Avaliando o Impacto de Mudanças na Arquitetura de Memória entre Gerações de GPGPUs no Desempenho das Otimizações de Computações de Estênceis *

Rodrigo L. Machado [1], **Thiago C. Nasciutti** [1], **Jairo Panetta** [1]

[1]Divisão de Ciência da Computação
Instituto Tecnológico de Aeronáutica (ITA) – São José dos Campos, SP

{rodrigo21lm, thiagonasciutti, jairo.panetta}@gmail.com

Abstract. *This article explore the impact of coding optimization, with focus on memory hierarchy, in two GPGPUs from distinct generations. The results are compared and explained in light of the memory hierarchy variation between generations.*

Resumo. *Este trabalho explora o impacto de otimizações de código, com foco na hierarquia de memória, em duas gerações distintas de GPGPUs. Os resultados são comparados e explicados através das mudanças na constituição da memória ocorridas entre uma geração e outra.*

1. Introdução

Otimizar computações de estênceis é tema de diversas pesquisas como [1], [2], [3] e [4]. O trabalho [4] analisa o impacto de diversas otimizações de memória na velocidade de computações de estênceis na GPGPU[1] NVIDIA K80. As otimizações substituem releituras da memória global por releituras da memória local. Utilizando um experimento padronizado que mede o desempenho de cada otimização em função do tamanho do estêncil e do tamanho do problema, o trabalho conclui que tais otimizações aceleram substancialmente a computação para múltiplos tamanhos do estêncil e do problema, e que a otimização de melhor desempenho varia com os tamanhos do estêncil e do problema.

A NVIDIA K80, utilizada em [4], é uma GPGPU da quarta geração de arquitetura NVIDIA, denominada arquitetura Kepler. As GPGPUs mais recentes são da sexta geração de arquitetura, denominada Pascal. Há diferenças na arquitetura de memória dessas gerações. Este trabalho utiliza a P100, GPGPU de arquitetura Pascal, para verificar se os resultados obtidos no trabalho original [4], na arquitetura Kepler, se mantém na Pascal.

2. Resumo Teórico

Equações diferenciais parciais como a equação do calor podem ser resolvidas por soluções explicitas do sistema de equações lineares advindo do método de diferenças finitas. Em cada instante do tempo, cada ponto do domínio é atualizado por uma combinação linear

*Este trabalho foi parcialmente financiado pelo Termo de Cooperação 0050.0102253.16.9 entre a Petrobras e a Universidade Federal do Rio Grande do Sul

[1]General Purpose Graphics Processing Unit

dos valores em alguns pontos vizinhos no instante anterior. Este tipo de computação é denominado computação de estêncil, pois a mesma computação é aplicada em todos os pontos do domínio. A Figura 1 apresenta os estênceis 3D utilizados neste trabalho. Os tamanhos dos estênceis crescem com a redução do erro de truncamento da aproximação discreta das derivadas espaciais.

Uma GPGPU é constituída por um conjunto de multiprocessadores (*SMX*) e uma memória global. A GPGPU executa funções denominadas *kernel*, enviadas pela CPU. Em cada envio a CPU define quantas threads irão executar o kernel e agrupa essas threads em blocos. Cada bloco é executado por um SMX. A memória de uma GPGPU é composta por uma memória global, acessível à todos os SMX, e uma memória local, exclusiva de cada SMX. A memória local ainda é subdividida em cache L1, cache somente leitura (*Read Only Cache*) e memória compartilhada (*Shared Memory*). Na arquitetura Kepler retratada na Figura 2, a memória cache L1 e a memória compartilhada dividem o mesmo hardware enquanto a cache somente leitura ocupa um espaço diferente. Na arquitetura Pascal essa organização se inverte. A cache L1 e a cache somente leitura dividem o mesmo espaço enquanto a memória compartilhada fica separada.

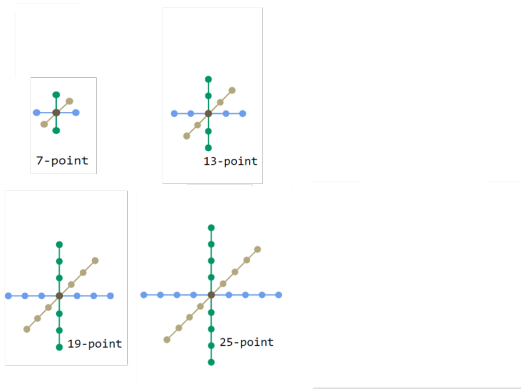

Figura 1. Estênceis de 7 a 25 pontos

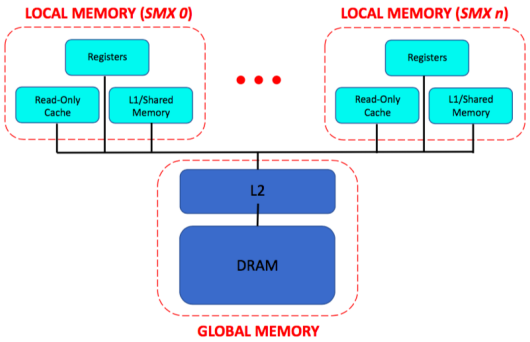

Figura 2. Arquitetura de Memória da K80

3. Otimizações

Assim como no trabalho original, utilizamos três implementações diferentes da equação do calor. A primeira delas chamada BASE, cuja atribuição central do kernel esta apresentada no Código 1, é a implementação usual da solução da equação de calor, sem qualquer otimização. A segunda implementação utiliza a cache somente leitura para otimizar a computação. Para isso ela utiliza chamadas de função *__ldg()* para armazenar os dados nessa memória. A esta implementação chamaremos de CSL e a atribuição central do kernel pode ser vista no Código 2. Observe que esta otimização requer apenas a inclusão de métodos *__ldg()* no código usual. A terceira implementação utiliza a memória compartilhada. Iremos nos referir a ela por COMP e sua implementação pode ser vista no Código 3. Esta memória é reservada utilizando a diretiva *__shared__* . A complexidade adicional desta otimização ocorre pelo usuário ter que armazenar e preencher manualmente a memória reservada, acarretando código extenso e não imediato.

Código 1. BASE

```
b[index] = coeff[0] * a[index] +
  coeff[1] * a[index-1] +
  coeff[2] * a[index+1] +
  coeff[3] * a[index-dimx] +
  coeff[4] * a[index+dimx] +
  coeff[5] * a[index+(dimx*dimy)] +
  coeff[6] * a[index-(dimx*dimy)];
```

Código 2. CSL

```
b[index] = coeff[0] * __ldg(&a[index]) +
  coeff[1] * __ldg(&a[index-1]) +
  coeff[2] * __ldg(&a[index+1]) +
  coeff[3] * __ldg(&a[index-dimx]) +
  coeff[4] * __ldg(&a[index+dimx]) +
  coeff[5] * __ldg(&a[index+(dimx*dimy)]) +
  coeff[6] * __ldg(&a[index-(dimx*dimy)]);
```

Código 3. COMP

```
__shared__ float ds_a[BY+2*R][BX+2*R];
// Load halos and center point
if (threadIdx.y < R) { ds_a[threadIdx.y][tx] = a[index-(R*dimx)];
  ds_a[threadIdx.y + BY + R][tx] = a[index+(BY*dimx)];}
if (threadIdx.x < R) { ds_a[ty][threadIdx.x] = a[index-R];
  ds_a[ty][threadIdx.x + BX + R] = a[index+BX];}
ds_a[ty][tx] = a[index];
__syncthreads();
b[index] = coeff[0] * ds_a[ty][tx] + coeff[1] * ds_a[ty][tx-1] +
  coeff[2] * ds_a[ty][tx+1] + coeff[3] * ds_a[ty-1][tx] +
  coeff[4] * ds_a[ty+1][tx] + coeff[5] * a[index-stride] +
  coeff[6] * a[index+stride];
```

4. Experimento

Reproduzimos o experimento padronizado de [4], variando o tamanho do domínio e o raio do estêncil. O domínio é um array 3D de ponto flutuante em precisão simples com dimensões $(x, 256, 256)$ com x variando de 32 à 2048 em passos de 32. Realizamos os experimentos para os estênceis com 7, 13, 19 e 24 pontos da Figura 1, fixando o número de threads por bloco em $(32, 1, 1)$.

Representando a arquitetura Kepler utilizamos a placa K80, composta por duas placas K40 independentes. A arquitetura Pascal foi representada por uma placa P100. Nas duas máquinas utilizamos o compilador *nvcc v9* com chaves de compilação *-m64 -Xcompiler -Wall -Xptxas -O3*. Como os tempos de execução foram medidos diretamente nas GPGPUs, as CPUs e os sistemas operacionais utilizados são irrelevantes.

5. Resultados e análise

A Figura 3 apresenta os desempenhos (GFlop/s) em função dos valores de x para cada estêncil nas duas máquinas. Observando que os desempenhos máximos são aproximadamente 130, 180, 210 e 220 GFlop/s na K80 e 480, 840, 1020 e 1200 GFlop/s na P100, concluimos que a Pascal é de 3,6 à 5,4 vezes mais rápida que a Kepler nesse problema.

Observamos que o desempenho de BASE parece estar ausente dos gráficos na Pascal. Ocorre que o desempenho de BASE e de CSL são idênticos na Pascal. Já na Kepler, o desempenho de CSL é muito superior ao de BASE. Isto deve-se à mudança na hierarquia de memória. Na Pascal a cache somente leitura se encontra junto da cache L1, diferentemente da Kepler. Por causa disso, na Pascal acessos à memória global podem ser armazenados no cache somente leitura, reproduzindo o efeito de inserir *__ldg()* em BASE na Kepler (vide [5]). Por isso o aumento de desempenho de BASE para CSL na Kepler desaparece na Pascal.

O terceiro ponto importante é que a otimização de melhor desempenho mudou. Na Kepler era majoritariamente a CSL, enquanto na Pascal é majoritariamente a COMP.

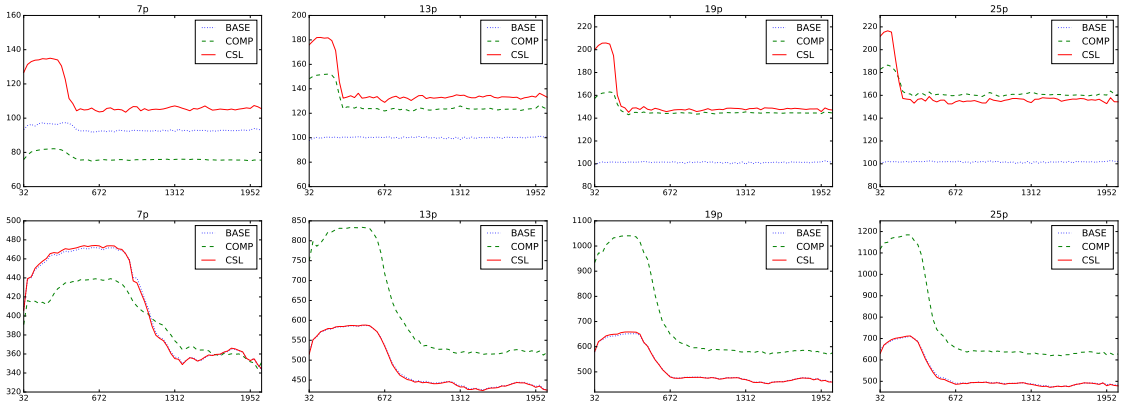

Figura 3. Desempenho em GFlop/s na K80 (linha superior) e na P100 (linha inferior) em função de x para estênceis de 7 a 25 pontos

Atribuímos isto a dois fatos. Primeiro, a memória compartilhada é maior na Pascal que na Kepler, como atestado em [5]. Segundo, os acessos no eixo z em COMP podem ser automaticamente colocados no cache somente leitura na Pascal mas não na Kepler, reduzindo ainda mais o número de acessos à memória global.

6. Conclusão e Trabalhos Futuros

Substituir releituras da memória global por releituras da memória local aumenta o desempenho em ambas as placas. Em particular para a Pascal isto ocorre somente no uso da memória compartilhada devido às mudanças na arquitetura de memória entre uma geração e outra. Todavia, devido à estas mudanças a memória compartilhada ficou maior e esta otimização ganhou desempenho.

Como trabalhos futuros pretendemos utilizar profilers para averiguar algumas hipóteses citadas. Por exemplo, gostaríamos de medir o impacto do aumento da memória somente leitura medindo a taxa de cache hit em ambas as placas. Além disso também pretendemos explorar as outras otimizações retratadas em [4] como a internalização do laço em z e o reuso de registradores.

Referências

[1] Andreolli C, Thierry P, Borges L, Skinner G and Yount C (2015), *Characterization and Optimization Methodology Applied to Stencil Computations* in Reinders J and Jeffers J (eds.) High Performance Parallelism Pearls, Morgan Kaufmann.

[2] Micikevicius, P (2009), *3d finite difference computation on gpus using cuda* in Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, ACM.

[3] Serpa MS, Cruz EH, Diener M, Krause AM, Farres A, Rosas C, Panetta J, Hanzich M and Navaux PO (2017), *Strategies to improve the performance of a geophysics model for different manycore systems*, WAMCA.

[4] Thiago C. Nasciutti, Jairo Panetta (2016), *Impacto da Arquitetura de Memória de GPGPUs na Velocidade da Computação de Estênceis*, WSCAD.

[5] NVIDIA, *Tuning CUDA applications for Pascal v9.1.85*, NVIDIA online documentation.