GPGPU Compiler optimizations

Diogo Nunes Sampaio
sampaio@dcc.ufmg.br

How to optimize the binary code that goes into GPUs?

The increasing programmability of GPUs is attracting many developers. GPUs are a cost and power efficient alternative for massively data parallel computation. At the logic level, they are built around a model called the **S**ingle **I**nstruction, **M**ultiple **T**hreads paradigm. However, at the implementation level, GPU threads are combined in groups that follow the **S**ingle **I**nstruction, **M**ultiple **D**ata paradigm. Given that GPUs are relatively new, there are not many compiler optimizations that target them specifically. Our objective is to help the compiler community to fill up this gap. In particular, our research focus on compilers analyses and optimizations tailored specifically to SIMD architectures.

Figure 1

Nvidia's Geforce 680

- 128 Gflops double precision
- 3090 Gflops single precision
- 1536 cores, 1006MHz
- 2GB GDDR5 mem
- 192 GB/s mem bandwidth
- 195 W
- US $500

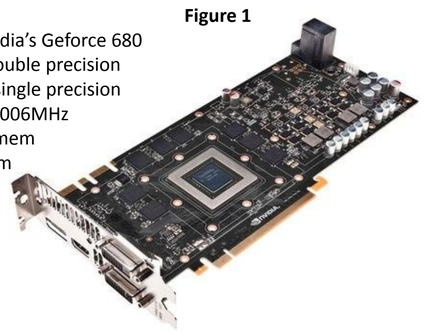

Figure 2

Nvidia's Tesla M2090

- 665 Gflops double precision
- 1331 Gflops single precision
- 512 cores, 1300MHz
- 6GB GDDR5 mem
- 177 GB/s mem bandwidth
- 250W
- US $2760

GPUs

- Were designed for 3D graphics processing and rastering
  - Data parallel computing (vertix, textures, peixels)
- Huge computing growth
  - Parallelism increase
    - Multiple vector processor
  - Clock increase

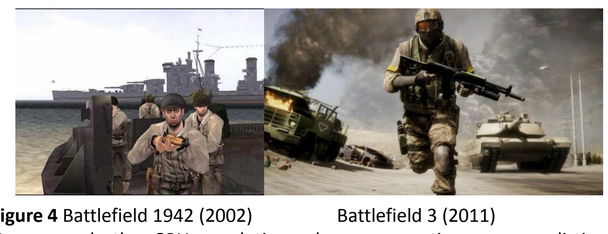

Figure 4 Battlefield 1942 (2002) Battlefield 3 (2011)
Games push the GPUs evolution, always presenting more realistic graphics and better physics simulation. Above two figures of games of the same franchise.

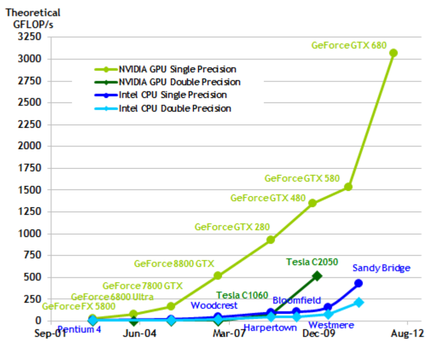

Figure 5
Nvidia's chart comparing GPU and CPU processing capacity X release date

The Single Instruction Multiple Threads execution model

- Up to dozens of thousands virtual threads (grid) created simultaneity
- Virtual threads are grouped in blocks
- Each block executes in only one vector processor (Stream Multiprocessor)
- Each block is executed in subgroups called warps
- SIMT: Special hardware that handles virtual threads execution over vector processors (SIMD)

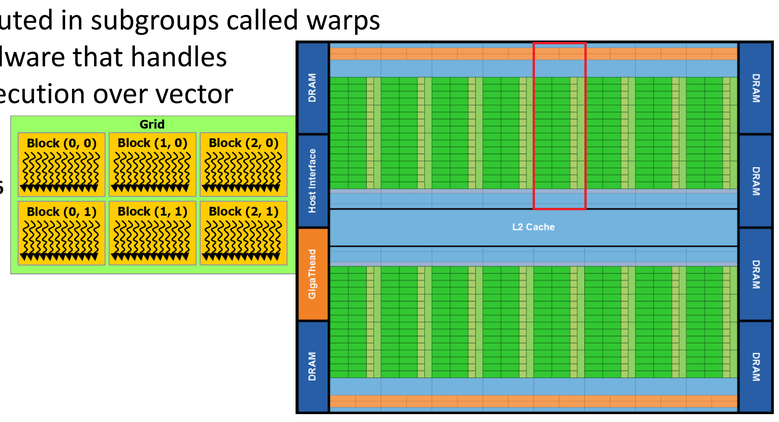

Figure 6
Left: A grid of 6 blocks
Right: GPU with 16 SMs of 32 threads each (warp).
A single SM marked in red

GPU compilers and APIs

- Brand dependent APIs
- C / C++ language extensions
  - OpenCL, CUDA, DirectCompute
- Proprietary compilers:
  - ATI, Nvidia
  - Generate binary code
- Open source compilers:
  - Ocelot
    - ptx ⇒ ptx
    - Intermediate code

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Figure 7 CUDA for C example
Kernels, that are functions that run on the GPU, have a syntax that differs from a conventional function only by the "<<<" and ">>>" tags and that always return void.
On kernels exists special variables used to name each thread:

- threadIdx: Thread index in the block
- blockIdx: Block index on a grid
- blockDim: Blocks count on a grid

Divergence analysis

- Classifies variables based on the values seen by the threads of a warp
  - Constant: All threads see the same value
  - Divergent: Threads see different values
- Used on the open source compiler Ocelot
- Conservative analysis: Tells what variables are definitely constant, and the variables that might diverge, as it may depend on execution input, as Ocelot is limited to never know blocks sizes and shapes

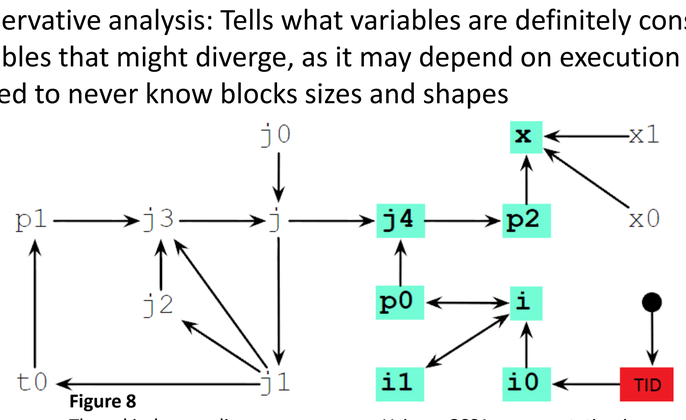

Figure 8
Thread index are divergence sources. Using a GSSA representation is build a graph with data and control dependency between variables. The divergence is computed using graph propagation algorithm.

Divergence analysis with affine constrains

- Uses constant propagation idea
  - Evaluates instructions to describe the result as a equation of the input
- Classifies variables based as an expression based on the thread index: $V = A\ tid + B$
  - Constant: $A = 0,\ B\ is\ known\ at\ compile\ time$
  - Uniform: $A = 0,\ B\ is\ unknown\ at\ compile\ time$
  - Constant affine: $A \neq 0,\ A\ and\ B\ know\ at\ c.t.$
  - Affine: $A \neq 0,\ A\ known, B\ unknown$ at c. t.
  - Divergent: $A\ unknown\ at\ compile\ time$

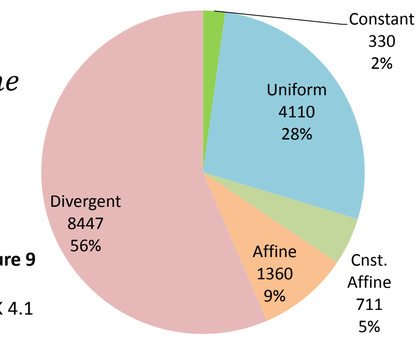

Figure 9
Data distribution between the 5 classes above, in 27 applications taken from Rodinia 2.0.1 and Nvidia Cuda SDK 4.1

Divergence aware register allocator

"Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important—if not the most important—optimizations." John L. Hennessy, David A. Patterson; Computer Architecture: A Quantitative Approach

The on processor memory is divided in two sections, cache and shared memory. Because of the high number of threads that a GPU supports simultaneously, each thread might get very little space in the cache. Therefore, an important issue in GPGPU code generation is how to maximize shared memory usage. Furthermore, missing data in caches is particularly painful in this domain, because retrieving values from off-chip memory is very expensive. We have been working on a register allocator for GPUs:

- Use of divergence analysis to spill variables in different ways
- Based on divergence analysis:
  - Constant: Spill to shared memory, a single memory position by warp
  - Divergent: Spill to local memory, a different memory position per thread
- Based on divergence analysis with affine constrains:
  - Constant: No memory usage, just uses value straight on the variable position
  - Constant affine: No memory usage, rematerializes data
  - Uniform: Spill to shared memory, a single memory position by warp
  - Affine: Spill to shared memory, a single memory position by warp. Does data decomposition on stores and rematerialization on loads. Stores on memory value unknown at compile time

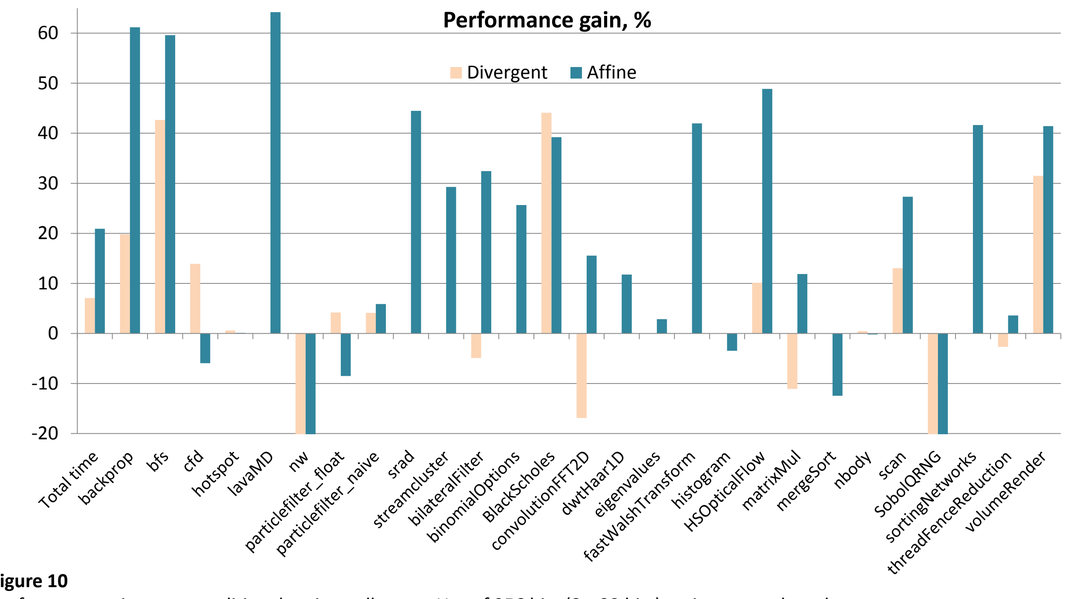

Figure 10
Performance gain over a traditional register allocator. Use of 256 bits (8 x 32 bits) registers per thread. Compiled with modifications over Ocelot svn review 1824.

All register allocators are based on the traditional register allocation, that is a linear scan and a same spill policy.

Backprop, bfs e labaMD ran in less than a half of the time. However nw and SobolQRNG ran respectively 4.7 and 2.3 times slower

Divergence aware register allocators require to maintain the spill position for each thread, so the amount of registers free for usage is lower by the amount of register requires to address a memory position.

Tests over a 32 bit Linux, with a Geforce GTX 470, cuda toolkit 4.1, gcc 4.4.3

Future work

- Spill policy based on divergence type
- Study if is possible to identify variables that require single shared memory position for all threads on a SM instead of single position per warp
- Variables value range ➔ smaller memory positions
- Use of more affective spill register algorithms
- Submit affine analysis and register allocator to Ocelot

Contributions

- Divergence Analysis and Optimizations, PACT 2011, **16% acceptance**
- Divergence Analysis with Affine Constrains
- A Register Allocator for SIMD Machines
- Open source code in Ocelot, a industrial strength compiler

About me

- M.Sc. Student at UFMG
- Main interests: Compilers, GPGPU, computer graphics, physics simulation
- Adviser: Fernando Magno Quintão <dcc.ufmg.br/~fernando/>

UNIVERSIDADE FEDERAL DE MINAS GERAIS · INCIPIT VITA NOVA · 7 DE SETEMBRO DE 1927
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PPGCC PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG
UFMG - ICEx
CAPES 5 anos
FAPEMIG 25 ANOS