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

João M. de Queiroz Filho[1], Rogério A. Gonçalves[1] e Alfredo Goldman[2]

[1]Departamento de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão - Paraná - Brasil

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

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

II Workshop em Computação Heterogênea (WCH 2018)

Sumário

Introdução

- O uso de dispositivos aceleradores como `GPUs` em conjunto com `CPUs` *multicore* tem sido bastante amplo.
- Porém o modelo de programação exige que o desenvolvedor declare explicitamente as dimensões do arranjo de *threads* que deve ser criado para a execução de um *kernel*.
- O que muitas vezes levam à utilização de definições genéricas, a escolha de configurações aleatórias ou o uso de um valor padrão.
- O que pode subutilizar os recursos de processamento.

Motivação e Objetivo I

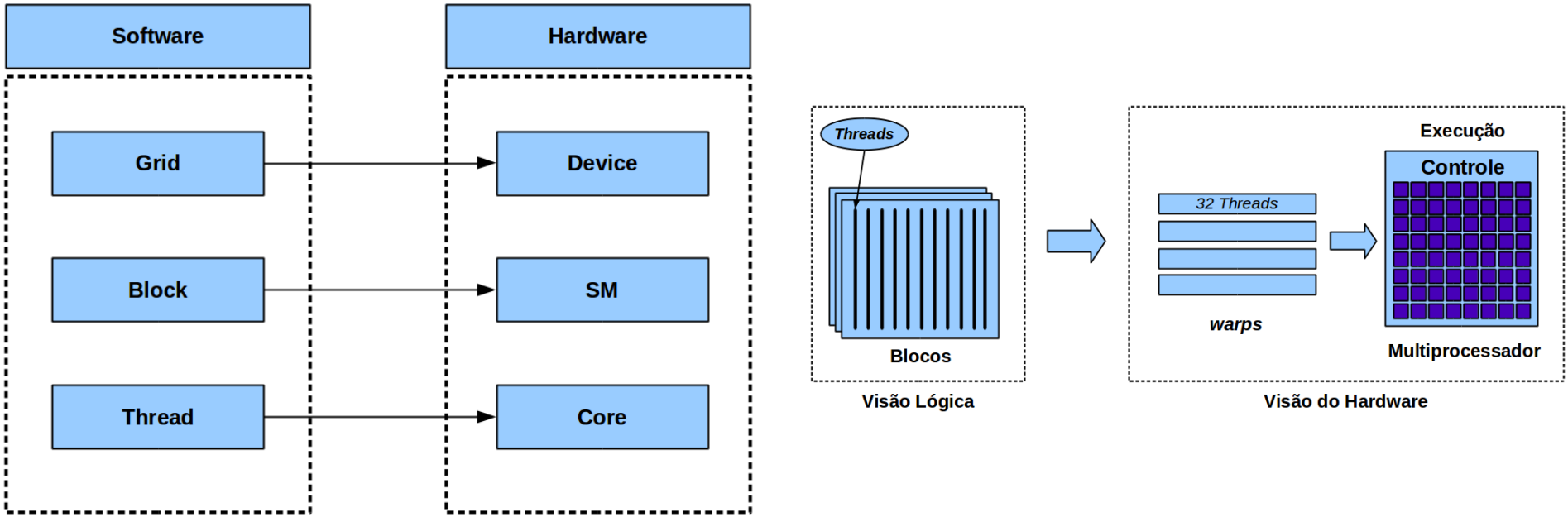
- A proposta deste trabalho está relacionada com o ajuste de código às caracterısticas arquiteturais.
- Busca no conjunto de configurações válidas para *grids*, blocos e *threads*.
- A ideia é ajustar os códigos dos *kernels* dependendo da arquitetura da GPU para a melhoria de desempenho.
- Migrando iterações de laços para os *ids* do arranjo de *threads*

Exemplo de *kernel*

```
1 int main(){
2   /* Declaracao e transferencias de dados foram suprimidas. */
3
4   dim3 grid(32,32,1);
5   dim3 bloco(32,1,1);
6   VecAdd<<<grid, bloco>>>(d_A, d_B, d_C, N);
7
8   return 0;
9 }
```

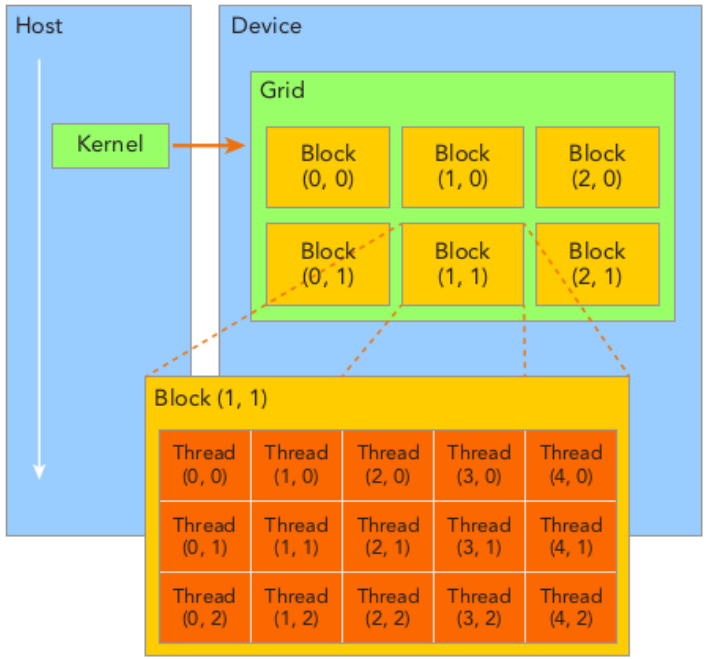
Mapeamento *Software* e *Hardware*

- No lançamento da execução de um *kernel*: `funcaokernel<<grid, block>>(...);`
- Ocorrerá um mapeamento *software* $\rightarrow$ *hardware*



Kernel: *grids* e blocos de *threads*

- Para uma chamada no *kernel* utilizamos *grids* e blocos, sendo definidos como:
  - variáveis do tipo `dim3`
  - grid(x,y,z) e block(x,y,z)
  - config(gx, gy, gz, bx, by, bz).
  - Então para um $N = 1024$, uma das possíveis configurações é $(2, 2, 1)$,$(1, 128, 2)$.
  - Temos um *grid* de 2×2 blocos com 128×2 *threads* executando em cada bloco.



Escolha das Dimensões: Mais ou menos operações

```
1  __device__ int getGlobalIdx_1D_1D() {
2    int threadId = blockIdx.x * blockDim.x + threadIdx.x;
3    return threadId;
4  }
5  ...
6  __device__ int getGlobalIdx_2D_1D() {
7      int blockId = blockIdx.y * gridDim.x + blockIdx.x;
8      int threadId = blockId * blockDim.x + threadIdx.x;
9      return threadId;
10 }
11 ...
12 __device__ int getGlobalIdx_3D_3D() {
13   int blockId = blockIdx.x + blockIdx.y * gridDim.x + gridDim.x
14                 * gridDim.y * blockIdx.z;
15   int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
16             + (threadIdx.z * (blockDim.x * blockDim.y))
17             + (threadIdx.y * blockDim.x) + threadIdx.x;
18   return threadId;
19 }
```

Ajustes no lado do *software*

- Código com 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 }
```

- Versões de *kernels*: com 3, 2, 1 e nenhum laço.

```
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 }
```

Configurações para *grids* e blocos

- Para realizar a escolha das configurações, o desenvolvedor deve respeitar certas condições e restrições determinadas pelo *hardware*.
  1. O total de *threads* criadas em uma configuração será $gx \times gy \times gz \times bx \times by \times bz$.
  2. $(bx \times by \times bz \leq 1024)$, número máximo de *threads* por bloco.
  3. $(bx \times by \times bz)$ tem que ser divisível por 32, as *threads* são organizadas em grupos de 32 em *warps*.
  4. Total de *threads* criadas deve ser menor que o tamanho de N.

GPU GTX 780 - Arquitetura Kepler
CUDA Cores: 2304
***Warp size*:** 32
Máximo número de *threads* por SM: 2048
Máximo número de *threads* por bloco: 1024
Máxima dimensão de *threads* por bloco: (1024, 1024, 64)
Máximo tamanho de *grids*: (2147483647, 65535, 65535)
Memória: 3GB

- *Autotuning:* técnica de otimização de *software*, que tem a função de explorar o espaço de configurações.

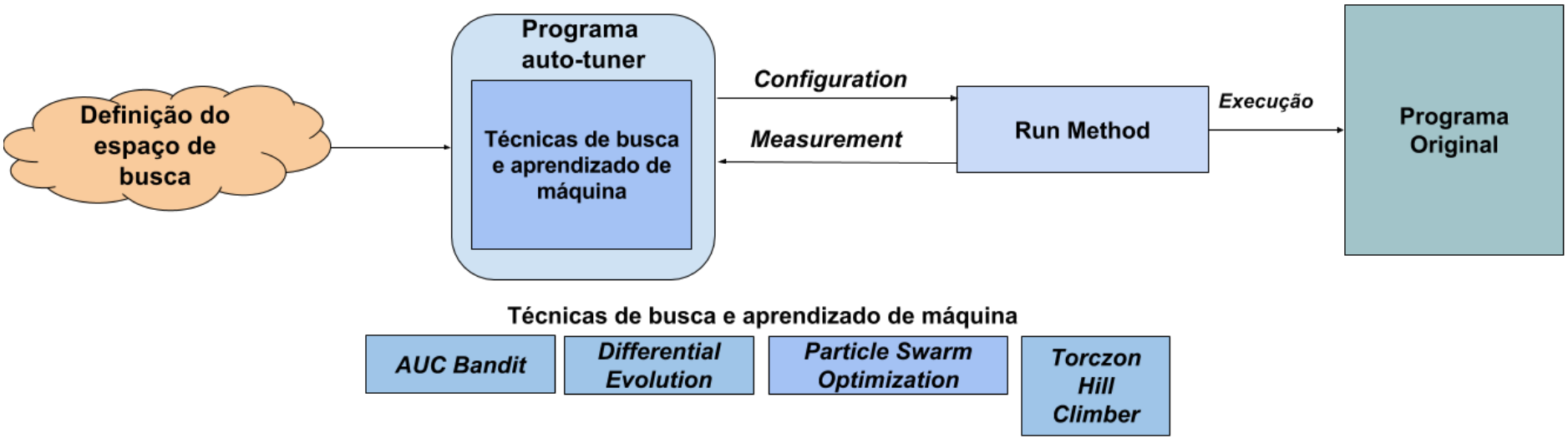


Figura 1: Esquemático do funcionamento do OpenTuner. *Fonte:* Ansel et al. [2014]

Metodologia

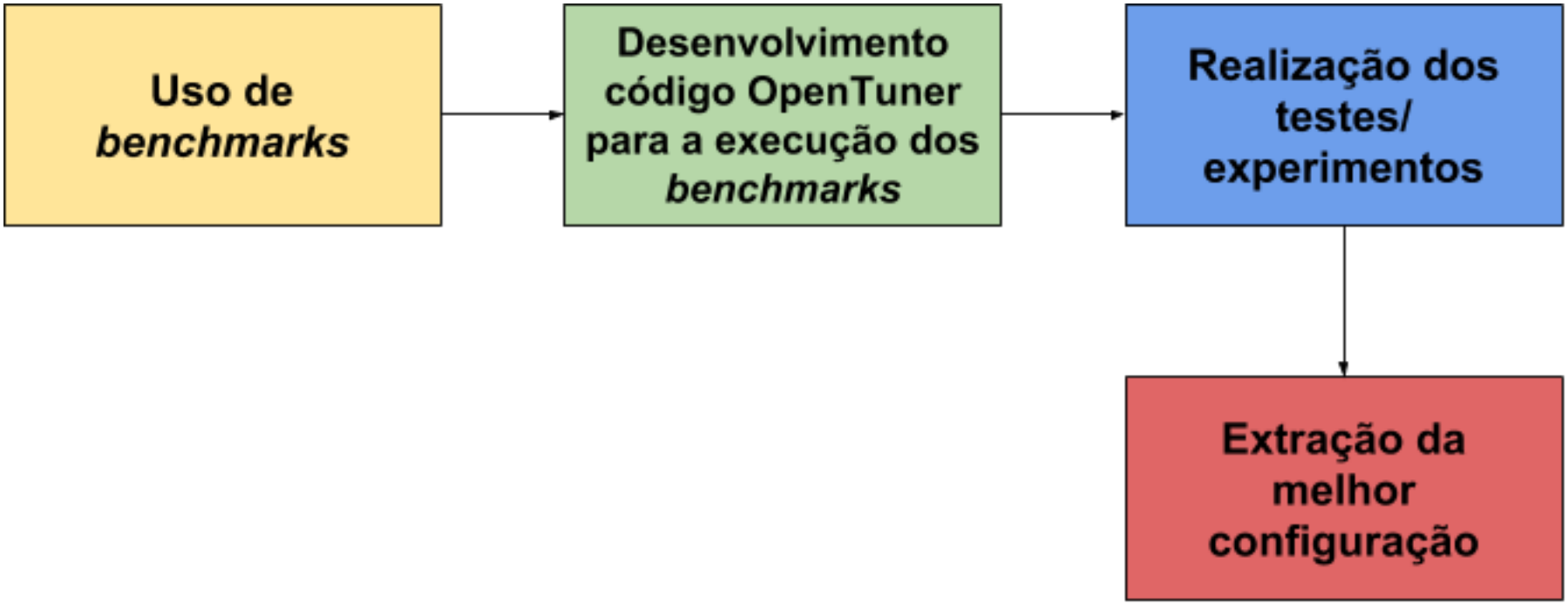


Figura 2: Etapas da realização da pesquisa

- `sm_efficiency`: porcentagem de tempo que pelo menos um *warp* está ativo no multiprocessador.
- `nvprof --query-metrics --query-events`

Experimento *sincos* por força bruta

Tabela 1: Teste por força-bruta métrica *sm_efficiency*

N	TitanX		
	Kernel	Configuração	sm_efficiency
320	0	200,1,8,8,4,2	50.8%
—	—	—	—
320	0	1,20,16,1,16,20	60.81%
320	0	40,1,10,64,4,1	65.88%
—	—	—	—
320	1	320,10,1,1,8,4	99.05%
320	1	50,1,64,4,2,4	99.1%
320	1	8,400,1,1,8,4	99.14%
320	1	32,100,1,1,8,4	99.19%
320	1	80,40,1,1,8,4	99.25%

Experimento sincos pelo OpenTuner

Tabela 2: *Sincos* métrica *sm_efficiency* GPU-GTX-780

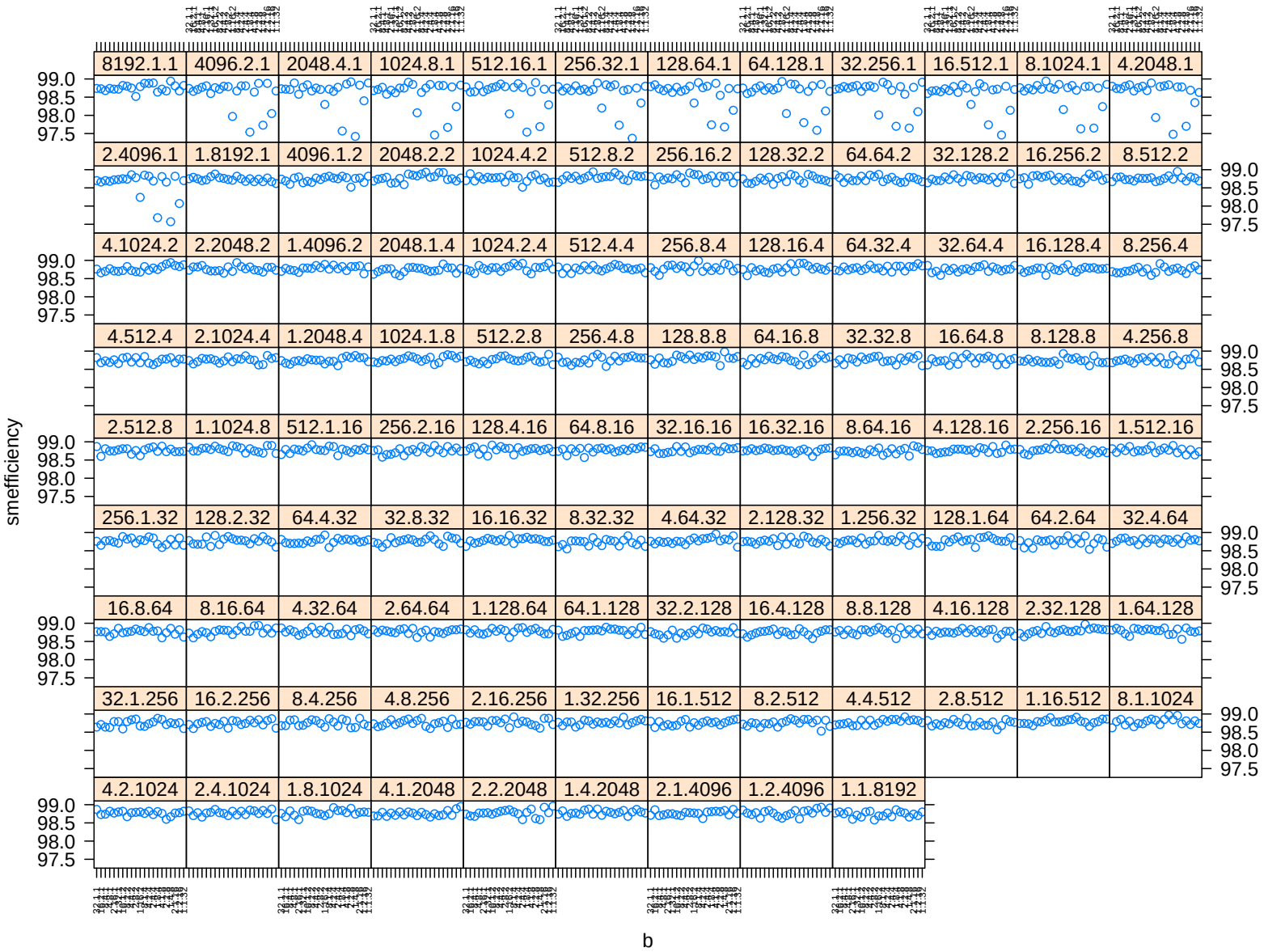
N	Kernel	gx	gy	gz	bx	by	bz	sm_efficiency
64	1	1	16	8	1	8	4	97.44%
96	1	4	3	1	4	32	6	98.9%
128	1	32	16	1	1	4	8	98.4%
160	1	80	10	1	1	32	1	99.49 %
192	1	12	4	1	1	48	16	99.79%
224	1	1	4	392	32	1	1	99.73%
256	1	128	16	1	1	16	2	99.68 %
288	1	72	18	2	2	2	16	99.81%
320	1	5	160	4	16	1	2	99.85%

Experimento sincos pelo OpenTuner

Tabela 3: *Sincos* métrica *sm_efficiency* GPU-TitanX

N	Kernel	gx	gy	gz	bx	by	bz	sm_efficiency
64	1	2	16	4	8	4	1	93.02%
96	1	144	1	2	2	2	8	99.63%
128	1	2	32	4	1	16	4	99.8%
160	1	8	50	1	1	2	32	99.52%
192	1	8	8	9	2	8	4	97.38%
224	1	1	2	784	2	4	4	99.16%
288	1	1296	2	1	1	8	4	99.6%
320	1	2	160	10	4	2	4	99.47%
352	1	1	88	44	16	2	1	99.9%

Detecção de Padrões



Conclusão

- O desenvolvedor não irá usufruir do melhor desempenho da arquitetura fazendo escolhas de dimensões aleatoriamente
- Utilizar do método força-bruta não é a melhor escolha
- Ferramentas de *autotuning* ajudam na busca por uma configuração melhor em um tempo hábil
- Não conseguimos ainda detectar um padrão. :-(

Próximos Passos

- Preparar mais *benchmarks* para testes.
- Análise dos *logs* das escolhas que a ferramenta faz no processo de em busca para a detecção de padrões.

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 I

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014. URL `http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf`.

Perguntas?

Obrigado!

Rogério Aparecido Gonçalves
rogerioag@utfpr.edu.br
João Martins de Queiroz Filho
joaomfilho1995@gmail.com