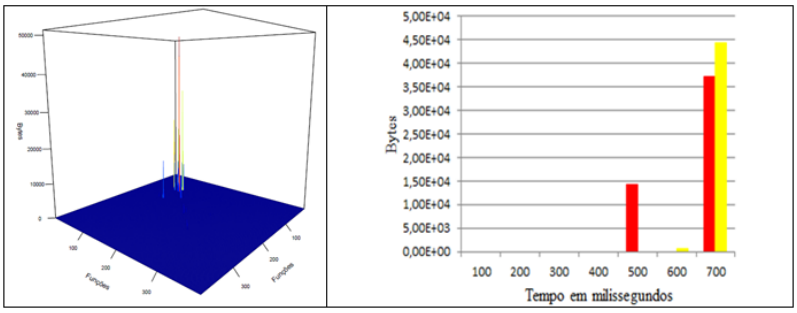


Figura 7. Distribuição da comunicação ao final da aplicação e durante sua execução.

6. Conclusão

Para utilizar todo o potencial de plataformas multiprocessadas, pode-se agrupar a aplicação em grupos menores de modo que cada grupo seja executado em uma unidade de processamento específica, reduzindo assim o gargalo de comunicação, pois um determinado grupo pode executar de forma mais eficiente em um elemento de processamento do que em outro.

Com o propósito de explorar todo o potencial, existem métodos para agrupar aplicações que se baseiam na comunicação total existente na aplicação. Entretanto, é importante destacar que o meio de comunicação é um fator que deve ser considerado em um agrupamento para que que seja possível utilizá-lo de forma mais eficiente, evitando que a comunicação seja realizada quando ocorrer uma elevada concorrência para o uso do mesmo.

Além disso, outro fator que deve ser considerar é o comportamento da comunicação de uma aplicação durante sua execução e não apenas ao final de sua execução, pois com essa informação, é possível verificar de forma pontual os instantes que apresentam uma elevada comunicação entre funções.

Dessa forma, conclui-se que é possível encontrar um agrupamento ótimo utilizando a comunicação distribuída através do tempo, de modo a reduzir o volume de dados comunicados no instante em que ocorre o gargalo. É importante considerar também o meio de comunicação e seu comportamento durante a execução da aplicação, pois durante um agrupamento, deve-se considerar qual a largura de banda que está disponível naquele instante para decidir qual a melhor forma de agrupar a aplicação.

Tabela 2. Média dos tempos de comunicação obtidos utilizando a taxa de transferência crescente.

Aplicação	Comunicação Acumulada	Comunicação Distribuída	Ganho
KLT	0,642 ± 0,202	0,542 ± 0,179	1,184
Sparse 1.3a	0,123 ± 0,024	0,073 ± 0,011	1,695
Barnes	0,0032 ± 0,0015	0,0012 ± 0,0010	2,587
FMM	0,000009 ± 0,000001	0,000008 ± 0,000001	1,148
Ocean	0,0008 ± 0,0001	0,0006 ± 0,0001	1,265

Tabela 3. Média dos tempos de comunicação obtidos utilizando a taxa de transferência decrescente.

Aplicação	Comunicação Acumulada	Comunicação Distribuída	Ganho
KLT	2,794 ± 0,897	2,182 ± 0,870	1,281
Sparse 1.3a	0,556 ± 0,101	0,365 ± 0,066	1,523
Barnes	0,0126 ± 0,0066	0,0048 ± 0,0046	2,621
FMM	0,000026 ± 0,000004	0,000022 ± 0,000003	1,198
Ocean	0,0037 ± 0,0004	0,0028 ± 0,0005	1,312

Referências

Ashraf, I., Ostadzadeh, S. A., Meeuws, R., and Bertels, K. (2013). Evaluation methodology for data communication-aware application partitioning. In *Euro-Par 2013: Parallel Processing Workshops*, pages 739–748. Springer.

Berkhin, P. (2006). A survey of clustering data mining techniques. In *Grouping multidimensional data*, pages 25–71. Springer.

Bonato, V., Marques, E., and Constantinides, G. (2008). A parallel hardware architecture for scale and rotation invariant feature detection. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(12):1703–1712.

Borkar, S. and Chien, A. A. (2011). The future of microprocessors. *Commun. ACM*, 54(5):67–77.

Borkar, S. Y., Dubey, P., Kahn, K. C., Kuck, D. J., Mulder, H., Ramanathan, E. R. M., Thomas, V., Corporation, I., and Pawlowski, S. S. (2005). Intel ® processor and platform evolution for the next decade executive summary.

Che, S., Li, J., Sheaffer, J., Skadron, K., and Lach, J. (2008). Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107.

Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126.

Greengard, L. (1988). *The rapid evaluation of potential fields in particle systems*. MIT press.

Haynes, W. (2013). Student's t-test. In *Encyclopedia of Systems Biology*, pages 2023–2025. Springer.

Horowitz, M. and Dally, W. (2004). How scaling will change processor architecture. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pages 132–133. IEEE.

I. Ashraf and V.M. Sima and K.L.M. Bertels (2015). Intra-application data-communication characterization. In *Proc. 1st International Workshop on Communication Architectures at Extreme Scale*, Frankfurt, Germany.

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Kundert, K. S. and Sangiovanni-Vincentelli, A. (1988). Sparse user's guide–a sparse linear equation solver version 1.3 a. *University of California, Berkeley*, 178.

Lucas, B. D., Kanade, T., et al. (1981). An iterative image registration technique with an application to stereo vision.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

Seward, J. and Nethercote, N. (2005). Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30.

Singh, J. P. (1993). Parallel hierarchical n-body methods and their implications for multiprocessors.

Singh, J. P., Weber, W.-D., and Gupta, A. (1992). Splash: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1):5–44.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH computer architecture news*, volume 23, pages 24–36. ACM.

Zaki, M. J., Meira Jr, W., and Meira, W. (2014). *Data mining and analysis: fundamental concepts and algorithms*. Cambridge University Press.

Modelo de Predição de Desempenho Integrado à Exploração do Espaço de Projetos

Mateus T. dos Santos, Thiago R. de Oliveira, Rhayssa Sonohata, Casio Krebs, Liana Duenha, Ricardo R. dos Santos

[1]Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)
Campo Grande – MS – Brasil

{mateustostesdossantos,rhayssa.sonahata,casiokbrebs}@gmail.com

noscombr@hotmail.com, {lianaduenha,ricardo}@facom.ufms.br

Abstract. *This paper presentes a performance predictor based on the Support Vector Machines (SVM) technique. The predictor has been applied to predict performance on heterogeneous multi and manycore platforms. The predictor has also been integrated into a design space exploration (DSE) tool named MultiExplorer to improve the performance and accuracy of platforms performance at design time. We have trained, tested, and evaluated the predictor using applications from the SPLASH-2 and PARSEC benchmarks running on heterogeneous platforms on the Sniper performance simulator.*

Resumo. *Este artigo apresenta uma técnica de predição de desempenho, baseada na técnica de Máquinas de Vetores de Suporte (SVM - Support Vector Machine), para plataformas com múltiplos processadores heterogêneos. O preditor foi integrado junto à ferramenta MultiExplorer visando melhorar as estimativas de desempenho na solução de problemas de exploração de espaço de projeto. O preditor foi treinado, testado e validado utilizando aplicações dos benchmarks SPLASH-2 e PARSEC e com diferentes modelos de arquiteturas heterogêneas.*

1. Introdução

Durante o ciclo de desenvolvimento de um *sistema em chip* (SoC), faz-se necessário o uso combinado de metodologias e ferramentas que permitam a exploração do modelo arquitetural e microarquitetural, a fim de se atingir objetivos pré-definidos e respeitar restrições de projeto. O processo consiste em um refinamento iterativo a partir de uma configuração inicial e pode ser dividido em duas etapas principais: a simulação em nível de sistema para obtenção de estatísticas de desempenho e físicas do projeto e, a partir destes resultados, a variação sistematizada de parâmetros do projeto de modo a atingir objetivos pré-determinados, em geral, maximização do desempenho do sistema sem extrapolar limites de consumo e área. Este processo é denominado **exploração do espaço de projetos**.

A evolução direta do processo tecnológico mantendo restrições de consumo energético e área resultará em áreas do chip que não poderão estar ativas concomitantemente com o restante do circuito [Raghunathan et al. 2013]. Essa limitação

de utilização é chamada de *utilization wall* e a área que deve ser mantida "desligada" do restante do circuito é denominada *dark silicon*. *Dark silicon* mostra-se presente em sistemas desenvolvidos com transistores fabricados em processos tecnológicos abaixo de 90nm [Dennard et al. 2007] e pode chegar a 75%-85% da área do chip em processos tecnológicos de $8nm$, quando comparado com processo tecnológico de $45nm$ [Esmaeilzadeh et al. 2012, Borkar 2009] (Figura 1).

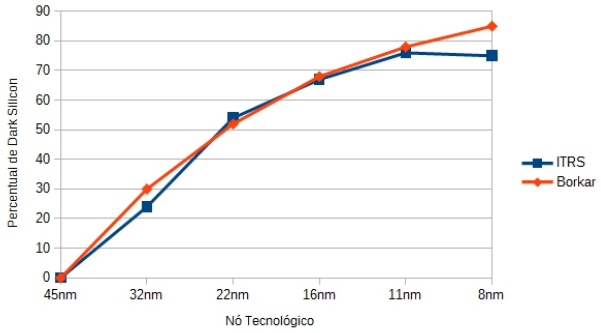


Figura 1. Porcentagens de *dark silicon* estimadas ao longo de processos tecnológicos [Esmaeilzadeh et al. 2012].

A ferramenta *MultiExplorer* [Santos et al. 2017] consiste em um ambiente para exploração do espaço de projetos de plataformas com múltiplos núcleos com suporte a heterogeneidade e ciente de *dark silicon*, utilizando uma uma estratégia multiobjetivo baseada no algoritmo genético NSGA-II [Deb et al. 2002]. Originalmente, a estratégia utilizada pela ferramenta *MultiExplorer* para estimar desempenho das plataformas resultantes do processo de exploração do espaço de projetos foi baseada na soma dos desempenhos individuais dos núcleos. Tal abordagem tem se mostrado distante da predição real das plataformas, especialmente, ao tratar de plataformas com núcleos heterogêneos. Diante dessa limitação, uma alternativa viável é adotar sistemas preditores de desempenho, os quais possibilitam uma significativa redução de tempo quando comparado com simulações e, ao mesmo tempo, possibilitam maior confiabilidade (maiores taxas de acerto) sobre as predições.

Este trabalho apresenta o desenvolvimento de um preditor de desempenho baseado em aprendizado de máquina (mais precisamente, *support vector machines*). O preditor possibilita obter as estimativas de desempenho das plataformas sugeridas pela ferramenta *MultiExplorer* minimizando o tempo (custo computacional) e com taxas de erro controladas. Os experimentos e resultados apresentados demonstram a viabilidade do preditor e a acurácia dessa estratégia em comparação com a abordagem previamente adotada pela ferramenta *MultiExplorer*.

Este artigo está organizado da seguinte forma: a Seção 2 descreve trabalhos relacionados à predição de desempenho de processadores ou de sistemas multiprocessadores e relaciona-os a nossa proposta; a Seção 3 descreve a técnica utilizada para desenvolver o preditor de desempenho; a Seção 4 mostra como aplicar o preditor de desempenho à exploração de espaço de projeto; a Seção 5 ilustra os experimentos de validação e avaliação do preditor e a Seção 6 conclui este artigo.

2. Trabalhos Relacionados

Esta seção apresenta um conjunto de trabalhos científicos relacionados à predição de desempenho de sistemas computacionais e os relaciona com o trabalho proposto neste artigo.

Joseph et al. [Joseph et al. 2006] propõem um modelo de regressão linear para relacionar o desempenho de processadores a um conjunto de parâmetros microarquiteturais com o objetivo de guiar projetistas na tomada de decisão. O modelo é avaliado por meio do FAFSIM, um simulador superescalar com precisão de ciclos e utiliza 26 parâmetros microarquiteturais. Avaliou-se o impacto dos parâmetros no IPC (instruções por ciclo) do sistema com um nível de confiança de 95% e erros menores de 1% e, com base nos resultados obtidos, foi possível inferir que a profundidade do pipeline, tamanho do *buffer* de reordenação que dá suporte a especulação e o tamanho da fila de espera são os parâmetros arquiteturais que mais influenciam no IPC do processador.

O trabalho apresentado por Benjamin C. Lee et al. [Lee and Brooks 2006] apresenta predição de desempenho e consumo energético utilizando modelo de regressão não-linear *Spline* sobre dados experimentais extraídos do Turandot, um simulador de processador superescalar genérico e parametrizável, com suporte à execução fora de ordem. Os preditores de desempenho apresentaram erro médio de 4,9%. 50% a 90% das previsões atingem taxas de erro inferiores a 10%, com erro máximo é de 20% a 33%. As predições foram estimadas a partir de 2000 exemplos, os quais representam uma proporção muito pequena de um espaço de projeto com 22 bilhões de pontos.

O trabalho apresentado por Curtis Maury et al. [Curtis-Maury et al. 2008] analisa o impacto de escalabilidade dinâmica de tensão e frequência (DVFS) e ajuste dinâmico de concorrência (DCT) para a redução do consumo de energia durante o processamento.A validação deste trabalho foi feita por meio de uma biblioteca compilada juntamente com OpenMP [Dagum and Menon 1998], executando em sistemas com 2 e 4 núcleos. Obteve-se redução de consumo de energia de 19%, redução de 6% na potência e melhoria de 14% no desempenho.

Engin Ïpek et al. [Ïpek et al. 2006] basearam-se no benchmark SESC e criaram um modelo para predição de desempenho. Um modelo de regressão não-linear foi adotado para implementação de uma rede neural.Entretanto, sua aplicação não é viável para projetos com um grande espaço de exploração de projeto, pelo fato de que a quantidade de simulações necessárias para o treinamento em um projeto com mais de 1 bilhão de pontos gastaria tempo exorbitante de experimentação, além do tempo de treinamento.

A Tabela 1 sintetiza os trabalhos descritos nesta seção, caracterizando-os de acordo com seus objetivos, sistema-alvo e metodologia. O preditor de desempenho proposto neste trabalho tem como diferencial a integração em um *framework* de exploração de espaço de projetos, que leva em consideração heterogeneidade e *dark-silicon*.

3. Preditor de Desempenho Baseado em SVM

Support Vector Machines (SVMs) são sistemas de aprendizado de máquina que analisam os dados e reconhecem padrões com enfoque em classificação e análise de regressão. As SVMs adaptadas para casos de regressão são chamadas de *Support Vector Regression Machines (SVRs)*, as quais foram introduzidas por Cortes e Vapnik [Cortes and Vapnik 1995]

Tabela 1. Caracterização de trabalhos relacionados à predição de desempenho de sistemas computacionais.

	Simulador	Objetivo	Sistema-alvo	Metodologia
Joseph et al. [Joseph et al. 2006]	FAFSIM	Relacionar parâmetros arquiturais e desempenho	Sistema single-core	Regressão linear
Benjamin C. Lee et al. [Lee and Brooks 2006]	Turandot	Predição de desempenho e consumo energético	Sistemas *single-core*	Regressão não-linear Polinomial com Spline
Curtis Maury et al. [Curtis-Maury et al. 2008]	Sistema *multicore* 2008) +lib OpenMP	Impacto de DVFS e DCT no desempenho e consumo	Sistemas *multi-core*	Monitoramento do hardware + análise estatística
Engin Ïpek et al. [Ïpek et al. 2006]	SESC	Predição de desempenho	Sistemas *multi-core*	Rede neural
Preditor baseado em SVM proposto neste trabalho	MultiExplorer	Predição de desempenho para exploração de espaço de projetos ciente de *dark-silicon*	Sistemas multicore heterogêneos	*Support vector machines*

e mantêm as principais características do algoritmo original. SVMs superam outros sistemas de aprendizado em várias aplicações como, por exemplo: reconhecimento de imagem, classificação independente de aspecto, classificação baseada em núcleos de processamento, reconhecimento de dígitos escritos à mão, detecção de homologia de proteína, entre outras [Cristianini and Shawe-Taylor 2000].

O preditor baseado em SVR foi definido com base em um conjunto de treinamento $T = \{(X_i, d_i)\} \in R^m \times R$, $i = 1 \ldots n$, consistindo em n pares de dados (X_i, d_i), onde X_i corresponde ao vetor de entrada com m parâmetros que definem uma plataforma heterogênea e d_i corresponde ao desempenho esperado da plataforma. O objetivo do problema de regressão é, a partir dos dados de treinamento, determinar uma função que relacione com precisão futuros valores de entrada de uma plataforma ao seu desempenho estimado.

Os dados do conjunto de treinamento para o preditor foram extraídos a partir de simulações utilizando o simulador Sniper [Carlson et al. 2014] e as aplicações Barnes, FFT e Radix do (da *suite benchamrk* Splash2 [Woo et al. 1995]) e Fluidanimate e Swaptions (da *suite benchamrk* PARSEC [Bienia et al. 2008]). Foram utilizados quatro modelos de processadores: Smithfield, Quark, Atom, Arm54 e Arm57. Os principais parâmetros que definem um modelo de processador são: frequência dos *cores*, tamanho das caches L1 e L2, tempo de resposta das caches, largura de banda e associatividade de cache, entre outros. O conjunto de treinamento foi definido com base em combinações entre quantidade de *cores* do tipo base (Smithfield) e quantidade de *cores* IP (Quark, Atom, Arm54 e Arm57), limitadas em um total de 32 cores e o desempenho alcançado pela plataforma com base na métrica IPC descrita na Seção 4. Foram utilizados 2000 exemplos de treinamento.

O treinamento foi realizado a partir da ferramenta ScikitLearn [Pedregosa et al. 2011], que forneceu a técnica de aprendizado de máquina SVR já devidamente validada, juntamente com métodos para calibração e avaliação para o modelo resultante. Utilizamos da técnica de validação cruzada para verificar a capacidade de generalização de um modelo, a qual é amplamente utilizada em problemas que possuem como objetivo a predição. A técnica de validação cruzada *hold-out* divide o conjunto total de dados em dois subconjuntos, um para treinamento e outro para teste (validação). 80% do conjunto T foi utilizado para o treinamento e 20% para avaliação

(validação) das respostas aplicando-se a métrica *erro quadrático médio normalizado* (NMSE). Quanto mais preciso o modelo, menor NMSE. Um preditor ideal tem NMSE igual a zero, enquanto um preditor trivial tem NMSE igual a 1.

Do conjunto de treinamento resultante, 20% foram separados para avaliação do preditor utilizando a métrica *Score* ou coeficiente de correlação R^2, como a medida de qualidade do ajuste do modelo. O *score* é definido na Equação 1, em que u é a soma residual de quadrados e v é a soma total de quadrados (Equação 2), considerando que $resp$ corresponde as respostas já conhecidas e validadas, $pred$ corresponde a predição realizada pelo modelo, n corresponde ao número de exemplos utilizados e $media\_resp$ é a média de todas as respostas utilizadas.

$$Score = (1 - u/v) \tag{1}$$

$$u = \frac{\sum_{i=1}^{n}(resp - pred)^2}{n}, v = \frac{\sum_{i=1}^{n}(resp - media\_resp)^2}{n} \tag{2}$$

A SVR foi configurada com o *kernel* RBF (do inglês, *Radial Basis Function*) ou *kernel* Gaussiano, o qual é definido por uma função K, cujas entradas são vetores x_i e x_j do espaço de entradas da como na Equação 3:

$$K(x_i, x_j) = exp(-\gamma||x_i - x_j||^2) \tag{3}$$

Os principais parâmetros para avaliação e minimização de erro do RBF são γ e *C*. A calibração do parâmetro C é utilizada para a minimização de erros de predição, pois o valor de C controla a compensação dos erros cometidos durante a fase de treinamento; quanto maior o valor C, maior a penalização associada aos erros cometidos. C é um parâmetro que relaciona-se diretamente com a complexidade das hipóteses que podem ser geradas pelo SVR; um valor pequeno de C permite decisões mais simples, comumente resultando em taxas de erro mais altas. Altos valores de C tornam o SVR mais complexo e especializado em seu treinamento, permitindo menores taxas de erro nos casos para os quais ele foi treinado.

A Tabela 2 sumariza resultados da avaliação do preditor de desempenho para dois modelos de processador (Atom e Quark) utilizando o *kernel* RBF com diferentes parâmetros γ e C. Os parâmetros foram encontrados utilizando-se validação cruzada ou *grid search* em conjunto com a curva de validação em C para se encontrar os intervalos de estabilização do aprendizado do modelo SVR. O valor de *Score* próximo a 1 indica que o preditor está muito próximo do limite do que se pode aprender sobre os dados fornecidos. Também é possível ver que a progressão do parâmetro C influencia diretamente na métrica *Score* e que infere-se aumento muito pequeno na precisão para valores de C a partir de 2500.

Um preditor perfeito obteria métrica NMSE igual a 0. Podemos verificar que, para ambos os processadores, ocorrem maiores variações em NMSE para valores de C entre 100 e 2000 e que, a medida que C se aproxima do valor 2500, diminui a redução de NMSE, mostrando novamente a estabilização na acurácia do preditor. Os valores mostrados na Tabela 2 são relativos a três valores de γ (1, 5 e 10), os quais foram os melhores

Tabela 2. Resultados de avaliação do preditor de desempenho para dois modelos de processador (Atom e Quark) utilizando o *kernel* RBF.

	Atom			Quark		
	C	**Score**	**NMSE**	**C**	**Score**	**NMSE**
$\gamma = 1$	2500	0.97435	0.00157	2500	0.97571	0.00176
	2000	0.96920	0.00183	2000	0.97117	0.00188
	1500	0.95998	0.00213	1500	0.95970	0.00221
	1000	0.93071	0.00266	1000	0.92216	0.00269
	500	0.72954	0.00496	500	0.70543	0.00562
	100	0.17874	0.10683	100	0.16592	0.11571
$\gamma = 5$	2500	0.97435	0.00130	2500	0.97571	0.00131
	2000	0.96920	0.00132	2000	0.97117	0.00142
	1500	0.95998	0.00171	1500	0.95970	0.00182
	1000	0.93071	0.00247	1000	0.92216	0.00281
	500	0.72954	0.00657	500	0.70543	0.00819
	100	0.17874	0.08557	100	0.16592	0.09404
$\gamma = 10$	2500	0.97435	0.00156	2500	0.97571	0.00170
	2000	0.96920	0.00199	2000	0.97117	0.00222
	1500	0.95998	0.00232	1500	0.95970	0.00258
	1000	0.93071	0.00333	1000	0.92216	0.00378
	500	0.72954	0.01052	500	0.70543	0.01278
	100	0.17874	0.11099	100	0.16592	0.12074

resultados encontrados por meio da aplicação de *grid search*. Para valores de γ maiores que 10 ou menores que 1, obtivemos perda considerável na métrica NMSE.

4. Predição de Desempenho em Exploração de Espaço de Projeto

Exploração de espaço de projeto refere-se à atividade de explorar e avaliar combinações do conjunto de parâmetros arquiteturas durante a fase de projeto. Diante da grande quantidade de parâmetros dos sistemas atuais e, consequentemente, do vasto espaço de projeto resultante de suas combinações, torna-se inviável basear-se somente na experiência de projetistas ou no desempenho teórico esperado dos componentes envolvidos. Assim, torna-se cada vez mais necessária uma metodologia que automatize parte do processo e auxilie sistematicamente o projetista durante as tomadas de decisão.

A ferramenta MultiExplorer [Santos et al. 2017] possibilita exploração do projeto da partir de uma série de parâmetros e suas restrições. Além de estimar a área de *dark silicon* do projeto, a ferramenta tenta encontrar alternativas de projeto para reduzir o *dark silicon*, utilizando-se de uma extensão do algoritmo genético NSGA-II. A ferramenta sistematiza o processo de otimização de um projeto original de acordo com dois objetivos principais que são naturalmente conflitantes entre si: a minimização da densidade de potência e a maximização de desempenho.

O fluxo do MultiExplorer pode ser visto na Figura 2. Dada a especificação do projeto original fornecida pelo usuário e de qual simulador de desempenho e benchmarks deseja-se utilizar, é realizada a simulação funcional que fornece como saída os relatórios estatísticos de desempenho. Atualmente, o usuário pode escolher um dos seguintes simuladores de desempenho: MPSoCBench, Sniper e Multi2Sim [Santos et al. 2017].

Especificamente, o módulo de exploração de espaço de projetos ciente de *dark silicon* (DS-DSE) do MultiExplorer é o responsável por aplicar uma estratégia multiobjetivo com a intenção de devolver, ao final do processo, uma plataforma alternativa (em quantidade e tipo de núcleos), em que a área de *dark silicon* seja menor que a do projeto original e o desempenho seja mantido. Para tal, utiliza-se de um banco de dados de

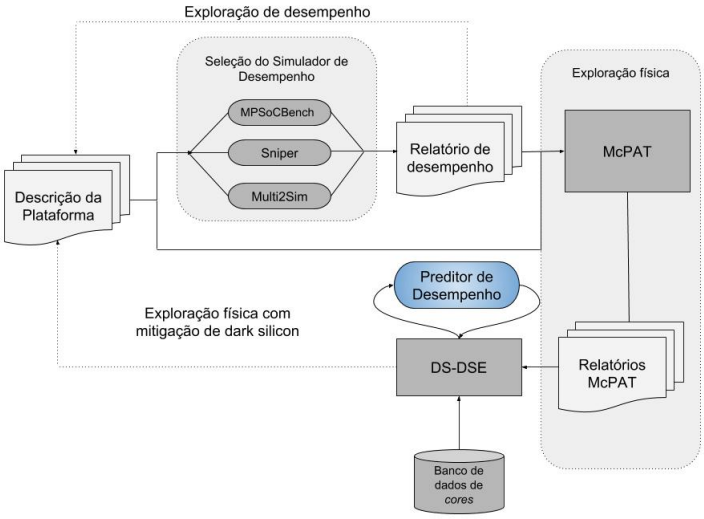


Figura 2. Fluxo de execução do MultiExplorer

núcleos (*cores*) previamente configurado com informações de consumo, área e desempenho. Esta é a etapa denominada *exploração física com mitigação de dark silicon*.

Os modelos de processadores presentes no banco de núcleos são caracterizados pela sua área, potência, densidade de potência e IPC (instruções por ciclo), os quais são estimados diretamente pelas ferramentas de estimativas de desempenho e físicas do MultiExplorer.

Entretanto, o desempenho de uma plataforma heterogênea não é retornado de forma precisa pelo MultiExplorer. O escalonamento de instruções do programa realizado pelo simulador desconsidera a heterogeneidade dos núcleos da plataforma e distribui igualitariamente a carga de trabalho entre eles. Como as aplicações do *benchmark* utilizado nos experimentos são baseadas em sincronização por barreiras, o desempenho da plataforma torna-se limitado pelo núcleo mais lento. Tal situação não reflete os recursos de exploração de paralelismo existentes nos sistemas atuais. Isto posto, propõe-se, no âmbito deste tralho, estimar o IPC médio da plataforma computacional a partir de um escalonamento em que as instruções serão escalonadas de acordo com a capacidade de processamento, mensurada usando a frequência de *clock* de cada núcleo.

Uma plataforma heterogênea é definida por um conjunto de núcleos de configurações distintas. Seja M a quantidade de tipos de núcleos e seja I o total de instruções das aplicações executadas na plataforma computacional. Supondo um escalonamento das instruções de acordo com a frequência de operação dos núcleos, a Equação 4 indica a quantidade de instruções que serão executadas pelos núcleos de um tipo i:

$$I_i = I \times (\frac{F_i}{F}), 1 \leq i \leq M \tag{4}$$

Após esse escalonamento e execução de instruções em cada tipo de núcleo, pode-se calcular a quantidade de ciclos necessárias para que cada núcleo do tipo i execute as instruções alocadas a ele e, consequentemente, o valor de IPC_i (IPC de um núcleo do tipo i). Finalmente, calcula-se o IPC médio da plataforma (Equação 5), o qual será utilizado

como a métrica de desempenho para os experimentos, a partir dos IPCs dos processadores de cada modelo e uma ponderação pela quantidade de instruções executas por grupo de processadores de um mesmo modelo.

$$IPC = \sum_{i=1}^{M} (IPC_i * \frac{I_i}{I}) \quad (5)$$

5. Experimentos e Resultados

Os experimentos e resultados apresentados nesta seção visam demonstrar a integração de um sistema de exploração de espaço de projetos com o preditor de desempenho apresentado além de comparar os resultados desse preditor com os do algoritmo NSGA-II. Nos experimentos a seguir, os núcleos de processadores utilizados no banco de dados foram modelados tendo como referência os dados presentes na Tabela 3. Ressalta-se, entretanto, que os dados presentes no banco de dados indicam estimativas de desempenho, área e potência total de apenas 1 núcleo de cada modelo de processador da Tabela 3, simulados utilizando tecnologia de 32nm para os transistores.

Todas as plataformas de processadores foram simuladas adotando o Sniper como simulador de desempenho com as aplicações barnes, fft e radix do benchmark SPLASH-2, e Fluidanimate e Swaptions do benchmark PARSEC.

Tabela 3. Configurações das plataformas utilizadas nos experimentos.

Processador	Tecnologia(nm)	Área Die (mm^2)	Frequência(GHz)	Total cores	Caches
1 - Smithfield 820	90	206	2.8	2	L1-2-16KB, L2-2-1MB
2 - Quark x1000	32	6,5	0.4	1	L1-1-16KB
3 - Atom Silvermont	22	5,5	0.5	1	L1-1-56KB, L2-1-1MB
4 - ARM Cortex-A53	22	7,2	1.6	1	L1-1-80KB, L2-1-512MB
5 - ARM Cortex-A57	22	13	2.0	1	L1-1-80KB, L2-1-512MB

Como primeira etapa do experimento, estimativas de *dark silicon* (Tabela 4) foram obtidas a partir de configurações de um processador de referência (Smithfield). Esse modelo de processador foi submetido a simulação física usando o fluxo de projeto do MultiExplorer (ferramenta McPAT [Li et al. 2009]) com diferentes processos tecnológicos (90nm, 65nm, 45nm, 32nm) visando identificar se a densidade de potência do chip aumenta e, como consequência, o surgimento de áreas de *dark silicon*. Ressalta-se que a obtenção dos dados da Tabela 4 seguem os conceitos e metodologia propostos em [Santos et al. 2017].

A Tabela 4 apresenta resultados de estimativas físicas (área, potência de pico, densidade de potência e *dark silicon*) considerando apenas a evolução dos processos de fabricação da plataforma *multicore* Smithfield. Nota-se que mesmo mantendo a frequência de *clock* constante ao longo do avanço dos processos tecnológicos, há aumento da densidade de potência gerada pelo aumento da potência de pico. Trabalhos anteriores [Santos et al. 2017] já concluíram que este aumento é devido ao incremento da potência de fuga. Devido à utilização de diferentes metodologias de estimativas de acordo com o processo tecnológico, o estimador físico da ferramenta MultiExplorer, adaptado a partir do McPAT [Li et al. 2009], alcançou 5,52% de *dark silicon* o que representa uma área de $8,5mm^2$ sobre o chip no processo de 45nm.

Tabela 4. Estimativas físicas e frequência de *clock* constante: processador Smithfield.

Avanço tecnológico p	1	2	3	4
Quantidade de núcleos	2	4	8	16
Tecnologia (nm)	90	65	45	32
Frequência (GHz)	2,8	2,8	2,8	2,8
Área do *chip* (mm^2)	203,14	179,23	154,09	157,32
Potência de pico (W)	74,28	100,26	151,51	143,47
Densidade de potência $(\frac{W}{mm^2})$	0,37	0,56	0,98	0,92
Área do núcleo (mm^2)	64,71	35,16	16,93	9,32
Potência de pico do núcleo (W)	73,13	98,70	149,19	142,78
Densidade de potência do núcleo $(\frac{W}{mm^2})$	0,57	0,70	1,10	0,96
Percentual de *dark silicon* $(\%)$	-	2,46	5,52	3,76

A segunda etapa experimental consiste em, a partir dos dados gerados na Tabela 4, realizar a exploração do espaço de projeto na plataforma de 32nm (coluna 5 da Tabela 4). O objetivo nesse passo é buscar plataformas alternativas que minimizem a densidade de potência (e como consequência o *dark silicon*), maximize o desempenho, tendo como restrição a área do chip.

Para isto, utilizou-se o banco de dados de núcleos de processadores com base na Tabela 3, mas agora com parâmetros físicos equivalentes à tecnologia de fabricação de $32nm$, conforme apresentado na Tabela 5.

Tabela 5. Parâmetros físicos presentes no banco de dados dos núcleos com tecnologia de 32nm.

Núcleo	**Potência**	**Área** (mm^2)	**Desempenho**
Smithfield	8,9	9,32	6428
Quark x1000	1,06	6,42	502,53
ARM A53	5,5	7,2	3125,68
ARM A57	12,13	13	4006,64
Atom Silvermont	2,51	5,5	648,47

O experimento realizado consiste em identificar e apresentar plataformas de processadores alternativas (compostas por núcleos originais e núcleos IP - fornecidos pelo banco de dados), na tecnologia de 32nm. A exploração de espaço do projeto visando a obtenção de arquiteturas alternativas que minimizem o *dark silicon* foi realizada a partir de uma otimização multiobjetivo em que a área do chip destinada aos núcleos é uma restrição, e os objetivos são maximizar o desempenho e minimizar a densidade de potência. Para exploração dos parâmetros físicos utilizou-se o algoritmo NSGA-II já implementado no fluxo do MultiExplorer, com as seguintes configurações:

- Taxa de Cruzamento=50%;
- Taxa de Mutação=10%;
- Tamanho da População=12;
- Quantidade de Gerações=50000.

A taxa de cruzamento de 50% garante que os filhos herdem metade das características de cada pai. A taxa de mutação para a maioria dos problemas é um valor baixo; porém, como o indivíduo modelado neste problema possui poucos genes, uma taxa de mutação tão baixa não garante a variabilidade da população. O tamanho da população e a quantidade de gerações se basearam em experimentos empíricos realizados neste trabalho.

O algoritmos NSGA-II foi comparado com um algoritmos de força bruta em [Santos et al. 2017] utilizando as mesmas configurações dos experimentos apresentados nesse trabalho. Foi possível observar que a plataforma com melhor desempenho retornada pelo NSGA-II tem o mesmo desempenho da plataforma obtida no força bruta e a melhor densidade de potência atingida no NSGA-II é também igual a solução obtida no algoritmo força bruta.

Objetivando eliminar totalmente o *dark silicon*, a densidade de potência dos núcleos no projeto proposto deve ser menor ou igual à densidade de potência do projeto de referência com 90nm $dp_t \leq 0,57$ (linha 10 e coluna 2 da Tabela 4). Além disso, deseja-se que o desempenho obtido seja maior que o do projeto original com 16 núcleos ($d_t > 37810$), obtido a partir da multiplicação do desempenho do núcleo pela quantidade de núcleos, e que a plataforma respeite a restrição de área total do chip $a_t \leq 157,32$, calculada pela multiplicação da área do *core* pela quantidade de *cores* ($a_t = a_o \times n_o$).

A Tabela 6 apresenta resultados do experimento com a exploração do espaço de projeto usando NSGA-II com preferência *a posteriori* de $dp_t < 0,57$ e restrição de área $a_t < 157,32$. Todas as soluções obtidas são viáveis e respeitam o limite de área e densidade de potência. Entretanto, o desempenho otimista utilizado originalmente pelo algoritmo é distante das estimativas mais acuradas retornadas pelo preditor.

Interessante notar que não houve qualquer solução livre de *dark silicon* que gerou desempenho superior ($d_t > 37810$) em comparação com a plataforma original. Mesmo assim, optou-se por apresentar os resultados para comparar a qualidade das soluções livres de *dark silicon* com relação à proximidade do desempenho requerido. Vale ressaltar também que a solução NSGA5 atende às restrições de área e densidade de potência, mas não alcança o objetivo de alcançar desempenho $d_t > 37810$.

Tabela 6. Soluções da proposta com restrição $dp \leq 0,57$.

Solução	n_o	n_{ip}	T_{ip}	**Área** a_t (mm^2)	**Densidade de Potência** dp_t $(\frac{W}{mm^2})$	**Desempenho (preditor)** d_t	**Desempenho original** d_o
NSGA1	8	12	9	151.6	0.55	22086	57454
NSGA2	7	14	9	155.12	0.50	19980	52031
NSGA3	6	15	9	152.22	0.46	17968	46106
NSGA4	5	17	9	155.74	0.40	12871	40683
NSGA5	4	18	9	152.84	0.36	11093	34758

A Tabela 6 também compara o desempenho estimado originalmente ao desempenho estimado pelo preditor SVR. A estimativa original escala linearmente em função da quantidade de *cores* da plataforma, o que se mostra como uma abordagem otimista ou de melhor caso. Por outro lado, a estimativa do preditor SVR considera a heterogeneidade da plataforma e o escalonamento da carga de trabalho entre os *cores*, resultando em uma estimativa de desempenho mais precisa. Considerando que o desempenho do sistema é um objetivo significativo do algoritmo de exploração do espaço de projeto, a precisão de sua predição torna mais confiável a escolha das melhores configurações para mitigar *dark-silicon*.

6. Conclusão

Este trabalho apresentou o projeto e desenvolvimento de um preditor de desempenho baseado na técnica de máquinas de vetores de suporte (SVM - *Support Vector Machines*).

O preditor foi integrado junto a infraestrutura de exploração de espaço de projetos denominado MultiExplorer. A utilização de um preditor de desempenho com alta taxa de correlação (R^2) junto a uma infraestrutura de exploração de espaço de projetos mostra-se uma ferramenta viável e factível para possibilitar maiores níveis de confiabilidade para proposição de alternativas arquiteturais.

Os experimentos realizados compararam os resultados do preditor e uma ferramenta de exploração de projetos. É possível observar pelos resultados apresentados na seção de Experimentos que o preditor adiciona qualidade a escolha de arquiteturas viáveis como alternativas para mitigar o *dark silicon*, maximizar o desempenho mantendo as decisões e restrições de área de projeto. Extensões deste trabalho podem ser voltados para a sua aplicação em arquiteturas heterogêneas compostas por GPUs e FPGAs.

Referências

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University.

Borkar, S. (2009). Design perspectives on 22nm CMOS and beyond. In *Proceedings of the 46th Annual Design Automation Conference*, pages 93–94. ACM.

Carlson, T. E., Heirman, W., Eyerman, S., Hur, I., and Eeckhout, L. (2014). An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Cristianini, N. and Shawe-Taylor, J. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press.

Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., De Supinski, B. R., and Schulz, M. (2008). Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 250–259. ACM.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

Dennard, R. H., Cai, J., and Kumar, A. (2007). A perspective on today's scaling challenges and possible future directions. *Solid-State Electronics*, 51(4):518–525.

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. (2012). Dark silicon and the end of multicore scaling. *IEEE Micro*, 39(3):122–134.

Ïpek, E., McKee, S. A., Caruana, R., de Supinski, B. R., and Schulz, M. (2006). *Efficiently exploring architectural design spaces via predictive modeling*, volume 41. ACM.

Joseph, P., Vaswani, K., and Thazhuthaveetil, M. J. (2006). Construction and use of linear regression models for processor performance analysis. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 99–108. IEEE.

Lee, B. C. and Brooks, D. M. (2006). Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 185–194. ACM.

Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. (2009). Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Raghunathan, B., Turakhia, Y., Garg, S., and Marculescu, D. (2013). Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors. In *Proceedings of the DATE*, pages 39–44. EDA Consortium.

Santos, R., Duenha, L., Silva, A. C., Sousa, M., Tedesco, L. A., Melgarejo, J. C., Santos, T., Azevedo, R., and Moreno, E. (2017). Dark-silicon aware design space exploration. *Journal of Parallel and Distributed Computing*.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH computer architecture news*, volume 23, pages 24–36. ACM.

A Flexible Instruction Set Architecture Filter for Custom Soft-core Processors

Erinaldo Pereira Silva[1], **Carlos Alberto Oliveira de Souza Junior**[1], **Thadeu Antonio Ferreira de Melo**[1], **Eduardo Marques**[1]

[1]Institute of Mathematical and Computer Sciences (ICMC) - University of São Paulo

{erinaldo,caosjr,thadeu.costa}@usp.br, emarques@icmc.usp.br

Abstract. *The flexibility provided by soft-core custom processors allows for optimization at the software and hardware levels. This paper presents an approach for automatic generation of soft-core processors with specialized Instruction Set Architectures (ISA) for any given target application. We developed a custom open source soft-core processor that is compatible with Altera's development kits and can run code compiled for Nios II platforms. This processor is fully integrated with SOPC Builder and Qsys environment, both provided by Altera. We also introduce a workflow that could be used in any system based on an open soft-core processor.*

Resumo. *A flexibilidade dos processadores customizados soft-core permite otimizar sistemas tanto em software como em hardware. Este artigo apresenta uma abordagem para geração de processadores soft-core com um conjunto especializado de instruções de arquitetura para qualquer aplicação alvo. Foi desenvolvido um processador soft-core de código aberto que é compatível com kits de desenvolvimento da Altera, e que pode executar código compilado para plataformas Nios II. Esse processador foi totalmente integrado com o SOPC Builder e o ambiente Qsys, ambos da Altera. Também é apresentado um fluxo de trabalho que poderia ser usado para qualquer sistema baseado em um processador soft-core de código aberto.*

1. Introduction

Soft-core processors are mostly found in embedded systems based on FPGAs (Field Programmable Gate Arrays). Nios II [Intel 2017b], MicroBlaze [Xilinx 2017] and Leon3 [Gaisler 2017] are the among the most popular soft-core processors currently in use by the scientific community and commercial applications. Leon3 has an open source version available: people can download the code and see the inner design of the processor. Nios II and MicroBlaze have a proprietary architecture: their source code is encrypted and restricted to the products developed or licensed by their respective FPGA suppliers (Altera/Xilinx).

Soft-core processors usually allow some level of customization, which increases the flexibility of the design when compared to hard-core processors (ASIC). Designers and developers can take advantage of this flexibility to make optimizations in hardware, by changing the processor's architecture, and also in software, by improving the source code of the program which the processor runs. However, those mainstream soft-core

processors make available only a limited set of hardware parameterization options, such as allowing to change the cache size or introducing custom instructions. In the case of an encrypted processor, the restrictions are those imposed by the tools provided by the manufacturer, which are the only tools capable of decrypting the processor's source code. Therefore, encrypted processors do not allow deeper changes in their architectures, such as changing their pipeline stages.

The Nios II and Microblaze soft-core processors are intended for FPGA projects and allow developers to integrate custom hardware modules or instructions into these processors. However, the developer cannot customize freely and is obliged to follow a standard format set by the manufacturer. This article aims at exploring the impact of customization of soft-core processors through the use of a technique that reduces the set of instructions of a processor, removing all hardware that is not required by the application to be executed, thus generating a soft-core specific to the application.

[Yiannacouras et al. 2007] presented a proposal for exploring and customizing soft-core processors (reducing their ISA) for a specific application and compared the resulting processors with the various Nios II versions. It also demonstrated that by generating a processor specifically for an application it was possible to obtain an average improvement of 14.1% in FPGA occupied area over the analyzed benchmarks.

[Rajotte et al. 2011] also applied the technique of reducing the ISA and customizing instructions for a specific type of application. However, it focused on the performance gains, showing that the processor with a reduced ISA obtained a speedup of 1.57x compared to Microblaze and 2.57x compared to the same processor proposed but with complete ISA and without custom instructions. Thus, reducing a processor's ISA can also increase performance, besides reducing area requirements.

[Wold et al. 2012] implements the same technique coupled with a classification method, generating a specific processor for a target application which achieves 15% performance improvement, while reducing area.

This paper is organized as follows. Section 2 introduces the concepts of a flexible soft-core processor. Section 3 presents an overview of the BSP Processor. In Section 4 we detail our approach. The evaluation and results are shown in Section 5. Section 6 presents the conclusions.

2. Implementing an ISA Filter for a Custom Soft-core Processor

Changing an open source architecture requires a great effort from the developer and significant time to test it. Such amount of effort is usually unfeasible, either due to the scope of a project or due to time-to-market constraints.

Despite the ease of programming for a soft-core processor, it is not typical for an embedded program to use its full ISA. This causes a waste of hardware resources since the program does not need every instruction implemented by the processor. For instance, if the processor is running a program that does not perform any divisions, there is no need for the processor to handle DIV instructions, and its ALU (Arithmetic Logic Unit) does not need to implement a division operation. Sometimes, programs do not require floating-point operations, allowing for the entire removal of a complex hardware unit such as the FPU (Floating Point Unit).

This paper presents an approach to reduce the number of hardware resources required by a soft-core processor taking into account the software which is going to be executed. This processor is "tailor-made" for each program that it runs. We used a custom open source soft-core processor, namely the BSP Processor [da Silva Pereira 2014], which is compatible with Altera's Nios II ISA. This processor is also fully compatible with the Nios II bus and can be used with the development tools provided by Altera.

To enable our approach, we developed an automated parser that looks into the assembly listings of a given program (compiled for Nios II) and generates the processor's HDL, which is compiled to RTL and can then be synthesized using Altera Quartus II. Both HDL and RTL are open, allowing developers to acquire a better understanding of the intrinsic architecture details and of the processor's operation. Developers can also make any changes whenever they need to.

3. Custom Flexible Processor

We adopted BSP [da Silva Pereira 2014] as our open source soft-core processor. BSP is based on a simple MIPS architecture [Arvind and Asanovic 2015]. BSP and Nios II share the same ISA. Nios II has three basic configurations (Standard, Fast and Economy) with different microarchitectural details. From those, BSP is more similar to "Standard" configuration.

BSP is described using the BSV (Bluespec System Verilog) [Bluespec 2017] language, which is available free of cost for academic and learning purposes. When compared to Verilog or VHDL, BSV is a higher level HDL that is more user-friendly and has structures closer to programming languages like C, without losing fine-tune control over hardware microarchitectural details.

3.1. BSP Architecture

Just like Nios II, we can configure BSP to use either the von Neuman or the Harvard bus architecture. The BSP pipeline is close to the three-stage MIPS-I traditional architecture [Harris and Harris 2007]. The three stages of the BSP are Fetch, Decode-Execute and Write-Back (see Figure 1). In the Fetch stage, the instructions are obtained from the instruction cache and sent to the next stage. In the Decode-Execute stage, the instructions are decoded, the Register File is read by accessing the registers used in the instruction, the instruction is executed and the results are sent to the next stage. In the Write-Back stage, the result is written in the Register File or sent to the memory bus.

Our processor has precisely 32 registers. Changing this number would mean modifying the instruction coding since it uses 5 bits to identify a register. Such modification would turn our ISA into an instruction set that is not compatible with NIOS II.

We designed the BSP architecture so that most of the instructions are executed in one clock cycle, a property usually desirable for a reduced instruction set computing (RISC) architecture. However, the division instructions were implemented using a sequential divider because a combinational divisor would generate a large critical path on the FPGA, directly reducing the attainable processor clock.

Regarding the cache memory, we have a different cache for instructions and data; we show them in Figure 1. We implement both of them with FPGA RAM blocks, which

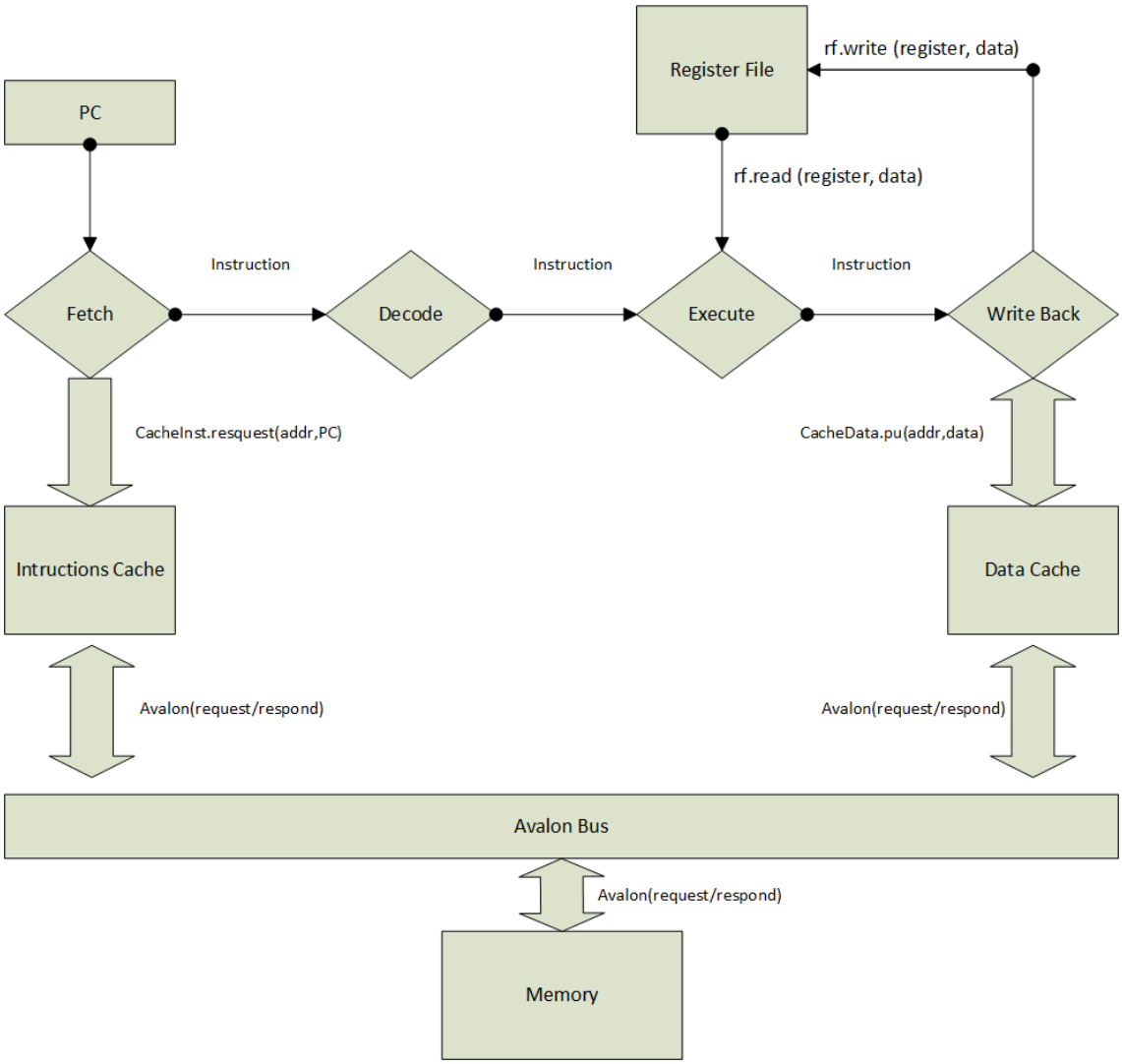


Figure 1. BSP pipeline

can respond to requests in a single clock cycle. The instruction cache is connected to a prefetch circuit which reads words in sequence from the memory bus whenever it is idle.

Both the instruction cache and the data cache have access to the memory bus – if both of them try to access the bus during the same cycle, one of them is chosen randomly, and the one which lost the conflict is tagged. The loser has the priority in the next round.

The cache memory employs direct mapping. An address is mapped to a cache line using its least significant bits. A parameter allows for choosing the number of lines available in each cache. However, the length of each line is fixed to 32 bits. That is a restriction imposed by the Avalon bus that works with the same word size as the processor.

We developed the processor in a modular way: we can design and develop each component of the architecture separately as an independent system. That is possible because of BSV, which eases plugging different modules according to the designer's specification. Designing a project in such a modular fashion facilitates describing it, validating the parts that compose it, conducting tests and writing documentation.

3.2. BSP Instruction Set

In order to achieve compatibility with Nios II's ISA, the BSP instructions obey the conventional format of Nios II instructions. The Nios II ISA has a regular set of 87 instructions, divided into three types of format: I-Type, R-Type, and J-Type. In addition, the Nios II assembly language includes 19 pseudo-instructions that are implemented using instructions from the regular set.

BSP implements 90% of the Nios II instruction set, as can be seen in Table 1, which lists the total number of implemented instructions per type. The instructions that have not been implemented include those that manipulate specific registers of the Nios II architecture when certain modules of the processor are present, such as the MMU and the MPU. BSP also omits instructions related to Nios II exception handling. BSP does not have an MMU nor an MPU and does not have a specific unit for exception control and handling.

Table 1. BSP and Nios II Instructions

Instruction Type	Nios II	BSP	%
R-Type	43	34	79
I-Type	42	41	97
J-Type	2	2	100
Pseudo	19	19	100
Total(without pseudo)	87	77	88
Total(with pseudo)	106	96	90

Although BSP is not well-suited to execute an operating system, due to the instructions that have not been implemented, it can successfully execute several computational tasks, such as those presented in Table 2, whose object code was generated by the Nios II compiler.

4. Instruction Set Architecture Filter

The Filter purpose is to analyze the source and object codes of an application to identify the instructions and their critical regions. This analysis is based on the application's profiling results and assembly code to decide which instructions the processor has to implement. The Filter then generates a customized processor with only the instructions necessary for execution.

The Filter is responsible for generating BSP cores with a specific reduced ISA. This process consists of compiling the benchmark, which in Figure 2 is represented by *program.c*, using *nios2-elf-gcc*, which builds *program.o*. That object file is an input for *nios2-elf-ld*, which generates the file *program.bin*; the binary file is given as input to *nios2-elf-objdump* to generate the *program.asm*, file containing the program's assembly listings. The Filter reads the program and identifies the required instructions, finally generating the *Instruction.bsv* file, which specifies the processor's instruction set and how to decode it, and the *Processor.bsv* file, which implements the instructions. The *.bsv* files are provided to the Bluespec compiler to generate a Verilog RTL file which can be fully synthesized by Quartus II and programmed to an FPGA.

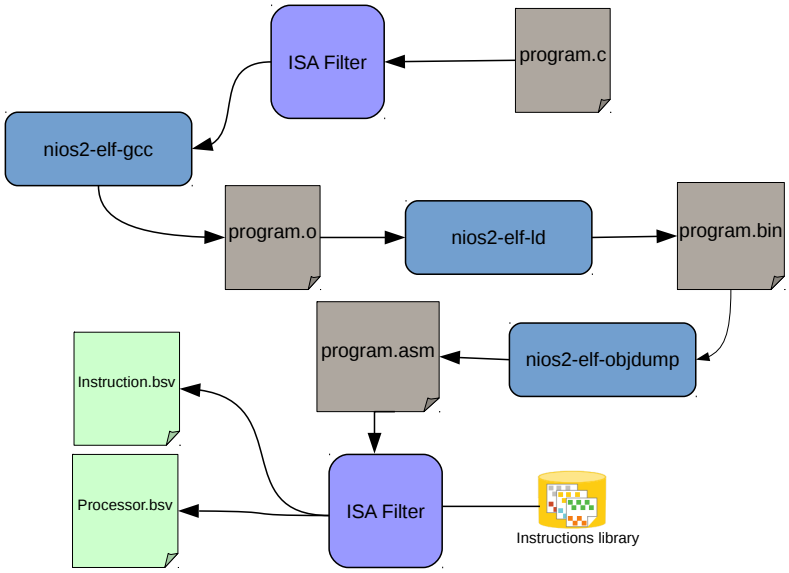


Figure 2. Instruction filter workflow

In the process of generating a customizable BSP for the target application, the Filter uses the disassembles the object code produced by the Nios II compiler and automatically generates a BSP with a more restricted set of instructions. The process of generating the processor is described in Figure 2.

We are currently working on a profiling-based approach, which aims at generating accelerators for executing the application bottlenecks, as pointed out by the profiling results. Our follow-up project is currently adopting the OpenCL HDL [Intel 2017a] for describing those accelerators. Our target application is a case study of the Brazilian Weather and Climate Forecasting System (BRAMS) [CPTEC/INPE 2015]. The process of generating the accelerators is described in Figure 3.

5. Experiment Evaluation

We evaluated our approach by using the representative set of 29 benchmarks shown in Table 2. This table also shows the number of assembly instructions generated by compiling the C source code. Table 2 also shows the number of instructions which remained in the ISA of each specialized processor. The full version of the BSP has 96 instructions, whereas the larger ISA (see *divlu*) among the specialized versions has only 65 instructions.

For each benchmark, we created a specialized BSP Processor with a reduced ISA, employing the flow described in Section 4. Then, we compared every specialized processor with the full version of the BSP. We synthesized the RTL targetting the Altera Stratix V family, taking into advantage the fact that BSP is fully integrated to the SOPC Builder and Qsys environments. Since the customization process is done offline, every change in the application requires a static reconfiguration.

The versions with reduced ISA need, in average, 35% fewer logic elements (i.e.

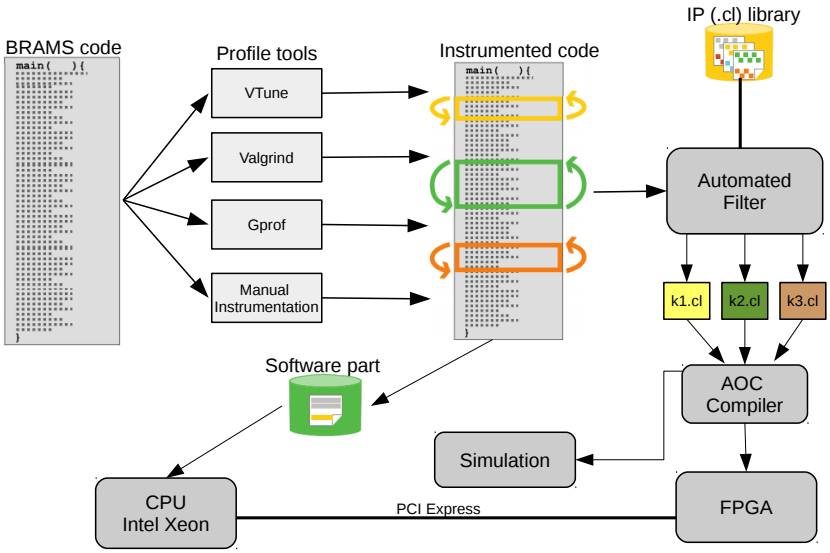


Figure 3. BRAMS OpenCL filter structure (in development)

ALMs) than the full BSP (see Figure 4). In the best case (see *viterbi_gsm*), the reduced ISA version is 40% smaller, while in the worst case (see *isqrt1*) the number of ALMs was reduced by 7%.

Table 2. Instructions usage for each program

Benchmark	*Kernel LOC*	*Total of Assembly Instructions*	*# of Instructions in the Reduced ISA*
adpcm_coder	67	700	57
adpcm_decoder	52	497	55
boundary	18	1990	63
bubble_sort	14	216	42
change_brightness	24	219	39
compositing	12	327	51
conv_3x3	81	295	49
crc32	15	893	59
divlu	16	2350	65
gcd1	15	1652	59
idct_8x8	226	1739	64
isqrt1	21	930	63
isqrt2	16	916	60
isqrt3	17	239	45
isqrt4	18	226	41
mad_8x8	35	340	56
mad_16x16	36	464	56
median_3x3	82	781	54
modexp	11	229	38
max	9	1647	60
motion_estimation	22	1703	58
perimeter	35	451	51
pix_sat	24	905	59
rgb_to_hsv	57	529	55
rng	177	377	26
sad_8x8	17	3123	61
sad_16x16	17	3160	61
sobel	51	420	52
viterbi_gsm	37	460	54

Figure 5 presents the number of registers used in each case. The data shows an average use of 90% of registers when compared to the version with the full ISA. This

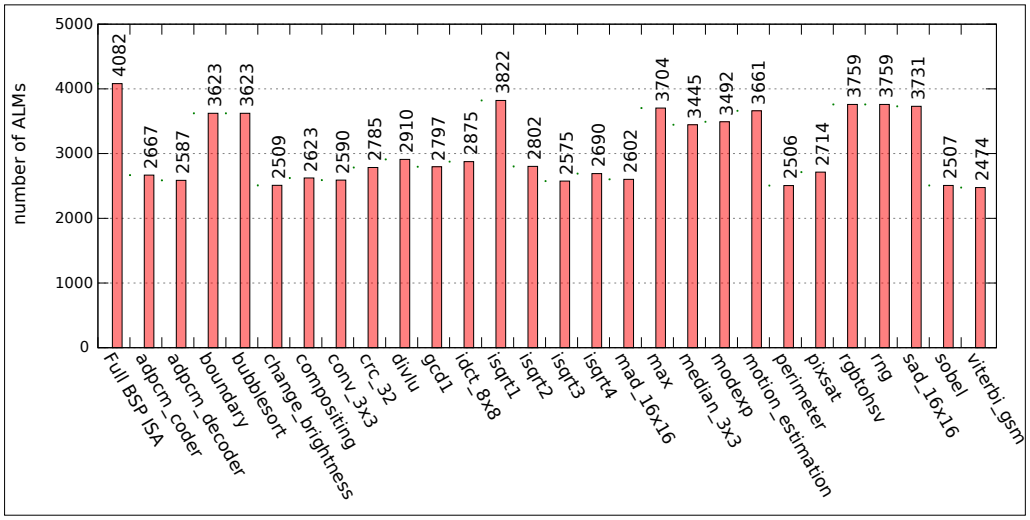


Figure 4. Number of ALMs obtained for each benchmark

reduction in the number of required registers has an impact on the total FPGA area. Similarly, Figure 6 shows the reduction in the number of LUTs.

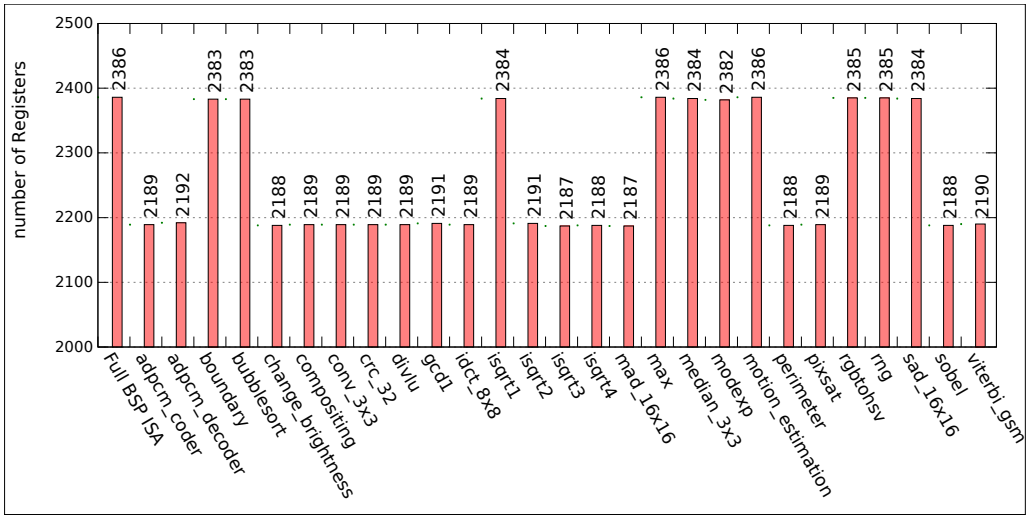


Figure 5. Number of registers used by reduced BSP (full BSP = 100%)

Figure 7 presents the ratio of the maximum frequency achieved by each version to the frequency achieved by the full BSP. It is possible to observe that the average frequency increase was of about 120.68%. Comparing to the data presented in Figure 4, the increase in frequency appears to be related to the reduction in the number of logic and DSP elements needed by the design.

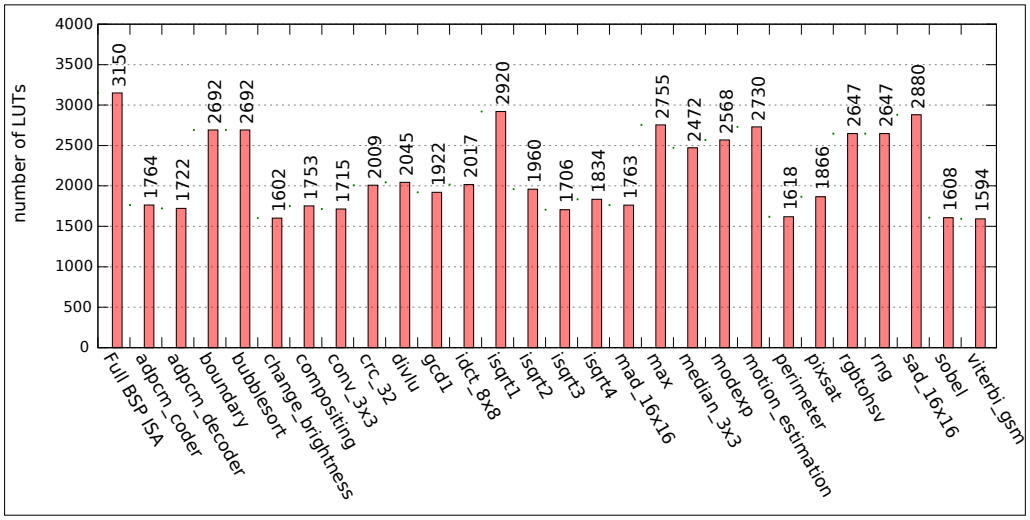


Figure 6. Number of LUTs by used reduced BSP (full BSP = 100%)

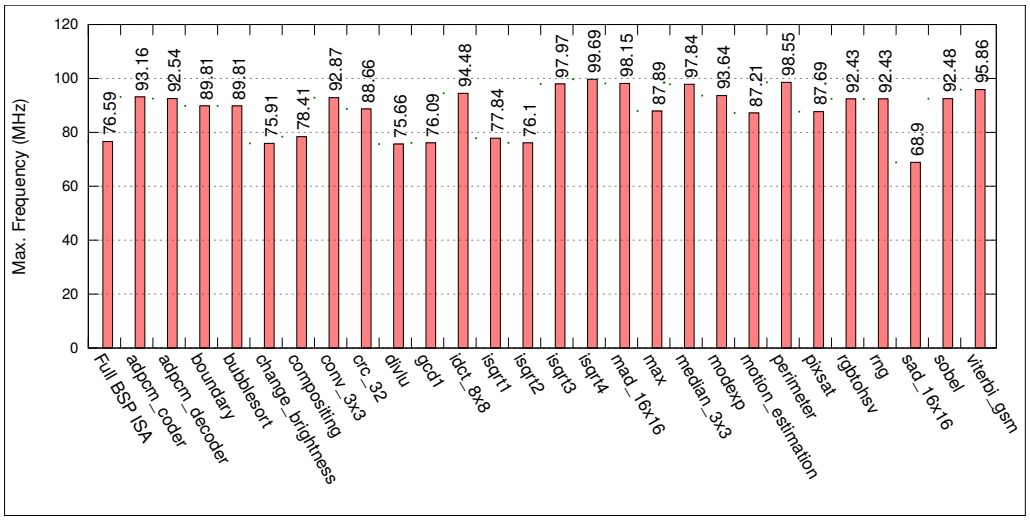


Figure 7. Max Frequency of reduced BSP (full BSP = 100%)

6. Conclusions

This paper presented yet another approach for automatically generating soft-cores with reduced ISA targetted at specific applications. We have shown that it is possible to reduce area and still increase performance. One of the advantages of the proposed processor is its compatibility with the Nios II's ISA and with the development tools provided by Altera.

In the future, we intend to develop custom instructions using OpenCL and to integrate the proposed environment into HARP 2 [Sheffield 2016], envisioning the ability to adjust the compiler based on the target architecture.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

References

Arvind and Asanovic, K. (2015). Complex Digital Systems. http://csg.csail.mit.edu/6.884/index.html. Course Handouts.

Bluespec (2017). Learning Bluespec. http://wiki.bluespec.com/. Tutorials and Examples.

CPTEC/INPE (2015). Brazilian developments on the Regional Atmospheric Modelling System. http://brams.cptec.inpe.br/. Model Description.

da Silva Pereira, E. (2014). Design of an open-source processor in Bluespec based on Soft-core processor from Altera Nios II. *Institute of Mathematical and Computer Sciences (ICMC) - University of São Paulo*. https://doi.org/10.11606/d.55.2014.tde-25092014-094648.

Gaisler, C. (2017). LEON3 processor. http://www.gaisler.com/index.php/products/processors/leon3.

Harris, D. M. and Harris, S. L. (2007). *Digital Design and Computer Architecture*. Elsevier, Incorporated, 1st edition.

Intel (2017a). Intel FPGA SDK for OpenCL. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

Intel (2017b). Nios II Gen2 Processor Reference Guide. https://www.intel.com/content/www/us/en/programmable/documentation/iga1420498949526.html.

Rajotte, S., Gil, D. C., and Langlois, J. M. P. (2011). Combining ISA extensions and subsetting for improved ASIP performance and cost. In *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, pages 653–656.

Sheffield, D. (2016). Xeon + FPGA: The HARP program. https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf. Xeon + FPGA - Presentation ISCA 2016.

Wold, A., Koch, D., and Torresen, J. (2012). Design techniques for increasing performance and resource utilization of reconfigurable soft CPUs. In *2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 50–55.

Xilinx (2017). MicroBlaze Soft Processor Core. https://www.xilinx.com/products/design-tools/microblaze.html.

Yiannacouras, P., Steffan, J. G., and Rose, J. (2007). Exploration and customization of fpga-based soft processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):266–277.

Integrando o MetaTrader5 com Aceleradores FPGA via OpenCL Named Pipes

Cláudio R. Costa[1], **Leandro de S. Rosa**[1], **Vanderlei Bonato**[1]

[1] Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) – São Carlos – SP – Brazil

{clauiocosta, leandrors, vbonato}@usp.br

Abstract. *Today, financial markets operations move hundreds of billions of dollars, through thousands of operations per stock every day, attracting many investors who seek to profit from stock market exchanges. To maximise profits, investors tend to analyse as many stocks as possible, what naturally leads to computer-aided trading. To realise high-frequency trading, traditional computers do not suffice, resulting in the necessity to create accelerators to enable tight timing constraints to be matched. This project aims to integrate the MetaTrader5 software, which is used to perform operations in the financial market, and the Intel FPGA SDK for OpenCL, which allows the quick creation of hardware accelerators for OpenCL applications. This integration will improve the production of low-power and high-performance hardware accelerators for financial applications, improving the stocks analysis and, consequently, the profit.*

Resumo. *Hoje em dia, operações nos mercados financeiros movimentam centenas de bilhões de dólares, através de milhares de operações por ações todos os dias, atraindo muitos investidores que buscam lucrar com as bolsas de valores. Para maximizar os lucros, os investidores tendem a analisar o máximo de ações possíveis, o que naturalmente leva à negociações realizadas por computadores. Para realizar as negociações de alta frequência (HFT), os computadores tradicionais não são suficientes, o que cria a necessidade de aceleradores para permitir que as restrições de tempo sejam correspondidas. Este projeto visa integrar o software MetaTrader5, que é usado para realizar operações no mercado financeiro, e o Intel FPGA SDK for OpenCL, que permite a rápida criação de aceleradores de hardware para aplicações OpenCL. Essa integração melhorará a produção de aceleradores de hardware de baixo consumo e alto desempenho para aplicações financeiras, melhorando a análise de ações e, consequentemente, o lucro.*

1. Contextualização

Ao longo dos anos o mercado financeiro tem atraído cada vez mais investidores (clientes) e ações com a promessa da obtenção de lucro através de trocas e arrecadações de investimentos para empresas.

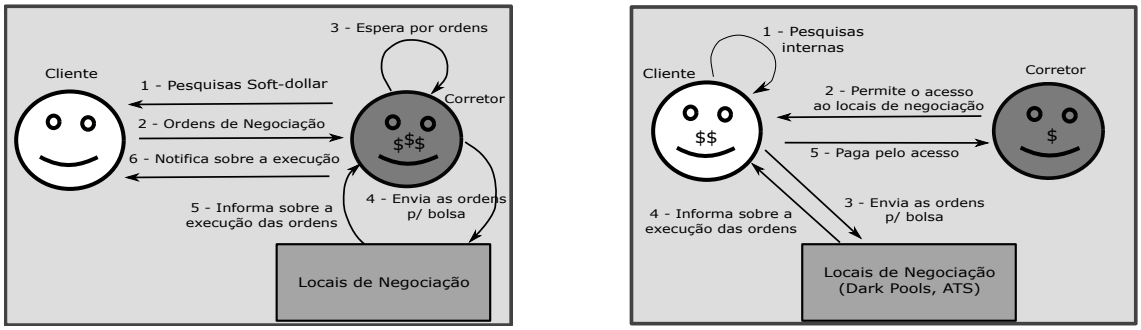
A Figura 1a apresenta o fluxo do mercado financeiro tradicional, que segue as seguintes etapas:

1. Os corretores divulgavam suas ideias para seus clientes, geralmente com inúmeras chamadas telefônicas;
2. Quando o cliente decidia comprar a ideia, ele telefonava para o corretor;
3. Depois de receber as ordens dos clientes, o corretor analisava as ordens e as encaminhava ao mercado;
4. As ordens eram encaminhadas aos locais de troca (bolsas) quando tinham o tamanho mínimo;
5. Ao chegar na bolsa, as ordens eram encaminhadas para os "especialistas" que eram os representantes de intercâmbio;
6. Os corretores informavam os clientes da execução das ordens e recebiam suas comissões e bônus.

Esse processo apresenta vários pontos desinteressantes para os investidores, como as altas comissões aos corretores, erros em ordens de trocas cometidos pelos corretores, atraso de execução de ordens pequenas (ordens esperavam por ordens semelhantes para formar um lote mínimo antes de serem levadas ao mercado), e imparcialidade e preferencia de clientes como grandes empresas [Aldridge 2013].

Além disso, o aumento do tamanho do mercado financeiro tornou inviável o uso da força de trabalho humano (corretores) para assistir os investidores em suas aplicações [Bouchaud et al. 2008].

Figura 1. Fluxos do Mercado Financeiro Adaptado de [Aldridge 2013].



(a) Tradicional. **(b) Moderno.**

Naturalmente, o paradigma de negócios no mercado financeiro mudou de centralizado no corretor, para centralizado no investidor, o que caracteriza o mercado financeiro moderno, apresentado na Figura 1b, que segue os seguintes passos:

1. O cliente gera suas próprias pesquisas e análises quantitativas, com base nas previsões de movimentos dos títulos e suas alocações;
2. O corretor concede privilégios aos clientes para acessar a bolsa diretamente por taxa combinada, assim os clientes utilizam a identificação do corretor para as negociações;
3. Os clientes utilizam seus próprios algoritmos de distribuição e roteamento para executar suas ordens de negociação na bolsa e outros locais de troca;
4. A bolsa, ou outros locais de negociação informam sobre a negociação diretamente ao cliente;
5. O corretor recebe informações sobre as negociações dos clientes e recebem pelo acesso aos privilégios

Para maximizar os lucros, os clientes devem analisar o maior número de ações possível, o que permite a criação de estimativas e modelos de predição, que por sua vez auxiliam na tomada de decisões de compra e venda. Surge então a necessidade da implementação de softwares que analisem grades volumes de dados e criem modelos e predições para auxiliar clientes e corretores a tomar decisões quanto as aplicações no mercado financeiro.

Na ultima década, notou-se que a realização de várias transações em um curto espaço de tempo, prática conhecida como *High-Frequency Trading* (HFT), possuí um grande potencial de lucro, com maior estabilidade e custos menores [Aldridge 2013]. Para habilitar a HFT, há a necessidade da criação de sistemas automatizados de negociação, conhecidos como Robôs de Negociação.

Os robôs de negociação são algoritmos desenvolvidos para automatizar a análise e a negociação nos mercados financeiros. Em um contexto de HFT, os robôs devem analisar grandes volumes de dados, gerar modelos baseados nesses dados, e tomar decisões de compras e vendas em um curto período. O uso de computadores tradicionais para tais aplicações pode demorar horas ou até mesmo dias [Santos 2018], o que inviabiliza a HFT.

Para viabilizar o uso de robôs em tal cenário, vem sido proposto o uso de computadores convencionais associados com aceleradores (GPUs e/ou FPGAs), como exemplificado pela *Maxeler Technologies* em associação com o banco *J.P. Morgan*, que desenvolveram uma arquitetura acelerada por GPUs e FPGAs para realização dos algoritmos *Monte Carlo Pricing* e *Collateralized Default Obligation Pricing* [Weston et al. 2012].

O papel dos aceleradores é executar os gargalos do código, explorando paralelismo e pipelines, de forma que o tempo necessário para se obter uma estimativa (latência) seja menor que um certo limite dado pelo usuário. Para que a latência do sistema não seja comprometida, é necessário que todos custos de comunicação e processamento não tornem a computação no acelerador inviável. Esse projeto tem como foco a criação de uma integração eficiente entre `host` e acelerador, e o uso de ferramentas modernas para a criação dos aceleradores, de forma que os custos de comunicação ou o tempo de processamento não comprometam a latência do sistema.

2. Introdução

Dado o contexto atual em HFT, surge a problemática de como criar aceleradores eficientes, de baixo consumo energético e que atentam as restrições de tempo de um robô de negociações.

Segundo estimativa de [Borkar et al. 2015] a quantidade de dados que as organizações e as pessoas utilizam está dobrando a cada 18-24 meses, este dados devem ser transmitidos, armazenados, organizados e processados em tempo real para serem úteis. Com intuito de acompanhar tal estimativa, o uso de arquiteturas heterogêneas vem sido proposto, onde aceleradores especialmente desenvolvidos para tais aplicações são utilizados [Borkar and Chien 2011].

Diferentes tipos de processadores podem ser utilizados como aceleradores, como CPUs, GPUs e FPGAs, sendo que cada tipo de processador é mais adequado para a realização de tipos diferentes de computação [Castillo et al. 2009]. Assim, uma plataforma heterogênea (que combina mais de um tipo de processador) é ideal para a execução

de códigos complexos como os algoritmos por trás dos robôs de negociação.

Dentre os tipos de aceleradores, uma das vantagens das FPGAs é que podem ser reconfigurados, o que permite a criação de aceleradores específicos para cada aplicação e evita *overheads* desnecessários, resultando em aceleradores de baixa latência [Cardoso et al. 2017].

A criação de aceleradores FPGA, antes conhecida como custosa e lenta, vem sido alavancada pelos esforços na síntese de alto nível, que permite a criação dos aceleradores a partir de linguagens como `C` e `OpenCL`, aumentando a produtividade [Perina 2017]. Nesse projeto propomos a integração da aplicação financeira `MetaTrader5` (MT5) com o `Intel FPGA SDK for OpenCL` (AOCL).

A ferramenta MetaTrader 5 (MT5), desenvolvida pela MetaQuotes Software Corp., é um software para negociações on-line multimercado, que possibilita a análise técnica, o uso de robôs de negociação, e disponibiliza o *Limit Order Book* (LOB) (um registro de valores de ações e transações passadas) [MetaQuotes 2017a].

O MT5 oferece aos investidores várias formas de negociação de ações de empresas, moedas, e índices de ativos. A ferramenta disponibiliza uma série de recursos que podem ser controlados manualmente pelo operador ou totalmente automatizados por meio de robôs de negociação, implementados na linguagem MQL5. O ambiente inclui recursos de simulação (*Strategy Tester*) de algoritmos sobre dados históricos para validar estratégias e recursos para obter dados em tempo real, de forma que sistemas possam analisar e tomar decisões sem a intervenção humana. Assim, o MT5 oferece todo instrumental necessário para a obtenção de dados, criação de modelos e análises, e tomada de decisões pelo usuário ou robô de negociação.

O AOCL é um ambiente de desenvolvimento de aceleradores FPGA, onde o código OpenCL de um acelerador (`kernel`) é compilado em hardware [Intel 2018]. Durante o processo de compilação, várias otimizações são utilizadas para extrair paralelismo, criar-se pipelines [de Souza Rosa et al. 2018], e reduzir o transmissão de dados. O AOCL também contém bibliotecas de emulação, nas quais os *kernels* são compilados e emulados em um PC tradicional, sem a necessidade de síntese de hardware, o que demanda muito tempo para fins de teste [Perina 2017]. Assim, o AOCL permite a criação rápida de aceleradores FPGAs eficientes e de baixa latência, ideias para serem utilizados com robôs de negociação.

A integração das duas aplicações tem como objetivo desenvolver uma interface de comunicação que permitirá a implementação de sistemas automatizados diretamente em hardware. Como resultados, espera-se a padronização da comunicação entre as ferramentas, e a simplificação do uso de FPGAs como aceleradores de sistemas automatizados para o mercado financeiro.

O resto desse artigo se organiza da seguinte forma: A seção 3 apresenta os trabalhos relacionados. A seção 4 apresenta as implementações e a metodologia de desenvolvimento. A seção 5 apresenta os resultados obtidos. Por fim, a Seção 6 conclui o artigo.

3. Trabalhos Relacionados

As duas grandes áreas de pesquisa envolvidas nesse projeto são a integração de softwares com o MT5 e a criação de aceleradores para o mercado financeiro. As seções 3.1 e 3.2 apresentam trabalhos relacionados a essas duas áreas respectivamente.

O foco do trabalho proposto nesse projeto é unir uma aplicação específica (AOCL) ao MT5, que permite o aumento da produtividade de aceleradores para o mercado financeiro. Até onde estamos cientes, esse é o primeiro trabalho que propõe a integração dessas duas aplicações. Assim, a principal contribuição desse projeto impactará criação de aceleradores para o mercado financeiro.

3.1. Trabalhos que abordam a integração com o MT5

MT5 e MatLab [Emelyanov 2010] apresenta a integração do MT5 com o MatLab com as *Dynamic-Link Libraries* (DLLs). O trabalho foi dividido em três blocos principais: o primeiro realiza a preparação preliminar dos dados enviados/recebidos; o segundo realiza a comunicação entre os dois lados, aplicando as conversões necessárias; o terceiro realiza os cálculos conforme os parâmetros definidos no bloco MT5.

MT5 e PrimeXM Liquidity Aggregation Engine [MetaQuotes 2017b] apresenta o PrimeXM como um *gateway* que integra o MT5 com XCore. O *plugin* é apresentado com mecanismos sofisticados de sincronização, que garantem a consistência, enquanto o processamento paralelo permite alta taxa de transferência com baixa latência. Por ser proprietário, não há mais informações sobre a implementação do *plugin*.

3.2. Trabalhos que abordam Aceleradores de Hardware para o Mercado Financeiro

Run-time Reconfigurable Acceleration for Genetic Programming Fitness Evaluation [Funie et al. 2017] demonstra o uso de programação genética para identificar padrões complexos nos mercados financeiros, levando a estratégias comerciais mais avançadas. No entanto, a natureza computacionalmente intensiva dos algoritmos genéticos torna difícil a aplicação a problemas do mundo real, particularmente em cenários restritos em tempo real. Os autores propõem o uso de FPGAs para acelerar o algoritmo genético, acelerando em até $22\times$ o processamento do algoritmo quando comparado com a implementação em software `multithread` otimizada, ao mesmo tempo que obtém resultados financeiros comparáveis.

Multi-Agent Pre-Trade Analysis Acceleration in FPGA [Gerlein et al. 2014] apresenta uma arquitetura multi-agente em FPGAs para aplicações financeiras, que implementa um mecanismo comercial para análise pré-negociação, e um cenário de validação. Os resultados mostram que o cálculo de indicadores técnicos e a avaliação da estratégia de negociação para gerar sinais comerciais com uma latência de $550(ns)$ é realizável.

FPGA Low-Latency Library for HFT [Lockwood et al. 2012] analisa e descreve como FPGAs são utilizados nas negociações eletrônicas para se aproximar do objetivo da latência zero. Além disso, é apresentada uma biblioteca IP FPGA que implementa redes, E/S, interfaces de memória e analisadores de protocolo financeiro. A biblioteca fornece uma infraestrutura pré-construída que acelera o

desenvolvimento e verificação de novas aplicações financeiras. Em um exemplo de aplicação financeira usando a biblioteca obteve-se uma taxa de transferência Ethernet de 10Gb/s com uma latência fixa de ponta a ponta de $1(\mu s)$ (duas ordens de grandeza inferiores às implementações de software comparáveis).

HFT Acceleration Using FPGAs [Leber et al. 2011] apresentam o projeto de um hardware específico para acelerar aplicações de HFT, otimizado para interpretar os *feeds* de dados do mercado, permitindo uma latência mínima. A implementação suporta protocolos Ethernet, IP e UDP, e FAST (protocolo comum para transmitir feeds de mercado) em hardware. Também é desenvolvido um microcódigo, com um conjunto de instruções correspondente, e um compilador, que permite flexibilidade para suportar uma ampla gama de protocolos de negociação aplicados. Os resultados apresentam uma redução de latência de $4\times$ em comparação com a abordagem baseada em software convencional.

Accelerated Montecarlo Financial Simulation Over Low-Cost FPGA Cluster [Castillo et al. 2009] exploram diferentes opções (supercomputador, cluster de FPGAs ou GPUs) para acelerar a computação do problema de preços de opções usando o método de Montecarlo para resolver a fórmula *Black-Scholes*. Os resultados mostram a combinação GPU-CUDA com melhores resultados e ressaltam que em relação à escalabilidade, a GPU apresenta limitações de memória, impedindo o uso de GPUs para executar problemas muito grandes, o que limita severamente sua aplicabilidade.

4. Implementação

Nessa seção são apresentados os passos detalhados para a realização da integração entre o MT5 e o AOCL. Como a integração abrange a troca de dados entre duas aplicações, há a necessidade de gerenciamento pelo sistema operacional que utiliza de `buffers` para acelerar as transferências [Tanenbaum and Bos 2014].

Dentre as classes de `buffers`, os `Named Pipes` [MetaQuotes 2012] se destacam por permitirem a comunicação cliente/servidor entre programas (processos/threads) de forma organizada, uni ou bidirecionalmente.

Todas as instâncias de um `Named Pipe` compartilham o mesmo nome, mas cada instância contém seu próprio *buffer* e *handler*, fornecendo um canal de comunicação separado para cliente/servidor. O uso de instâncias permite que vários clientes utilizem o mesmo `Named Pipe` ao mesmo tempo. O `Named Pipe` permite a comunicação entre processos relacionados ou não relacionados, pois qualquer processo pode acessar o `Named Pipe`, porém, verificações de segurança são utilizadas para controlar o acesso apenas pelos processos desejados [Microsoft 2017].

Portanto, `Named Pipes` são ideais para a implementação da comunicação entre o MT5 e o AOCL, o que é feito através da implementação de dois módulos: O `Named Pipe Servidor`, e o `Named Pipe Cliente`.

A Figura 2 apresenta o diagrama de blocos com os componentes que compõem o projeto. O `Named Pipe Cliente` é implementado dentro to MT5, através de um módulo implementado na linguagem *MetaQuotes Laguage 5* (MQL5). O `Named Pipe Servidor` é implementado dentro da aplicação OpenCL (Visual Studio), que recebe os dados e os repassa para o acelerador FPGA. Por sua vez, a aplicação OpenCL se comunica

com o kernel (acelerador implementado em FPGA) através do barramento PCIe.

Note que tanto o MT5, quando a aplicação OpenCL são implementadas em um computador tradicional (CPU), que recebe o nome de `host`, ao passo que o kernel é executado no acelerador FPGA. Assim, o tempo total gasto para processar os dados em tal plataforma pode ser descrito como:

$$T_{total} = T_{MT5 \rightarrow OpenCL} + T_{OpenCL \rightarrow Kernel} + T_{Kernel} + T_{Kernel \rightarrow OpenCL} + T_{OpenCL \rightarrow MT5}$$

Onde $T_{x \rightarrow y}$ é o tempo para a transferência de dados entre x e y, e T_{Kernel} é o tempo de processamento dos dados no kernel.

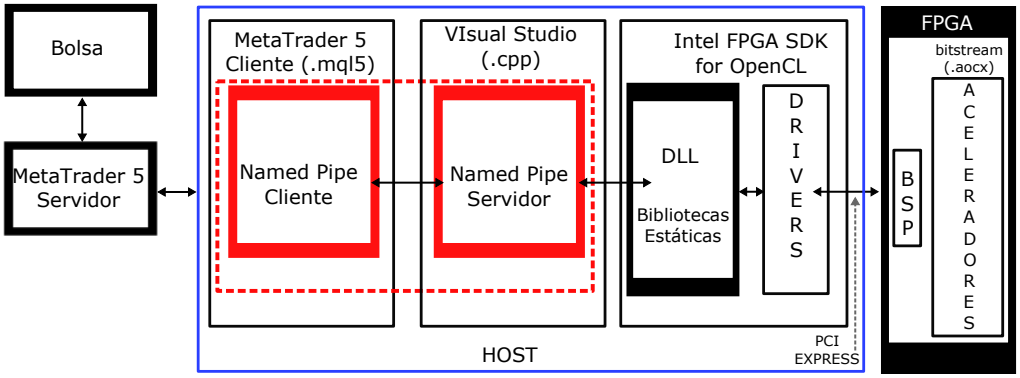


Figura 2. Visão Geral do Projeto

Portanto, para que a aplicação seja acelerada como um todo, deve-se garantir que os tempos de transferência não se tornem gargalos e que a computação no kernel seja rápida.

No estado atual do projeto, foram implementados os módulos `Named Pipe Servidor` e `Named Pipe Cliente` (destacados em vermelho na Figura 2. Isso permite a medição dos tempos de transferência entre o MT5 e a aplicação OpenCL $T_{MT5 \rightarrow OpenCL}$ e vice-versa $T_{OpenCL \rightarrow MT5}$.

O módulo `Named Pipe Servidor` é responsável pela comunicação (troca de dados) entre o AOCL e o MT5, e conversão dos tipos de dados para às chamadas dos métodos e funções. A implementação do `Named Pipe Servidor` é feita com a importação de uma biblioteca e manuseada com funções WinAPI padrões, como:

CreateNamedPipe() cria o Named Pipe;
ConnectConnectNamedPipe() habilita o servidor para aguardar conexões de cliente;
WriteFile() escreve dados no pipe;
ReadFile() faz a leitura dos dados do pipe;
FlushFileBuffers() esvazia os buffers acumulados;
DisconnectNamedPipe() desconecta o servidor;
CloseHandle() fecha o handler.

Após aberto um `Named Pipe` um *handler* de arquivo é retornado para ser utilizado nas operações de leitura e escrita. As funções de leitura e escrita foram implementadas de modo a transmitir dados em binário ou `strings` de texto ANSI de modo compatível com MQL5.

Além de realizar a comunicação com o `Named Pipe Cliente`, o `Named Pipe Servidor` é composto por bibliotecas estáticas e de vínculo dinâmico (DLLs) do AOCL. Para testar a integração foi utilizado o algoritmo `Monte Carlo Black-Scholes Asian Options Pricing` disponibilizado em [Altera 2013].

O módulo `Named Pipe Cliente` foi implementado com a linguagem de programação MQL5 e a classe `CFilePipe`. Esta classe foi utilizada pois contém uma verificação importante da disponibilidade dos dados [MetaQuotes 2012], pois em alguns casos os dados podem não ter sido transferidos ou terem se corrompido.

Para a realização dos testes, o código `kernel` é simulado no `host`, o que não influencia os tempos de transmissão entre o MT5 e o AOCL. Entretanto, não é possível ainda medir o tempo de transmissão entre a aplicação OpenCL e o Kernel e o tempo de processamento dos dados no kernel. A completa integração com o kernel e as medições dos tempos de transmissão e execução são os próximos passos de implementação nesse projeto.

5. Resultados Parciais

Para validar os resultados o `Named Pipe Cliente` foi executado no MT5 no ativo T-8058 (Mitsubishi Corp.) e o `Named Pipe Servidor` no ambiente de simulação do AOCL. A Figura 3 apresenta o console do MT5 com o resultado da troca de dados entre os `Named Pipe Servidor e Cliente`, sendo um teste de conexão e posteriormente são enviados $1024(Mb)$ com uma taxas de transferência acima de $1500(Mb/s)$.

pipeclient (#T-8058,H1)	Client: pipe opened
pipeclient (#T-8058,H1)	Server: Hello from pipe server received
pipeclient (#T-8058,H1)	Server: 1234567890 received
pipeclient (#T-8058,H1)	Client: 1024 Mb received at 1565 Mb per second
pipeclient (#T-8058,H1)	
pipeclient (#T-8058,H1)	Client: pipe opened
pipeclient (#T-8058,H1)	Server: Hello from pipe server received
pipeclient (#T-8058,H1)	Server: 1234567890 received
pipeclient (#T-8058,H1)	Client: 1024 Mb received at 1641 Mb per second

Figura 3. Console Named Pipe Cliente

A Figura 4 apresenta o console do `Named Pipe Servidor` com resultado da troca de dados entre Servidor e `Named Pipe Cliente`, que consiste de um teste de conexão e a transmissão de $1024(Mb)$ de dados do `Named Pipe Cliente`. A taxa de transferência obtida é acima de $1500(Mb/s)$. Considerando que para a realização de uma única simulação são necessários apenas a transferência de $256(b)$, e desprezando o tempo de escalonamento das tarefas do SO, seria possível realizar $6,144E6$ simulações. Portanto, o tempo médio de transferência, por simulação, é de $16,28(\mu s)$, suficiente para aplicações HFT de acordo com [Aldridge 2013].

Após a comunicação entre `Servidor e Cliente` o algoritmo `Monte Carlo Black-Scholes Asian Options Pricing` é executado no `host` (que emula a execução na FPGA). Para cada simulação ($256(b)$), o tempo total processamento no `host` é de $1,21(s)$, resultando em dias para todas simulações, o que corrobora com os resultados apresentados em [Santos 2018].

Esses resultados mostram que o tempo de comunicação não será um gargalo no sistema, entretanto, esses resultados ainda não dizem respeito à latência do sistema. O tempo de execução no `host` é um muito grande para aplicações HFT e comprometeria a latência do sistema como um todo. Isso ressalta a criticidade do uso de aceleradores e a importância da integração do MT5 com o AOCL para a criação destes, que é a proposta deste projeto.

Figura 4. Console Named Pipe Servidor

Nos próximos passos desse projeto serão implementadas a comunicação do host com a FPGA, eliminando a simulação do kernel OpenCL. Então poderão ser medidos os tempos de transmissão entre o host e o kernel, além do tempo de computação dos dados no kernel. Esses resultados serão utilizados para o cálculo da aceleração em comparação com o código executado apenas no host. Finalmente, métricas e otimizações de latência devem ser aplicadas nos protocolos de comunicação, para que se garanta que os requisitos de latência da aplicação não sejam comprometidos.

Como resultado, espera-se que esse caso de teste demonstre a integração do MT5 com o AOCL, os possíveis ganhos computacionais da aceleração de robôs de negociação em plataformas heterogêneas com FPGAs.

6. Conclusão

Esse projeto tem como objetivo realizar a integração entre o Intel FPGA SDK for OpenCL e o MetaTrader 5, realizando a comunicação e facilitando o uso de FPGAs como aceleradores de sistemas automatizados para o mercado financeiro.

Conclui-se que é possível a integração entre o Intel FPGA SDK for OpenCL e o MetaTrader 5 utilizando Named Pipes.

Com relação a troca de dados (vazão de dados), mesmo ainda sendo em um ambiente de simulação obteve-se um resultado satisfatório com uma taxa acima de $1500(Mb/s)$. Os tempos de processamento no `host` mostram que o uso de computadores tradicionais para tais aplicações é inviável, relevando a necessidade de aceleradores.

Como resultado da integração completa dos dois ambientes, espera-se que se obtenha uma taxa de transferência de dados suficiente para não comprometer a latência das

computação, e também a computação eficiente dos dados nos aceleradores FPGA, demonstrando sua usabilidade para HFT.

Referências

Aldridge, I. (2013). *High-frequency trading: a practical guide to algorithmic strategies and trading systems*. Wiley trading. Wiley, 2 edition.

Altera, C. (2013). Monte carlo black-scholes asian options pricing design example. Acessado 10 jan. 2018.

Borkar, S. and Chien, A. A. (2011). The future of microprocessors. *Commun. ACM*, 54(5):67–77.

Borkar, S. Y., Dubey, P., Kahn, K. C., Kuck, D. J., Mulder, H., Ramanathan, E. R. M., Thomas, V., Corporation, I., and Pawlowski, S. S. (2015). Intel ® processor and platform evolution for the next decade.

Bouchaud, J.-P., Doyne Farmer, J., and Lillo, F. (2008). How markets slowly digest changes in supply and demand. arXiv.

Cardoso, J. M., Coutinho, J. G. F., and Diniz, P. C. (2017). Chapter 7 - targeting heterogeneous computing platforms. In Cardoso, J. M., Coutinho, J. G. F., and Diniz, P. C., editors, *Embedded Computing for High Performance*, pages 227 – 254. Morgan Kaufmann, Boston.

Castillo, J., Bosque, J. L., Castillo, E., Huerta, P., and Martinez, J. I. (2009). Hardware accelerated montecarlo financial simulation over low cost fpga cluster. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8.

de Souza Rosa, L., Bouganis, C., and Bonato, V. (2018). Scaling up modulo scheduling for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1.

Emelyanov, A. (2010). Metatrader 5 and matlab interaction. Acessado 08 jan. 2018.

Funie, A.-I., Grigoras, P., Burovskiy, P., Luk, W., and Salmon, M. (2017). Run-time reconfigurable acceleration for genetic programming fitness evaluation in trading strategies. *Journal of Signal Processing Systems*.

Gerlein, E., McGinnity, T. M., Belatreche, A., Coleman, S., and Li, Y. (2014). Multi-agent pre-trade analysis acceleration in fpga. In *2014 IEEE Conference on Computational Intelligence for Financial Engineering Economics (CIFEr)*, pages 262–269.

Intel (2018). Intel® fpga sdk for opencl™ overview. Acessado 10 jan. 2018.

Leber, C., Geib, B., and Litz, H. (2011). High frequency trading acceleration using fpgas. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 317–322.

Lockwood, J. W., Gupte, A., Mehta, N., Blott, M., English, T., and Vissers, K. (2012). A low-latency library in fpga hardware for high-frequency trading (hft). In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, pages 9–16.

MetaQuotes (2012). Communicating with metatrader 5 using named pipes without using dlls. Acessado 11 dez. 2017.

MetaQuotes (2017a). Metatrader 5.

MetaQuotes (2017b). Metatrader 5 integration with the primexm liquidity aggregation engine. Acessado 09 jan. 2018.

Microsoft (2017). Named pipes. Acessado 03 jan. 2018.

Perina, A. B. (2017). Data mining for kernel mapping in a hybrid platform composed by fpga, gpu and manycore. Exame de Qualificação.

Santos, R. R. d. (2018). Um framework para agrupar funções com base no comportamento da comunicação de dados em plataformas multiprocessadas. Dissertação de Mestrado.

Tanenbaum, A. S. and Bos, H. (2014). *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition.

Weston, S., Spooner, J., Racanière, S., and Mencer, O. (2012). Rapid computation of value and risk for derivatives portfolios. *Concurrency and Computation: Practice and Experience*, 24(8):880–894.

Uso de Ferramenta de *Autotuning* para ajuste nas dimensões de *kernels* em Dispositivos Aceleradores (GPUs)

João Martins de Queiroz Filho[1], Rogério Aparecido Gonçalves[1] e Alfredo Goldman[2]

[1]Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brasil

[2]Instituto de Matemática e Estatística (IME)
Universidade de São Paulo (USP)
São Paulo – SP – Brasil

joaomfilho1995@gmail.com, rogerioag@utfpr.edu.br, gold@ime.usp.br

Abstract. *Accelerator devices have been widely used in Parallel Computing. In GPUs, one of the main difficulties is in the fact that the programming model leaves to the programmer the responsibility of the thread arrangement structure definition. Which result in generic definitions and performance degradation. The purpose of this work is to analyze the process of searching for configurations of grids and blocks of threads that present the best performance. To do this, some benchmarks were used, and they were testes with their configurations and the kernels using the OpenTuner. The best configuration is found based on metrics retrieved from GPUs using the nvprof profiling tool. In this work, we present the results obtained in a test with one of the codes with nested loops.*

Resumo. *Dispositivos aceleradores tem sido amplamente utilizados em Computação Paralela. Usando GPUs, uma das principais dificuldades está no fato do modelo de programação deixar a definição da estrutura do arranjo de threads sob a responsabilidade do programador, o que resulta em definições genéricas e na degradação do desempenho. O objetivo deste trabalho é analisar o processo de busca por configurações de grids, blocos e threads que apresentem o melhor desempenho. Para isso foram utilizados alguns benchmarks, que foram testados com suas configurações e os kernels utilizando-se a ferramenta OpenTuner. A melhor configuração é encontrada com base em métricas recuperadas das GPUs utilizando a ferramenta de perfilamento nvprof. Neste trabalho, apresentamos os resultados obtidos em um teste com um dos códigos com loops aninhados.*

1. Introdução

O uso de dispositivos aceleradores como GPUs em conjunto com CPUs *multicore* tem se intensificado na área de Computação Paralela. As GPUs disponibilizam centenas de núcleos *(cores)* que executam um determinado código através de centenas de *threads* concorrentes [STRINGHINI et al. 2012]. As GPUs da NVIDIA possuem diversas arquiteturas (Fermi, Kepler, Maxwell e Pascal), que introduziram melhorias que vão desde o aumento do número de multiprocessadores, do número de núcleos à quantidade de memória.

O modelo de programação de aplicações de propósito geral e científicas divide a carga de trabalho entre o elemento principal de processamento CPU e lança a execução de regiões de códigos paralelizáveis *(kernels)* para serem aceleradas em GPUs [Kirk and Hwu 2010]. O lançamento da execução de funções *kernel* possibilita o *offloading* de código do *host* para o *device*. No modelo clássico de execução, quando um *kernel* é lançado o controle é passado ao dispositivo e a CPU retomará o controle a cada término da chamada a um *kernel*. Em arquiteturas com suporte a *paralelismo dinâmico*, é possível fazer chamadas a outros *kernels* (ativações) a partir de um *kernel* em execução.

Nesse contexto, o processo de desenvolvimento de *software* traz aos desenvolvedores uma série de desafios e restrições que precisam ser mitigadas [Gaster et al. 2011]. Uma das principais restrições é que os *kits* abstraem o acesso aos recursos do *hardware* até certo nível, exigindo que o desenvolvedor declare explicitamente transferências de dados e as dimensões do arranjo de *threads* que deve ser criado para a execução de um *kernel*. Estas são questões que muitas vezes levam à utilização de definições genéricas, ou a escolha de configurações aleatórias ou com valores padrão, o que pode subutilizar os recursos e o poder de processamento. Assim, nossa proposta está relacionada com o ajuste de código às características arquiteturais e a busca no conjunto de configurações válidas para *grids*, *blocos* e *threads*.

Na Seção 2 apresentamos a ideia relacionada ao trabalho sobre as dimensões de *kernels* e os possíveis ajustes que podem ser feitos na execução são apresentados nas Seção 3. A metodologia utilizada no trabalho é apresentada na Seção 4. Os experimentos realizados e resultados obtidos estão na Seção 5. Finalmente, na Seção 6 são discutidas algumas conclusões obtidas e as próximas etapas do trabalho.

2. Dimensões de *Kernels*

Para o lançamento da execução de um *kernel* é necessário que seja especificada a configuração do arranjo de *threads* que será criado na ativação da função *kernel*. As dimensões do arranjo é especificado entre $<<< \#blocos, \#threads >>>$. Cada uma das *threads* irá executar uma cópia do código do *kernel* e possuirá um id global único que pode ser calculado com base em variáveis embutidas definidas pela plataforma CUDA e que são acessíveis dentro do código da função *kernel*.

A estrutura do arranjo e quais dimensões utilizar para compor a solução fica sob a responsabilidade ou critério do programador, pois varia conforme o domínio do problema. Também há restrições com base nas especificações de cada dispositivo (GPU).

Uma configuração completa é formada pelas dimensões de $grid(gx, gy, gz)$ e $bloco(bx, by, bz)$, o que define a estrutura de um arranjo de $gx \times gy \times gz \times bx \times by \times bz$ *threads*. Para o desenvolvedor utilizar as 6 dimensões possíveis com a combinação de *grid* e *bloco* deve-se respeitar algumas restrições, como por exemplo, que número de *threads* por bloco ($bx \times by \times bz \leq 1024$) não deve ultrapassar o limite máximo para o dispositivo.

Assim, ajustar o código dos *kernels* para a arquitetura alvo é essencial para se garantir a utilização eficiente dos recursos disponíveis. Entretanto, não é uma tarefa trivial, pois exige um certo conhecimento sobre os componentes arquiteturais da GPU e sobre como ajustar o código para que alcance o melhor desempenho possível.

3. Ajustes do lado do *hardware* e do *software*

Para verificar se a escolha de uma determinada configuração ou se a priorização de uma das dimensões influencia no desempenho é necessário testar alguns ajustes tanto do lado do *hardware* tanto para o *software*.

No **lado do *hardware*** é necessário definir as dimensões do arranjo de *threads* composto por `grid` e `bloco`. Por exemplo, para o número de 128 iterações de um laço a serem transferidas para os ids das *threads* do arranjo, temos 210 configurações possíveis. Desta forma, a escolha da melhor configuração que se ajusta às características do *hardware* torna-se um problema e para solucioná-lo seria necessário testar todas as possíveis configurações em uma abordagem por força bruta.

No **lado do *software*** teríamos o código resultante com a retirada das iterações do laço considerado. O Código 1 faz o cálculo de uma tabela de senos e cossenos. Note que o *statement* (S_1) que pertence ao corpo do laço mais interno é totalmente livre de dependências, pois não há acessos a elementos entre iterações. Desta forma, iterações de um ou mais laços poderiam ser migradas para o arranjo de *threads*.

Código 1. Exemplo de algoritmo que possui laços aninhados.

```
1 for (i = 0; i < nx; ++i) {
2   for (j = 0; j < ny; ++j) {
3     for (k = 0; k < nz; ++k) {
4       indice = (i * ny * nz) + (j * nz) + k;
5       xy[indice] = sin(x[indice]) + cos(y[indice]); // S_1
6     }
7   }
8 }
```

Como a ideia de ajuste entre *software* e *hardware* é de migrar dimensões do domínio de iteração formado pelos laços aninhados para dimensões do arranjo de *threads*, o Código 1 pode ser transformado em versões de *kernels*. As dimensões dos laços aninhados são retiradas e transferidas para o arranjo, por exemplo para o `kernel_0` os 3 laços aninhados foram transferidos e assim as versões são formadas até o `kernel_3` que não apresenta nenhum laço, apenas o corpo do laço mais interno. Esse processo de migração possibilita a busca de um equilíbrio entre as opções que executam menos *threads* com mais computação ou mais *threads* com menos computação.

Contudo, deve-se considerar a computação inserida pelo cálculo do id global de cada *thread*. Pois quanto mais dimensões utilizadas, mais operações serão necessárias para o cálculo do id global (cálculo de linearização). O Código 2 apresenta a versão de *kernel* para o `sincos` no qual foi mantido apenas o laço mais interno, as iterações dos outros dois laços foram migradas para o arranjo de *threads*.

Código 2. Exemplo de kernel do bechmark sincos com apenas o laço mais interno

```
1 __global__ void sincos_kernel_1(DATA_TYPE* x, DATA_TYPE* y, DATA_TYPE* xy, int
    nx, int ny, int nz, int funcId) {
2   int i, k, indice;
3   i = getGlobalIdFunc[funcId]();
4   for (k = 0; k < nz; k++) {
5     indice = (i * nz) + k;
6     xy[indice] = sin(x[indice]) + cos(y[indice]);
7   }
8 }
```

A chamada ao *kernel* recebe o índice da função (`funcId`) que deve ser utilizada para o cálculo do id global da *thread*. A chamada `getGlobalIdFunc[funcId]()` é feita utilizando o ponteiro de uma das 63 versões de funções de cálculo do id definidas. O valor de `funcId` é definido com base no número de dimensões utilizadas na composição do arranjo de *threads*. Por exemplo, para a execução com a configuração $(1, 5, 20, 16, 32, 2)$, são verificadas quais dimensões apresentam valores maiores que 1, formando o número $011111_2 = 31$, logo a função a ser utilizada é a de `funcId = 31`, que considera as dimensões y e z no *grid* e x, y, z para os blocos.

Vale destacar que a escolha da função de cálculo do id global da *thread* é importante para evitarmos computações desnecessárias, uma vez que quando maior o número de dimensões utilizadas, mais operações serão necessárias para a linearização do valor do id. Por exemplo, a função `getGlobalIdx_grid_3D_xyz_block_3D_xyz()` que possui funcId = 63, terá no total 14 operações aritméticas (5 adições e 9 multiplicações), enquanto que o número de operações pode ser reduzido se o número de dimensões necessárias for ajustado. O código com todas as versões de funções para o cálculo do *id* global da *thread* e os dados dos experimentos apresentados neste artigo estão disponíveis no `GitHub`[1].

4. Metodologia

Como o objetivo é avaliar o ajuste de *kernels* tanto do lado do *software* quanto do lado do *hardware*, inicialmente, foram escolhidos *benchmarks* que possuem três ou mais laços aninhados. O que utilizaremos como exemplo aqui é o `sincos` que faz o cálculo de uma tabela de senos e cossenos.

Diante da complexidade de se testar o conjunto de configurações para um número grande de iterações utilizando-se força bruta, optou-se pela utilização de ferramentas de otimização de *software (auto-tuning)*, a ferramenta utilizada neste trabalho foi o `OpenTuner` [Ansel et al. 2014].

O `OpenTuner` é uma ferramenta de código aberto, baseado em *scripts* na linguagem `Python`, utiliza técnicas de busca intercaladas, sendo que as técnicas colaboram para a convergência da busca no espaço de soluções. Na Figura 1 é apresentado como o `OpenTuner` realiza o processo de *auto-tuning*.

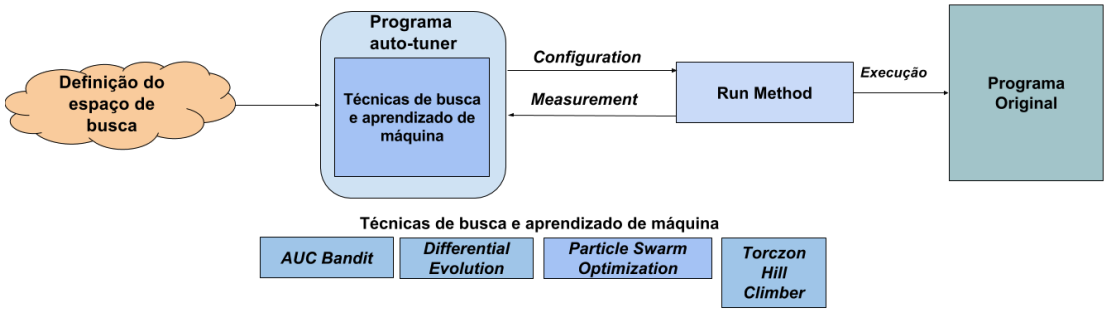


Figura 1. Funcionamento do OpenTuner. Fonte: [Ansel et al. 2014]

Primeiro deve ser especificado o espaço de soluções, este espaço é os parâmetros que o `OpenTuner` irá explorar utilizando as técnicas de busca. Segundo, o usuário

[1] `https://github.com/jfilhoGN/uso_autotuning_ajuste_dimensoes_gpu`

deve implementar os métodos especificados pela `API` do `OpenTuner` como o método *configuration* que faz as escolhas dos parâmetros e o método *run* que executa o programa alvo com os parâmetros e que após o término da execução coleta as medidas da métrica utilizada no teste.

Após a execução dos testes, o `OpenTuner` devolve a melhor configuração de execução para o *kernel* de acordo com o melhor resultado obtido pela métrica.

5. Experimentos e Primeiros Resultados

Nos experimentos foram utilizadas duas `GPUs` de arquiteturas `Kepler` (`GPU GTX 780`) e `Pascal` (`GPU TitanX`). Foram escolhidas as métricas a serem coletadas para a classificação das configurações.

A métrica *sm_efficiency* verifica o tempo que pelo menos um *warp* está ativo e *achieved_occupancy* que faz a média entre os *warps* ativos com o máximo número de *warps*. Estas métricas são coletadas do *hardware* utilizando-se a ferramenta de perfilamento `nvprof` da `NVIDIA`. O `nvprof` permite coletar e visualizar informações sobre a execução de *kernels* e das transferências entre as memórias. Para estes experimentos foram comparados os testes por força-bruta e o uso da ferramenta `OpenTuner`.

5.1. Experimento `sincos` por força-bruta

A dificuldade encontrada nesse experimento foi o tempo de processamento de todas as configurações. Uma vez que para $N = 320$ são geradas 132 mil execuções de testes. A execução dos testes por força bruta demorou em torno de 2 dias e foi utilizada a `GPU TitanX`.

Neste teste foi utilizado o tamanho de $N = 320$, sendo que a configuração que obteve menor resultado foi $(200, 1, 8, 8, 4, 2)$ para a métrica *sm_efficiency* em $50,8\%$. Já a melhor configuração $(80, 40, 1, 1, 8, 4)$ obteve *sm_efficiency* em 99.25%, mostrando também que o *kernel_1* é mais eficiente que os demais *kernels* para o algoritmo `sincos`.

Comparando os valores encontrados por força-bruta e por *autotuning* podemos ver que, utilizando força-bruta foi encontrada a configuração $(1600, 2, 1, 1, 8, 4)$ com eficiência de *warps* em 99.47%. Com o uso do `OpenTuner` foi encontrado a configuração $(2, 160, 10, 4, 2, 4)$ com eficiência também de 99.47%, evidenciando que o uso da ferramenta de *autotuning* encontra as melhores configurações de uma maneira rápida e correta.

5.2. Experimento `sincos` pelo `OpenTuner`

Os valores apresentados na Tabela 5.2 apresentam os resultados obtidos com a `GPU TitanX` e com a `GTX 780`. Todas as configurações alcançaram valores superiores a 90%, que são consideradas as melhores configurações por atingirem valores próximos a 100%. Para o mesmo tamanho de $N = 320$, a ferramenta `OpenTuner` encontrou a melhor configuração com 5 mil testes e gastou em torno de 4 horas, mostrando a melhora no desempenho da busca por uma solução no espaço de configurações.

Ainda na Tabela 5.2 também pode ser visto que foi a única métrica que os melhores resultados foram encontrados com o *kernel_1*, sendo ele as interações dos dois laços mais internos transferidos para o arranjo de *threads*. Os valores dos blocos muito

Tabela 1. Valores obtidos para *sm_efficiency* utilizando *kernel_1*

N	GTX 780 - Kepler			TitanX - Pascal		
	Kernel	**Configuração**	**Resultado**	**Kernel**	**Configuração**	**Resultado**
64	1	(1,16,8,1,8,4)	97.44%	1	(2,16,4,8,4,1)	93.02%
96	1	(4,3,1,4,32,6)	98.9%	1	(144,1,2,2,2,8)	99.63%
288	1	(72,18,2,2,2,16)	99.81%	1	(1296,2,1,1,8,4)	99.60%
320	1	(5,160,4,16,1,2)	99.85%	1	(2,160,10,4,2,4)	99.47%

próximos (valores entre $2, 4$ e 8), como exemplo as configurações $(1, 2, 784, 2, 4, 4)$ e $(32, 16, 1, 1, 4, 8)$ nos dispositivos `TitanX` e `GTX 780`, respectivamente, demonstram que pode haver uma possível tendência de configuração.

6. Conclusões e Trabalhos Futuros

Os resultados sugerem que o uso de ferramentas de *autotuning* é suficiente para realizar a busca pela melhor configuração em um grande espaço de possíveis soluções, ainda mais considerando que para determinados tamanhos a busca por força-bruta torna-se inviável.

Para continuação deste trabalho será feita uma análise do *log* das escolhas feitas pela ferramenta de *autotuning* durante o processo de busca. As escolhas feitas serão analisadas no intuito de encontrarmos padrões que indiquem uma possível ordem de precedência na utilização das dimensões que propiciem um melhor desempenho conforme a métrica considerada.

Agradecimentos

Os autores agradecem à `NVIDIA` pela doação de uma `GPU Titan X Pascal` através do `GPU Grant Program` para o projeto *Estudo Exploratório sobre Técnicas e Mecanismos para Paralelização Automática e Offloading de Código em Sistemas Heterogêneos* (UTFPR 916/2017).

Referências

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada.

Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. (2011). *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

STRINGHINI, D., GONCALVES, R., and GOLDMAN, A. (2012). Capítulo 7: Introdução à computação heterogênea. In *Luiz Carlos Albini, Alberto Ferreira de Souza, Renata Galante, Roberto Cesar Junior, Aurora Pozo. (Org.). XXXI Jornadas de Atualização em Informática.*, volume 21, pages 262–309, Curitiba, Paraná, Brazil. Sociedade Brasileira de Computação (SBC).

Aceleração por Hardware para solução das equações de Black-Scholes por método Monte Carlo

Thadeu Antonio Ferreira de Melo[1], **Erinaldo Pereira Silva**[1], **Eduardo Marques**[1]

[1]Instituto de Ciências Matemáticas e Computação – Universidade de São Paulo (USP)

Abstract. *The increasing automation of operations in the financial market has led investors to seek high-performance computing systems. In this competitive environment the demand for more powerful and energy efficient systems often comes up against the limitations of traditional architectures. Hardware Acceleration offers the possibility of performance gains and scalability. One This paper presents the work-in-progress summary and the initial results of hardware accelerators for Monte Carlo methods in solving the Black-Scholes equations for option pricing.*

Resumo. *A crescente automatização de operações no mercado financeiro tem levado investidores a buscar sistemas computacionais de alta performance. Nesse ambiente competitivo a demanda por sistemas mais poderosos e energeticamente eficientes esbarra muita vezes nas limitações das arquiteturas tradicionais. Aceleração por hardware oferece a possibilidade de ganhos de performance e escalabilidade. Este artigo apresenta o resumo do trabalho em andamento e os resultados iniciais de aceleradores de hardware para métodos Monte Carlo na solução das equações Black-Scholes para a precificação de opções.*

1. Introdução

O artigo publicado em 1973 por Fisher Black e Miron Scholes é considerado um dos marcos das ciências econômicas na segunda metade do século 20. Nesse trabalho, o trio conseguiu derivar um conjunto de equações que podem precificar de modo determinístico o preço de opções negociadas nas bolsas de valores do mundo todo. Os investidores não perderam tempo em usar Black-Scholes e já em 1977 elas já estavam implementadas em calculadoras eletrônicas de bolso para certos tipos de opções.

Um dos modos de solução as equações de Black-Scholes é por métodos Monte Carlos. Tais métodos oferecem uma maior flexibilidade e variabilidade do que soluções analíticas, que dependem de condições de contorno limitantes para suas soluções. Métodos Monte Carlo também podem incluir técnicas de Cadeias de Markov ou outras dependências no comportamento do mercado financeiro.

Métodos Monte Carlo, entretanto, são computacionalmente intensos, necessitando um grande volume de simulações para a obtenção de resultados com precisão boa o suficiente para a aplicação em questão. Em alguns casos, como para física quântica, em que várias dezenas de casas decimais de significância são necessárias o volume de simulações pode chegar à ordem de centenas de trilhões de simulações executando por dias em supercomputadores. Já para aplicações que lidam com informações em tempo real, como a precificação de ativos no mercado financeiro, o volume de simulações precisa ser equilibrado dentro de uma janela de tempo limitada.

Este artigo demostra as estratégias para o desenvolvimento de aceleradores de hardware em FPGA para a solução das equações de Black-Scholes por métodos Monte Carlo. O objetivo desses geradores é possibilitar ganhos de performance, em números de simulações por segundo, o que por consequência proporcionam precificação mais acurada. Foram avaliadas duas tecnologias para desenvolvimento, BlueSpec e Intel FPGA OpenCL. No primeiro caso, apesar de oferecer hardwares mais eficientes a dificuldade de integração com um sistema completo indica que OpenCL é uma alternativa mais adequada.

2. Síntese de Alto Nível

Um grande desafio para se obter os ganhos de performance possível em FPGA é a própria programabilidade. O modelo de programação em FPGAs geralmente utilizam práticas de RTL (register-transfer level), o que demandam significativo conhecimento de hardware para aceleradores de microarquitetura como para controladores, fluxo de dados e maquinas de estados finitos [Keating and Bricaud 2007]. Isso cria uma barreira de entrada para a grande maioria de programadores treinados em paradigmas de ambientes de software. Esse problema é ainda mais agravado quando o ambiente está em constante evolução; i.e. em um meio competitivo como o mercado financeiro o algoritmo a ser acelerado pode se tornar obsoleto no tempo que se leva para desenvolver o hardware.

Décadas de trabalho e pesquisa foram dedicadas nas ferramentas para facilitar a programabilidade das FPGAs. Hoje em dia linguagens para Síntese de Alto Nível (HLS - High Leve Sintesis)[Cong et al. 2011] são utilizadas para desenvolvimento de aceleradores de hardware a partir de códigos mais próximos dos padrões tradicionais de software, como C/C++ e Java. Como exemplos mais avançados dessas tecnologias é possível citar Xilinx Vivado HLS e Intel OpenCL SKD. Uma outra alternativa, mais próxima das linguagens de descrição de hardware tradicionais é a linguagem BlueSpec System Verilog; que oferece estruturas semelhantes à C/C++ ao mesmo tempo que possibilita o controle de sinais e máquinas de estados.

3. Geradores de Números Pseudo-Aleatórios

A base para qualquer boa solução por método Monte Carlo está no uso de Geradores de Números Pseudo-Aleatórios (PRNG na sigla em inglês) de qualidade adequada para o problema[Gamerman and Lopes 2006]. O tópico de PRNGs é vasto e estudado desde o princípio da computação eletrônica [Knuth 1997]. Grandes avanços foram obtidos nas ultimas décadas com a introdução de PRNGs do tipo LFSR (Linear Feedback Shift Registors), que possibilitam grande performance em arquiteturas de computadores modernas.

A distribuição gerada por LFSR é uniforme em inteiros [1, 2^n], onde n é o tamanho do registrador. Os números podem então serem normalizados para o intervalo]0,1] na representação float da arquitetura alvo. A estrutura desses geradores também pode ser vantajosa para implementação em hardware pois grande parte das operações são combinacionais e podem ser desenroladas em um único ciclo.

Em termos de qualidade para métodos Monte Carlos os PRNGs ideais precisam passar por pelo menos os seguintes critérios:

- Independência linear - Tanho do espaço de estados que o gerador é linearmente independente.

- Longo período - Referente ao tamanho da sequencia de números gerados até que ela comece a se repetir.
- Equidistribuição - Referente ao ”bom”espalhamento dos números gerados em sequência (sem aglomeração).

PRNGs que cumpram esses critérios podem ser utilizados de modo paralelo sem comprometimento de qualidade dá série. Essa propriedade possibilita ainda o uso de diferentes tipos de geradores ou geradores do mesmo tipo inicializado com sementes diferentes em paralelo.

Atualmente o PRNG mais popular entre as ferramentas de análise numérica é o gerador Mersenne Twister (MT19937) e seus derivados. Sendo esse o gerador padrão do Matlab e com implementações disponíveis para bibliotecas mais populares de C/C++ e Python. Em [Bonato et al. 2013] o gerador Mersenne Twister foi implementado tanto em VHDL quanto em BlueSpec e integrado em um sistema embarcado para filtro de partículas.

Mais recentemente os autores do Mersenne Twister propuseram uma nova variação de geradores LFSR chamado WELL. Esses gerados são mais flexíveis que o MT19937, com total independência de tamanho de a palavra (registrador) e implementação mais simples e direta. Além dessas vantagens os geradores WELL também preservam as qualidades estatísticas citadas anteriormente. Neste projeto nós expandimos a estrutura desenvolvida em [Bonato et al. 2013] para MT19937 para duas versões do gerador WELL em BlueSpec.

Como o BSV possibilita o uso de sintaxe muito semelhante ao C, assim como os conceitos de tipo de variáveis, foi possível fazer uma tradução quase de um para um entre os códigos C e o e BSV - vistos nos trechos de códigos a seguir, Códigos Fontes1 e 2.

Código-fonte 1. Trecho Well512 em BlueSpec

```
z0 = state[(st_i+15)&'h0000000f];
z1 =
   mat0neg(-16,st[st_i])^mat0neg(-15,st[st_i+13&'h0000000f]);
z2 = mat0pos(11,st[(st_i+9)&'h000000f]);
st[st_i] = z1 ^ z2;
st[(st_i+15)&'h000000f] = mat0neg (-2,z0)^ mat0neg(-18,z1)^
   mat3neg(-28,z2)^mat4neg(-5,'hda442d24,st[st_i]) ;
st_i = (st_i+15)&'h0000000f;
```

Código-fonte 2. Trecho Well512 em C

```
z0   = VRm1;
z1   = MAT0NEG (-16,V0)^MAT0NEG (-15, VM1);
z2   = MAT0POS (11, VM2) ;
newV1 = z1 ^ z2;
newV0 = MAT0NEG (-2,z0) ^ MAT0NEG(-18,z1)^ MAT3NEG(-28,z2) ^
   MAT4NEG(-5,0xda442d24U,newV1) ;
state_i = (state_i + 15) & 0x0000000fU;
```

A implementação dos PRNGs em OpenCL foi ainda mais direta, pois OpenCL foi definido por padrão a partir das especificações da linguagem C99. A grande vantagem

do OpenCL sobre BlueSpec se mostrou na integração com outros submódulos e com o programa principal de para a solução do algoritmo.

4. Transformada Box-Muller

Para a resolução das equações de Black-Scholes por Monte Carlo é necessário que a distribuição dos geradores seja uma distribuição Gaussiana Normal. É ai que entra a Transformada Box-Muller, que é capaz de gerar uma distribuição Normal a partir de uma distribuição aleatória uniforme.

A definição formal da transformada Box-Muller é: se U_1 e U_2 forem pares de números de uma série com distribuição uniforme e independente entre 0 e 1, então z_1 e z_2, definidos por 1, terão uma distribuição normal com média $\mu = 0$ e variância $\sigma^2 = 1$.

$$\begin{aligned} z_1 = R\cos(\Phi) &= \sqrt{-2\ln(U_1)}\cos(2\pi U_2) \\ z_2 = R\sin(\Phi) &= \sqrt{-2\ln(U_1)}\sin(2\pi U_2) \\ R^2 &= -2\ln U_1 \\ \Phi &= 2\pi U_2 \end{aligned} \tag{1}$$

A forma polar dessa transformada é mais interessante para o uso em GPUs modernas e implementações de hardware, pois não precisa utilizar as funções seno e cosseno. Com z_1 e z_2 definidos por 2

$$\begin{aligned} z_1 &= \sqrt{-2\ln(U_1)}\cos(2\pi U_2) = \sqrt{-2\ln(s)}(u/\sqrt{s}) = u\sqrt{-2\ln(s)/s} \\ z_2 &= \sqrt{-2\ln(U_1)}\sin(2\pi U_2) = \sqrt{-2\ln(s)}(v/\sqrt{s}) = v\sqrt{-2\ln(s)/s} \end{aligned} \tag{2}$$

A forma polar da transformada pode ser obtida definindo que: u e v de uma serie de números aleatórios uniformemente distribuídos independentes no intervalo entre [-1,+1] tendo $s = R^2 = u^2 + v^2$, descartando-se u e v caso $s \geq 1$ verifica-se com outro par (u, v) que cumpra a condição. Já que u e v são valores uniformemente distribuídos e apenas pontos dentro do circulo unitário são admitidos, os valores de s também serão uniformemente distribuídos entre [0,1]. Identificando o valor de s com o de U_1 e $\cos\theta/(2\pi)$ com o de U_2 na forma básica. Os valores de $\cos\theta = \cos(2\pi U_2)$ e $\sin\theta = \sin(2\pi U_2)$ na forma básica podem ser substituídos pelas proporções $\cos\theta = u/R = u/\sqrt{s}$ e $\sin\theta = v/R = v/\sqrt{s}$, respectivamente.

5. Resultados Experimentais

A tabela 1 apresenta os resultados dos hardwares dos geradores de números pseudo-aleatório codificados em BlueSpec na placa Stratix V. O relatório é gerado durante a compilação do código HDL descrevendo o hardware e na operação de *Fitter* pela IDE Quartus II. Os dados relevantes para avaliação são frequência máxima (em mega herz) e área da FPGA ocupada pelo hardware (em número de elementos lógicos - LE).

Os hardwares para as duas versões de geradores WELL passaram nos testes de qualidade da biblioteca TestU01 como esperado. A sequência de números inteiros também foi a mesma esperada das implementações em software para a mesma semente

Tabela 1. Relatório de compilação para Stratix V

	Freq. Max (MHz)	Área (EL)
WELL512a	237.42	1.164
WELL1024	189.0	2.490
MT19937	190.84	5.058

inicial. O resultado se manteve consistente para todas as sementes e para os intervalos da sequência testados.

Considerando a frequência máxima obtida pelo gerador WELL512a, sendo um número produzido por cíclo de clock o *throughput* máximo é de aproximadamente 237 milhões de números por segundo. Já a transformada Box-Muller em BlueSpec utilizou cerca de 15mil elementos lógicos e frequência máxima de 120 MHz. Entretanto as outras operações necessárias para a geração da simulação em ponto flutuante reduziram a performance para menos de 100 mil simulações por segundo. Outro fator limitante para as simulações com aclaradores em BlueSpec é a forma de transmissão de dados entre a placa FPGA e o sistema host executando em CPU.

Já a implementação do mesmo gerador WELL512a em OpenCL utiliza 5 mil elementos lógicos(EL) a uma frequência máxima de aproximadamente 170MHz. A penalidade em área e performance é algo que ainda precisa ser melhor avaliado em nossa pesquisa, mas os resultados iniciais estão em conformação com a nossa expectativa. A transformada Box-Muller em OpenCL utilizou 18 mil EL com frequência máxima de 80MHz.

Com a integração do gerador, transformada Box-Muller em OpenCL executando na placa Stratix V e o código host em C++ para solucionar Black-Scholes a performance geral ficou por volta de 70 milhões de simulações por segundo em um sistema com memória dividida entre a CPU e FPGA.

6. Conclusões

Nosso trabalho indica que aceleradores de hardware utilizando FPGAs podem ser desenvolvidos a partir de códigos de alto nível em C ou C++. A performance desses aceleradores depende do tipo de aplicação, favorecendo aquelas que fazem uso intenso de repetições (loops) com pouca dependência de dados e desvio de fluxo. Nesse caso simulações para métodos Monte Carlo se mostram ideal, com ganhos de performance na ordem de 50 vezes sobre software.

A flexibilidade oferecida por HLS como OpenCL permite aplicações possam ser desenvolvidas de modo iterativo e muito mais rapidamente em comparação com HDLs tradicionais. Aplicações como essas são abundantes e variadas no mercado financeiro atual, em que cada micro-segundo de vantagem pode oferecer retornos monetários significativos.

Referências

Bonato, V., Mazzotti, B. F., Fernandes, M. M., and Marques, E. (2013). A mersenne twister hardware implementation for the monte carlo localization algorithm. *J. Signal Process. Syst.*, 70(1):75–85.

Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., and Zhang, Z. (2011). High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491.

Gamerman, D. and Lopes, H. (2006). *Markov Chain Monte Carlo: Stochastic Simulation for Bayesian Inference, Second Edition*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis.

Keating, M. and Bricaud, P. (2007). *Reuse Methodology Manual for System-on-a-Chip Designs*. Springer Publishing Company, Incorporated, 3rd edition.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2 (3rd Ed.): Semi-numerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Palestra do WCH
Desenvolvendo aceleradores em FPGAs via PCIe com Bluespec

Prof. Dr. Paulo Matias
Prof. Dr. Ricardo Menotti
Universidade Federal de São Carlos (UFSCar)
matias@ufscar.br
menotti@dc.ufscar.br

Resumo: Hoje há uma proliferação de ferramentas automatizadas para a geração de aceleradores em FPGA. Por um lado, essas ferramentas facilitam a rápida exploração do espaço de projeto, mas por outro promovem o surgimento de uma massa de profissionais que não entendem conceitos básicos de arquitetura necessários à depuração de qualquer problema que ocorra em meio ao fluxo de trabalho. Esta palestra apresentará um exemplo de construção de um acelerador em FPGA utilizando ferramentas de alto nível de abstração, mas que não escondem totalmente os detalhes de arquitetura. Essa abordagem permitirá explicar, na prática, conceitos básicos como entrada e saída mapeada em memória e acesso direto à memória.

Lista de Autores