Impacto da Arquitetura de Memória de GPGPUs na Velocidade da Computação de Estênceis

Thiago C. Nasciutti[1], **Jairo Panetta**[1]

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

{thiagonasciutti,jairo.panetta}@gmail.com

Abstract. *This paper presents a memory hierarchy based performance analysis for 3D stencil computation in GPGPUs (General Purpose Graphics Processing Units). The work evaluates codes that explore shared memory, read only cache, inserting the Z loop into the kernel and register reuse. Each code is benchmarked for several stencil sizes and input domain sizes to evaluate their influence on performance. A detailed study shows the L2 cache influence on the performance and points that the preferred code is the combination of read only cache reuse, inserting the Z loop into the kernel and register reuse.*

Resumo. *Este trabalho apresenta análise de desempenho da computação de estênceis 3D em GPGPUs (Unidades de Processamento Gráfico de Propósito Geral) com foco no uso adequado da hierarquia de memória. São avaliadas codificações que exploram a memória compartilhada, o cache somente leitura, a internalização do laço em Z e o reuso de registradores. Cada codificação é experimentada em diversos tamanhos de estênceis e de domínio de entrada, permitindo observar a influência destes no desempenho final. Conclui-se que em algumas codificações o tamanho do cache L2 afeta o desempenho e que a codificação mais indicada é baseada na combinação do uso do cache somente leitura, internalização do laço em Z e reuso de registradores.*

1. Introdução

A computação de estênceis é largamente utilizada em aplicações científicas que resolvem equações diferenciais parciais por diferenças finitas sobre grades multi-dimensionais. A principal característica dessa computação é que o valor de um ponto da grade na próxima iteração depende apenas dos valores desse ponto e de seus vizinhos na iteração atual. Embora possua paralelismo abundante, o desempenho dessa computação é severamente limitado pela velocidade de acesso à memória (vide Seção 4).

Duas otimizações comumente utilizadas ([Datta et al. 2008] e [Datta et al. 2009]) substituem acessos à memória lenta por acessos à memória mais rápida, porém menor: particionar a grade em telhas (*tiling*) e desenrolar laços (*unrolling*). Além do grande esforço de codificar tais otimizações, a melhor forma de aplicá-las requer extensa experimentação, em particular nas GPGPUs devido à variada arquitetura de memória - caches L1 e L2, memória compartilhada (*shared memory*), cache somente leitura (*read only cache*) e memória central - além da localização dos laços (em cada *thread* ou em blocos de *threads*). Há pesquisas para minimizar esse esforço, expressando a

computação em linguagens específicas para estênceis (*domain specific languages*) como [Tang et al. 2011] que delegam a escolha das otimizações para o compilador. Obviamente, para que o compilador gere otimizações adequadas, é necessário conhecer o resultado da experimentação.

Este trabalho avalia o custo/benefício de nove codificações da computação de estênceis oriundas da discretização de Laplaciano tri-dimensional em GPGPUs, variando o tamanho do estêncil (ordem do erro de truncamento do Laplaciano) e o tamanho da grade. Admitimos que a equação diferencial parcial iguala a variação espacial de grandeza física (o Laplaciano) à variação temporal da mesma grandeza, e que a variação temporal requer os valores da grandeza física em dois passos no tempo. Admitimos codificações com o laço temporal externo ao aninhamento dos laços espaciais. As codificações contemplam o reuso (ou não) de memórias rápidas, variações no posicionamento dos laços espaciais e o reuso (ou não) de registradores. Os experimentos utilizaram a NVIDIA Tesla K80, de arquitetura Kepler [NVIDIA 2012], apenas em precisão simples. As contribuições deste trabalho, restritas a esse espaço de busca, são:

- O cache L2 tem influência substancial no desempenho das codificações que apenas reusam as memórias rápidas;
- Codificações que reusam as memórias rápidas e que movem o laço mais interno para o interior das *threads* reduzem a influência do cache L2;
- A influência do cache L2 praticamente desaparece pelo reuso dos registradores nas codificações que reusam as memórias rápidas e movem o laço mais interno para o interior das *threads*;
- Codificações que reusam a memória cache somente leitura tem desempenho superior e menor custo de codificação do que as que reusam memória compartilhada para todos os tamanhos da grade e para a maioria dos tamanhos dos estênceis.

Este trabalho apresenta a nomenclatura utilizada e os trabalhos relacionados na seção 2, descreve a arquitetura da máquina e o ambiente de software utilizados na seção 3, relata dois conjuntos de codificações e três experimentos nas seções 4 a 8, mostra a ocupação da GPGPU na seção 9 e conclui, indicando trabalhos futuros, na seção 10.

2. Fundamentação Teórica e Trabalhos Relacionados

O número de pontos de um estêncil que representa a discretização de um Laplaciano 3D depende da ordem do erro de truncamento da série de Taylor. Definimos o *raio* do estêncil como o número de pontos do estêncil em cada uma das seis direções a partir do ponto central. Assim, um estêncil com um ponto em cada direção possui 7 pontos e raio 1. Este trabalho avalia estênceis com raios 1, 2, 3, 4 e 5 (7, 13, 19, 25 e 31 pontos respectivamente) permitindo observar os efeitos do tamanho do estêncil no desempenho.

O trabalho pioneiro na otimização de computações de estênceis 3D em GPGPUs é [Micikevicius 2009]. Esse trabalho mostra que manter uma telha 2D na memória compartilhada em conjunto com o reuso dos registradores na dimensão espacial restante acelera substancialmente as computações de estênceis. Os experimentos contemplam estênceis de quatro raios distintos sobre cinco grades de tamanhos distintos. Embora os resultados sejam restritos às primeiras GPGPUs (Tesla S1060/1070), o trabalho é extremamente influente, sendo referenciado por [Nguyen et al. 2010], [Bauer et al. 2011], [Schäfer and Fey 2011],

[Krotkiewski and Dabrowski 2013], [Maruyama and Aoki 2014], [Mei and Chu 2015] e [Hu et al. 2015], dentre outros. Denotamos esta otimização por *pioneira*.

O trabalho de [Bauer et al. 2011] propõe dedicar algumas *warps* para o tráfego de dados entre a memória central e a memória compartilhada enquanto as demais *warps* realizam a computação por estênceis. Segundo os autores, essa otimização acelera de 10% a 15% a otimização pioneira em um estêncil 3D sobre três grades de tamanhos distintos. Denotamos essa otimização por *especialização de warps*.

Denominamos *blocagem temporal* à otimização que antecipa as iterações do laço temporal na telha carregada na memória compartilhada. Essa memória contém tanto a telha da iteração temporal atual quanto a telha da próxima iteração temporal, que é reutilizada para computar a telha na iteração posterior e assim por diante. Essa otimização, aplicada em GPGPUs por [Schäfer and Fey 2011], possui codificação extremamente trabalhosa. Os autores realizam experimentos em estêncil 3D de raio 1 sobre oito grades de tamanhos distintos e concluem que a blocagem temporal é mais eficiente que a otimização pioneira.

Contrastar o desempenho de diversas otimizações em duas arquiteturas de GPGPUs (Fermi e Kepler) é a proposta de [Maruyama and Aoki 2014]. Esse trabalho compara o desempenho da otimização pioneira com o desempenho da especialização de *warps*, da substituição da memória compartilhada pelo cache somente leitura, da blocagem temporal e de combinações dessas otimizações. Realizando experimentos em estêncil 3D de raio 1 sobre grades de tamanhos ótimos para cada otimização, conclui que a blocagem temporal possui o melhor desempenho.

Em contraste com os trabalhos citados, realizamos experimentos sobre estênceis de raios variados em grande volume de grades, restringindo as otimizações ao reuso dos planos 2D carregados na memória compartilhada e no cache somente leitura, incluindo (ou não) o laço na terceira dimensão nas *threads* e reutilizando (ou não) os registradores. Neste estudo detalhado, observamos o impacto do cache L2 e contrastamos o custo/desempenho das otimizações.

3. Ambiente de Execução

Os experimentos utilizaram uma placa NVIDIA Tesla K80, com *driver* versão 367.36 e compilador `nvcc` versão 8.0.26, com chaves de compilação `O3` e `arch sm_37`. Os processadores e o ambiente de software da placa são irrelevantes pois os tempos de execução foram medidos apenas na GPGPU e o tempo para trafegar dados entre a CPU e a GPGPU, além de externo ao trecho medido, é invariável com a codificação.

A NVIDIA Tesla K80 é composta por duas NVIDIA Tesla K40 independentes e ligeiramente modificadas. Este trabalho usa apenas uma das duas Tesla K40. Cada Tesla K40 contém 13 multiprocessadores ("SMX"). Todos os multiprocessadores compartilham uma única memória global (11,25GiB com ECC ligada) externa ao *chip* e um único cache L2 (1,5MB) interno ao *chip*. Cada multiprocessador contém memória cache somente leitura além de outra memória rápida que simultaneamente implementa cache L1 e memória compartilhada. Ao disparar o *kernel* é possível selecionar o tamanho do cache L1 e da memória compartilhada entre 16 e 48KB por bloco com um ou dois blocos por multiprocessador. A Figura 1 [NVIDIA 2012] ilustra a hierarquia de memória.

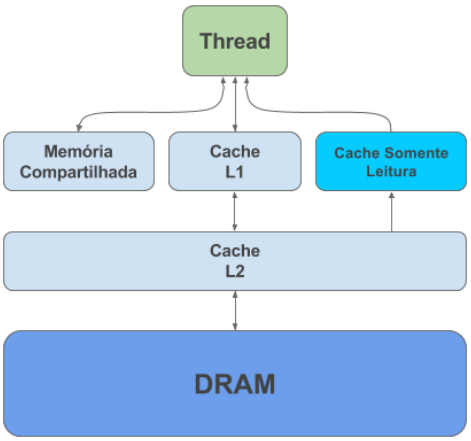

Figura 1. Hierarquia de memória para nVidia Tesla K80

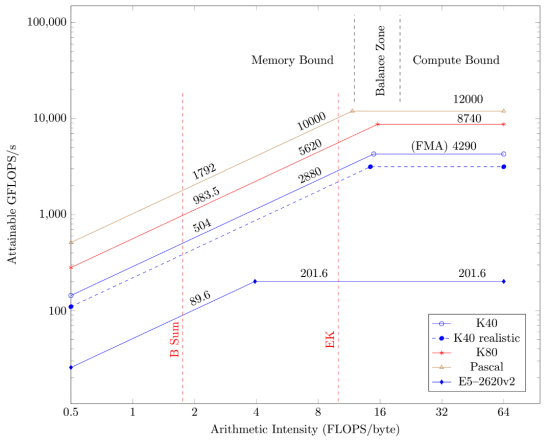

Figura 2. Roofline para nVidia Tesla K80

4. Primeiro Conjunto de Codificações

A *intensidade computacional* de um código é a razão entre a quantidade de operações de ponto flutuante e a quantidade de *bytes* lidos/escritos na memória lenta. O código básico de um estêncil de N pontos, descrito na seção 4.1, requer N operações de multiplicação, $N-1$ operações de adição, uma operação de escrita e N operações de leitura. Logo, a intensidade computacional desse código em precisão simples (4 *bytes* por *float*) é $I(N) = \frac{2N-1}{4(N+1)}$, que tende a 0,5 Flops/byte com o aumento de N.

O modelo Roofline [Williams et al. 2009] relaciona o desempenho máximo de um código à sua intensidade computacional considerando as velocidades de acesso à memória e de processamento de uma máquina. A Figura 2 [Perkins et al. 2015] mostra o modelo Roofline para a GPGPU Tesla K80. A figura aponta que o desempenho do código básico nessa máquina, com intensidade computacional de 0,5 Flops/byte, é limitado pela velocidade de acesso à memória, justificando a afirmação feita na introdução deste trabalho.

Ocorre que cada ponto de um estêncil de N pontos é lido em N posições do ponto central do estêncil na grade, das quais $\lceil \frac{2N}{3} \rceil$ leituras ocorrem quando esse ponto percorre o plano XY e as demais $\lfloor \frac{N}{3} \rfloor$ leituras ocorrem quando esse ponto percorre o eixo Z. Se fosse possível manter todo o plano XY em algum nível da memória rápida, a intensidade computacional seria substancialmente aumentada. Esse é o objetivo das codificações descritas nas seções 4.2 e 4.3.

4.1. Básico (BASE)

O código 1 é o trecho central do *kernel* CUDA da primeira codificacão, que denominamos *Básico (BASE)*, para estêncil de 7 pontos. Os laços espaciais são divididos entre as *threads* e os blocos de *threads*. O *kernel* é invocado no interior do laço temporal. A grade tri-dimensional de dimensões *(dimx, dimy, dimz)* é armazenada nos vetores *a* e *b* na memória global. A variável global *coeff*, armazenada na memória de constantes da GPGPU, contém os coeficientes do estêncil. Este código delega o reuso da memória rápida ao compilador e à arquitetura de memória, limitando a intensidade computacional à 0,5 Flops/byte.

Código 1. BASE

```
row = blockIdx.y * blockDim.y + threadIdx.y;
col = blockIdx.x * blockDim.x + threadIdx.x;
depth = blockIdx.z * blockDim.z + threadIdx.z;
index = (depth+1) * dimx * dimy +
  (row+1) * dimx + (col+1);
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

```
row = blockIdx.y * blockDim.y + threadIdx.y;
col = blockIdx.x * blockDim.x + threadIdx.x;
depth = blockIdx.z * blockDim.z + threadIdx.z;
index = (depth+1) * dimx * dimy +
  (row+1) * dimx + (col+1);
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)]);
```

4.2. Memória Compartilhada (COMP)

A memória compartilhada é significativamente mais rápida que a memória global mas seu tamanho é insuficiente para armazenar toda a grade. Esta otimização particiona a grade em telhas 2D (plano XY), carrega uma telha na memória compartilhada e computa os estênceis que compartilham a telha. O tamanho em *bytes* de uma telha é $M = 4(d_x + 2r)(d_y + 2r)$, onde d_x e d_y são as dimensões X e Y da telha e r é o raio do estêncil. Como este trabalho utiliza $d_X = 32$, $d_y = 16$ e estênceis de raios $r \leq 5$, a memória máxima de uma telha é 4,26 KB, inferior aos 48KB da memória compartilhada. Esta otimização lê da memória central os pontos da grade na direção Z, restringindo o reuso.

O código 3, denominado *Compartilhado (COMP)*, retrata esta otimização. Há grande esforço de codificação, pois o programador controla a memória compartilhada. O aumento na quantidade de linhas de código com relação ao *BASE* advém da carga das bordas da telha e do sincronismo para garantir que a telha XY esteja totalmente carregada antes de seu uso. Os laços espaciais e temporais se mantém na posição do *BASE*.

Código 3. COMP

```
__shared__ float ds_a[BY+2*R][BX+2*R];
tx = threadIdx.x + R; ty = threadIdx.y + R; tz = threadIdx.z + R;
row = blockIdx.y * blockDim.y + ty; col = blockIdx.x * blockDim.x + tx;
depth = blockIdx.z * blockDim.z + tz;
index = (depth) * dimx * dimy + (row) * dimx + (col);
stride = dimx * dimy;

// 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];
```

4.3. Cache Somente Leitura (CSL)

Em arquiteturas anteriores à Kepler existia uma memória interna ao *chip* dedicada a texturas. Sua utilização requeria comandos específicos que dificultavam sua aplicação. Na arquitetura Kepler essa memória não é exclusiva para operações em texturas, podendo ser referenciada por um ponteiro e é denominada cache somente leitura. Seu acesso é otimizado para aplicações com alta localidade espacial onde as *threads* irão buscar dados não necessariamente consecutivos na memória mas próximos uns ao outros em uma região multidimensional, tornando-a bastante adequada para computação de estênceis.

O acesso a essa memória é feito inserindo o qualificador *__ldg()* em cada leitura. Ao contrário do que ocorre com a memória compartilhada, todo o controle dos dados no cache somente leitura é feito pelo compilador e pela arquitetura de memória, simplificando muito sua codificação. Esta otimização está retratada no código 2, denominado *Cache Somente Leitura (CSL)*, e possui código muito próximo ao do *BASE*.

5. Primeiro Experimento

O primeiro experimento mede o desempenho das três codificações nos cinco estênceis sobre grade 256x256x256. Cada *thread* computa um único ponto da grade. Cada bloco de *threads* de tamanho 32x16x1 (na ordem X, Y, Z) computa uma telha. Blocos de *threads* particionam a grade em telhas. A Figura 3 apresenta os desempenhos (GFlops).

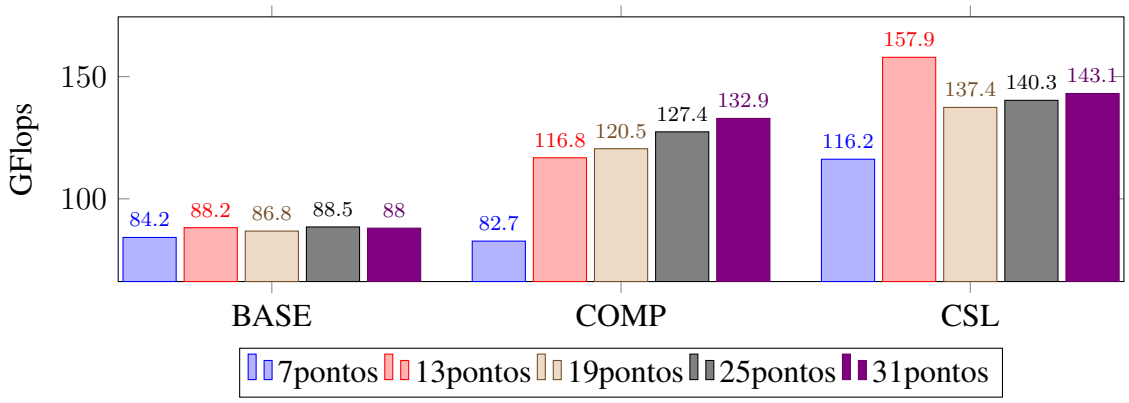

Figura 3. Desempenho medido para grade 256x256x256

O código BASE apresenta desempenho independente do tamanho do estêncil, indicando gargalo comum a todos os tamanhos. O código COMP possui desempenho superior ao BASE exceto no estêncil com 7 pontos. O código CSL é mais rápido que o COMP para todos os tamanhos de estênceis. O aumento de velocidade deve-se à arquitetura voltada a acessos com alta localidade espacial e à não ocorrência explícita de sincronismo e de divergência de *threads*. Como o esforço de implementação do código CSL é significativamente inferior ao do código COMP e o seu desempenho é superior, fica evidente que o código CSL é a melhor opção dentre as testadas para esta grade.

6. Segundo Experimento

Um fato intrigante na Figura 3 é a variação heterogênea do desempenho com o aumento do raio do estêncil em cada código. O segundo experimento investiga essa variação para

diversos tamanhos da grade. Mantendo as dimensões Y e Z fixas em 256 e variando a dimensão X entre 32 e 2048, com passo de 32, mediu-se o desempenho das codificações para cada um dos tamanhos de estêncil propostos. Os resultados apresentados na Figura 4 mostram que o desempenho cai abruptamente em valores de X que variam com o estêncil. O desempenho para X=256 encontra-se antes, durante ou depois dessa queda conforme o raio e o código, justificando a variação heterogênea da Figura 3.

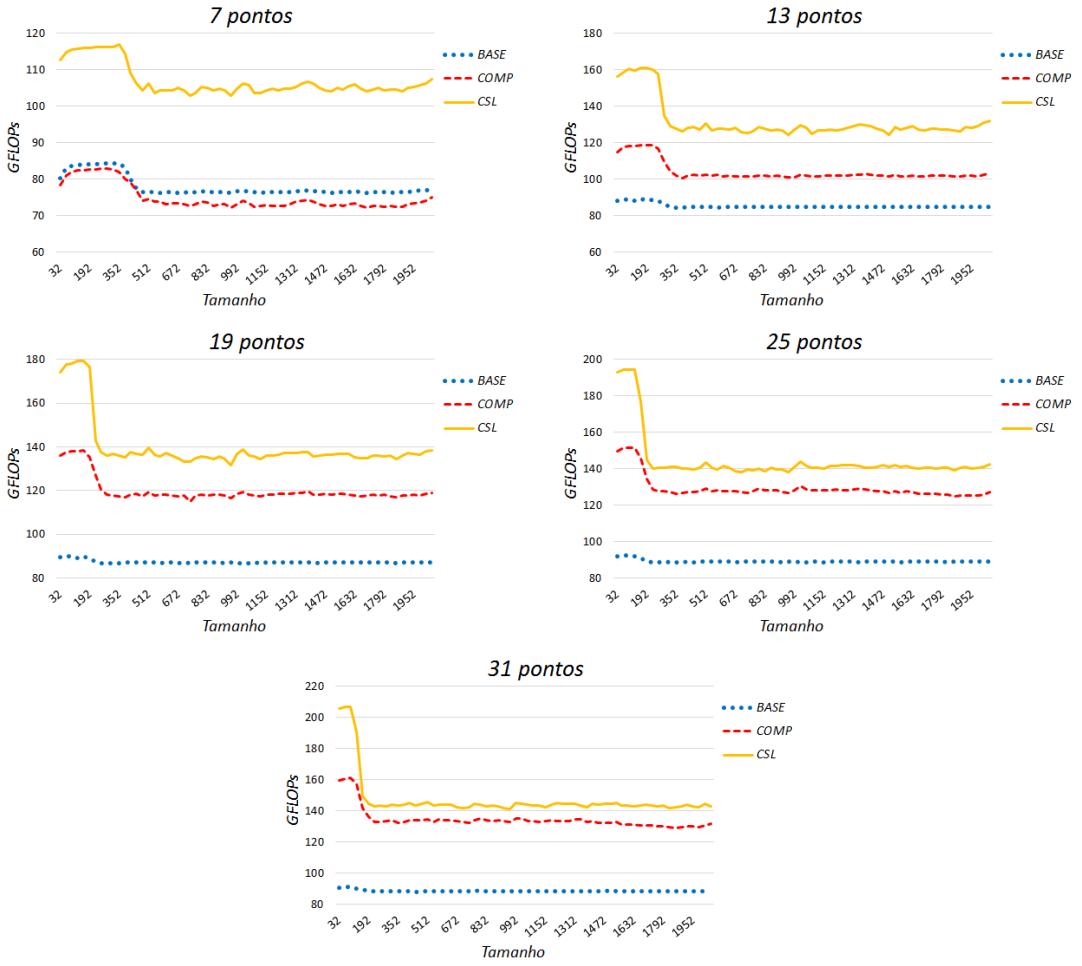

Figura 4. Desempenho das codificações BASE, COMP e CSL ao variar o tamanho da dimensão X da grade

A ferramenta de profiling `nvprof` da NVIDIA foi utilizada para investigar as causas da queda de desempenho. Foram selecionados três valores de X que representam o desempenho antes, durante e após sua queda. O codigo BASE foi executado com o estêncil de 7 pontos e X=320, 416 e 512. O código COMP foi executado com o estêncil de 19 pontos e X=160, 224 e 320. O código CSL foi executado com o estêncil de 31 pontos e X=96, 128 e 224. A Figura 5 apresenta a porcentagem de ”L2 cache hit” nessas execuções. Como todos os dados que são lidos ou escritos na memória global necessariamente passam pelo cache L2, o índice de cache hit está diretamente relacionado ao desempenho da computação.

Da Figura 5 pode-se concluir que o índice de cache hit no cache L2 diminui com o aumento da grade, causando a degradação de desempenho observada na Figura 4.

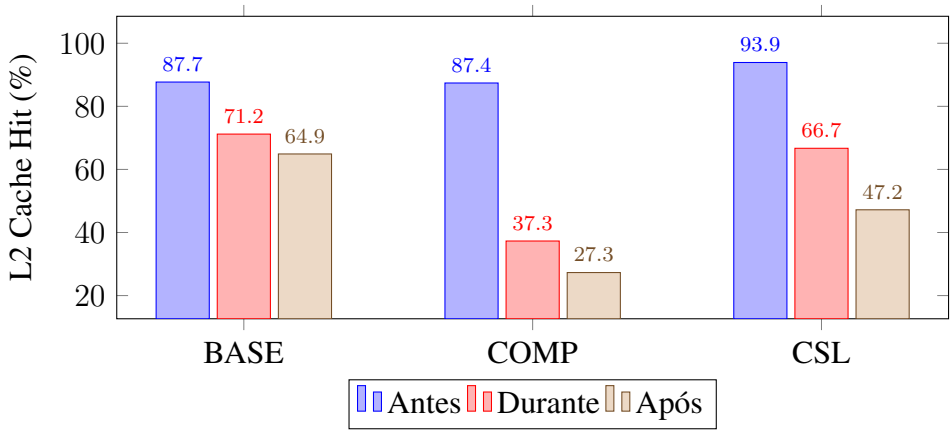

Figura 5. Índices de cache hit para cache L2. Antes, Durante e Após se referem a queda de desempenho observada de acordo com o tamanho de domínio de entrada escolhido

Para computar todos os estênceis com ponto central em um plano XY da grade é preciso carregar da memória todos os pontos do plano além dos pontos vizinhos nas três dimensões conforme ilustrado pela Figura 6.

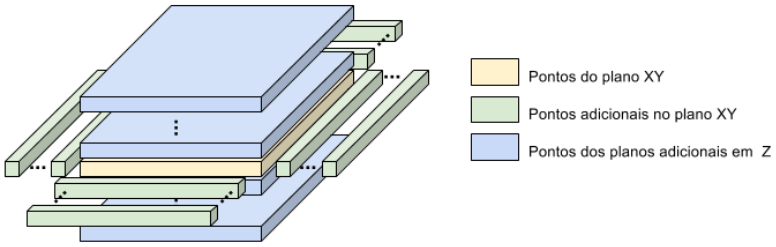

Figura 6. Pontos necessários para computar o estêncil em três dimensões de um plano XY

A Figura 6 auxilia o cálculo da memória necessária (em *bytes*) para computar o estêncil em todos os pontos do plano XY por $M = 4((D_x D_y)+2r(D_x+D_y)+2r(D_x D_y))$ onde D_x e D_y correspondem às dimensões X e Y da grade e r representa o raio do estêncil. O primeiro termo da soma corresponde aos pontos do plano XY, o segundo termo corresponde aos pontos da borda do plano XY e o terceiro termo corresponde aos pontos dos planos adicionais em Z. Para que estênceis com $D_y = 256$ caibam integralmente no cache L2 ($M = 1,5$MB), os valores máximos de D_x para raios de 1 a 5 são 510, 306, 218, 169 e 139, respectivamente. Observando a Figura 4 nota-se que esse valores coincidem com a queda de desempenho. Conclui-se que a queda de desempenho deve-se ao esgotamento do cache L2, o que havia sido evidenciado pela redução no índice de cache hit. Em suma, computar totalmente um plano XY esgota a capacidade do cache L2.

7. Segundo Conjunto de codificações

A Figura 6 também retrata os trechos da grade necessários para a computação de todos os pontos de um bloco de *threads*, bastando substituir "plano" por "bloco". Ocorre que dois blocos de *threads* com início no mesmo par (X,Y) e Z consecutivos compartilham boa

parte dos pontos. Consequentemente, cada setor do plano XY contido em um bloco de *threads* é lido da memória central diversas vezes. Aumentar o reuso desse setor do plano é a motivação do segundo conjunto de codificações.

Para aumentar o reuso dos setores do plano por blocos de *threads* com os mesmos pares (X,Y) e com Z consecutivos é necessário executar esses blocos em sequencia em um mesmo multiprocessador. Ocorre que a GPGPU não permite atribuir um bloco de *threads* a determinado multiprocessador. Uma forma de garantir o reuso é mover o laço em Z para o interior de cada *thread*. Denominamos esta técnica de *internalizar o laço em Z (INTZ)*, que foi implementada nas três codificações originais, gerando três novas codificações.

8. Terceiro Experimento

Os experimentos que medem o reuso do cache L2 reportados na Figura 5 foram refeitos para as três novas codificações, com resultados reportados na Figura 7. O contraste entre as duas figuras demonstra que internalizar o laço em Z eliminou a grande variação no reuso do cache L2.

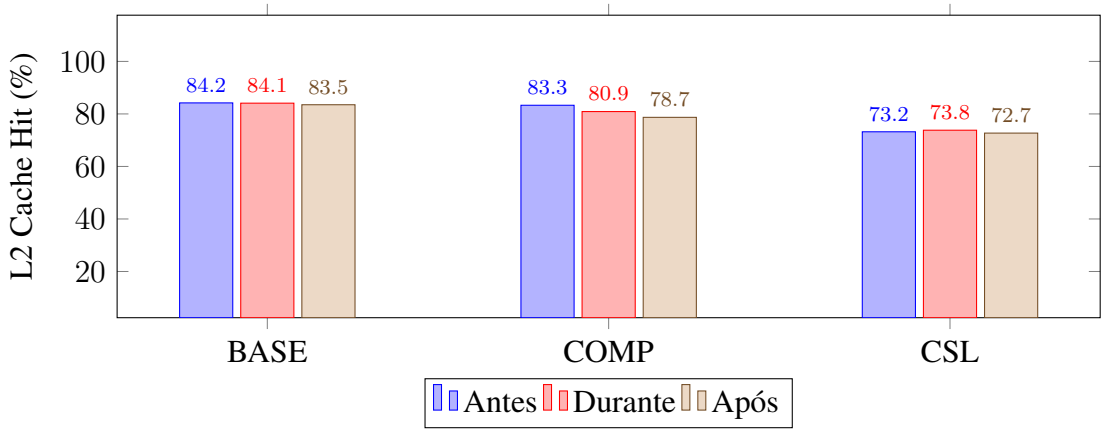

Figura 7. Índices de cache hit para cache L2 para codificações com internalização do laço em Z. Antes, Durante e Após se referem a queda de desempenho observada anteriormente de acordo com o tamanho de domínio de entrada escolhido

Mas há uma otimização no código pioneiro que ainda não foi testada: o reuso dos registradores na dimensão Z. Como tal técnica só pode ser implementada quando o laço em Z for internalizado nas *threads*, as codificações que internalizam o laço em Z foram implementadas de duas formas, com e sem o reuso dos registradores.

Os experimentos da Figura 4 foram repetidos para as seis novas codificações. A Figura 8 apresenta o desempenho (GFlops) das seis novas codificações em conjunto com as três codificações originais, organizada por código original e por raio. Para cada código original e raio, o gráfico correspondente retrata o desempenho do código original (linha cinza pontilhada), do código original com internalização do laço em Z sem reuso dos registradores (linha azul tracejada) e do código original com internalização do laço em Z e com reuso dos registradores (linha vermelha contínua), todos em função do tamanho da grade.

Os gráficos da Figura 8 demonstram que não há ganho em internalizar o laço em Z no código BASE, pela óbvia razão que esse código não reusa memória rápida. Mas há

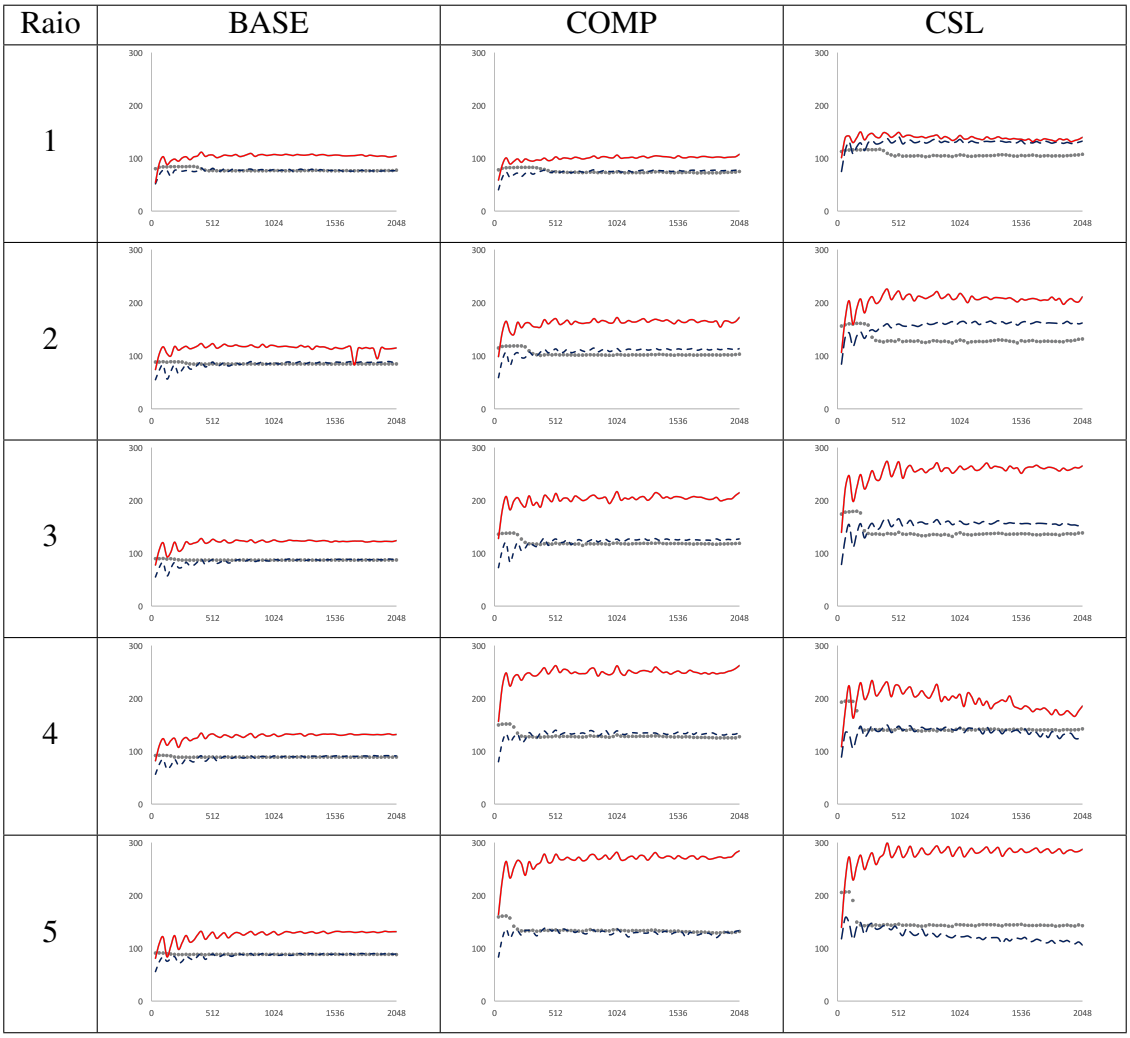

Figura 8. Desempenho para todas as codificações e tamanhos de estênceis. Linha cinza pontilhada: código original; Linha azul tracejada: código original com internalização do laço em Z sem reuso dos registradores; Linha vermelha contínua: código original com internalização do laço em Z e com reuso dos registradores

ganho em internalizar o laço em Z e reusar os registradores no código BASE, por substituir acessos a memória pelo reuso de registradores. Por sua vez, o COMP é beneficiado com as duas novas otimizações e seu desempenho é constantemente superior ao BASE. Já o desempenho do CSL melhora após internalizar o laço em Z apenas para raios pequenos, piorando para raios grandes, possivelmente pelo esgotamento da cache somente leitura. Uma indicação forte dessa afirmação é o ganho substancial do reuso dos registradores, pela substituição de acessos a essa memória pelo reuso de registradores. Creditamos a queda de desempenho com o aumento do domínio no caso CSL nos estênceis com raio 4 à provável colisão nas linhas de algum cache, a ser investigada posteriormente.

Concluímos que o código pioneiro pode ser substituído, com ganho de desempenho, pelo CSL com internalização do laço em Z e reuso dos registradores na grande maioria dos raios. A redução da complexidade da codificação é notável.

9. Ocupação da GPGPU

Durante a execução de um *kernel*, a ocupação de uma GPGPU pode ser medida através da relação entre a quantidade de *warps* ativas e o máximo de *warps* ativas permitido pelo hardware. Como os recursos da GPGPU são limitados, a ocupação máxima pode ficar abaixo de 100% dependendo do número de registradores alocados, do volume reservado de memória compartilhada e do tamanho do bloco de *threads*. Índices reduzidos de ocupação podem interferir negativamente no desempenho das aplicações. A Tabela 1 mostra a ocupação máxima teórica e a real medidas pelo profiler `nvprof` para os experimentos deste trabalho. Como todas as codificações apresentaram ocupação máxima teórica de 100% e ocupação real próxima da máxima pode-se concluir que a ocupação não afetou o desempenho medido.

Otimização	**7pontos**	**13pontos**	**19pontos**	**25pontos**	**31pontos**
BASE	100 (84,6)	100 (84,2)	100 (87,3)	100 (88,9)	100 (89,8)
INTZ	100 (91,1)	100 (96,2)	100 (96,9)	100 (94,9)	100 (96,9)
INTZ (Reg)	100 (86,7)	100 (88)	100 (91,4)	100 (93,6)	100 (93,9)
COMP	100 (91,7)	100 (90,5)	100 (91,6)	100 (91,6)	100 (91,2)
COMP + INTZ	100 (87,7)	100 (89,6)	100 (90,7)	100 (89,4)	100 (90)
COMP + INTZ (Reg)	100 (89,3)	100 (89,6)	100 (91,7)	100 (90,7)	100 (90,3)
CSL	100 (84,2)	100 (90,8)	100 (87,6)	100 (87)	100 (89,3)
CSL + INTZ	100 (97,7)	100 (97)	100 (95,7)	100 (97)	100 (96,1)
CSL + INTZ (Reg)	100 (96)	100 (97,4)	100 (92,8)	100 (94,5)	100 (95)

Tabela 1. Índice de ocupação teórico (medido) para cada código (%)

10. Conclusão e Trabalhos Futuros

O uso adequado da hierarquia de memória disponível no hardware é fundamental para atingir um bom desempenho na computação de estênceis. Devido a sua baixa intensidade computacional é necessário reutilizar as memórias mais rápidas. A otimização através do uso do cache somente leitura se mostrou muito eficiente pois o esforço de programação é mínimo e o resultado foi igual ou superior ao uso da memória compartilhada. O desempenho de qualquer código é afetado pelo raio do estêncil e pelo tamanho da grade. Há clara correlação entre a queda de desempenho e o esgotamento do cache L2, que pode ser aliviada pela internalização do laço em Z e praticamente eliminada pela inclusão do reuso dos registradores. Tais resultados podem mudar pela introdução de novidades arquitetônicas nas GPGPUs futuras, como foi o caso do cache somente leitura introduzida na arquitetura Kepler.

Avaliar o desempenho da especialização de *warps*, da blocagem temporal e de colisões no cache L2 são temas de trabalhos futuros.

Agradecimentos

Os autores agradecem à nVidia por disponibilizar o acesso à plataforma utilizada e à Paulo Roberto Pereira de Souza Filho e Pedro Pais Lopes pelas revisões cuidadosas e sugestões relevantes. Este trabalho foi parcialmente realizado com recursos do projeto HPC4E, financiamento número 689772 do acordo internacional entre o programa H2020 da Comunidade Européia e o MCTI/RNP.

Referências

Bauer, M., Cook, H., and Khailany, B. (2011). Cudadma: optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 12. ACM.

Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., and Yelick, K. (2009). Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review*, 51(1):129–159.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K. (2008). Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA. IEEE Press.

Hu, Y., Koppelman, D. M., Brandt, S. R., and Löffler, F. (2015). Model-driven auto-tuning of stencil computations on gpus. In *Histencils Workshop*, volume 2015.

Krotkiewski, M. and Dabrowski, M. (2013). Efficient 3d stencil computations using cuda. *Parallel Computing*, 39(10):533–548.

Maruyama, N. and Aoki, T. (2014). Optimizing Stencil Computations for NVIDIA Kepler GPUs. In Größlinger, A. and Köstler, H., editors, *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, pages 89–95, Vienna, Austria.

Mei, X. and Chu, X. (2015). Dissecting GPU memory hierarchy through microbenchmarking. *CoRR*, abs/1509.02308.

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, pages 79–84, New York, NY, USA. ACM.

Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. (2010). 3.5dd blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–13, Washington, DC, USA. IEEE Computer Society.

NVIDIA (2012). Kepler GK110 whitepaper.

Perkins, S., Marais, P., Zwart, J., Natarajan, I., and Smirnov, O. (2015). Montblanc: GPU accelerated radio interferometer measurement equations in support of bayesian inference for radio observations. *CoRR*, abs/1501.07719.

Schäfer, A. and Fey, D. (2011). High performance stencil code algorithms for gpgpus. In Sato, M., Matsuoka, S., Sloot, P. M., van Albada, G. D., and Dongarra, J., editors, *Proceedings of the International Conference on Computational Science, ICCS 2011*, volume 4, pages 2027 – 2036, Netherlands. Elsevier.

Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. (2011). The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM. Compilador de DSL para estenceis gerando codigo otimizado para caches em CPUs.

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76.