universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Trabalho 2

Semi-Global Matching stereo processing using CUDA

Curso	[8240] MI em Engenharia de Computadores e Telemática
Disciplina	[47022] Arquitetura de Computadores Avançada
Ano letivo	2015/2016

Alunos	[68779] Rui Oliveira
	[68129] Tomás Rodrigues
Prática	P1
Docente	Professor Nuno Lau / Professor José Luís Azevedo

Aveiro, 21 de Janeiro de 2016

Conteúdo

1 Introdução

Os CPUs[1] utilizados hoje em dia nos computadores são cada vez mais poderosos, capazes de realizar tarefas em cada vez menos tempo e possibilitar interatividade com tempos de resposta cada vez mais curtos. No entanto devido ao seu desenho concebido para uso geral e interativo, não consegue um débito tão elevado quanto alguns outros tipos de hardware. O GPU[2] é um componente de hardware capaz de elevadas taxas de processamento devido principalmente às suas caraterísticas de computação paralela conseguindo renderizar grafismo cada vez mais exigente e complexo. Eventualmente, começouse a investigar a possibilidade de fazer uso do GPU para efetuar também tarefas não gráficas. Assim, surgiu o GPGPU[3]: a capacidade de utilizar o GPU para computação tipicamente efetuada por um CPU mas de forma muitíssimo mais eficiente [1]. No entanto, existem algumas restrições que impedem que tudo o que é geralmente feito no CPU possa (ou deva) ser executado no GPU: necessidades muito específicas dos programas, que normalmente são pouco paralelizáveis e muito penalizados pela latência das cópias entre memória principal e memória do GPU, pois requerem tempos de resposta curtos para uma boa interatividade, são alguns exemplos.

Figura 1: CUDA - Nvidia

Pretende-se através deste relatório expor sob forma escrita, o nosso desempenho e objetivos alcançados no trabalho 2 desenvolvido no âmbito da unidade curricular de Arquitetura de Computadores Avançada.

Este trabalho foca-se na utilização de um GPU para a aplicação de um algoritmo de processamento de imagens e respetiva comparação com a performance de um CPU, para a mesma tarefa. O GPU é tipicamente composto por uma grande quantidade de pequenos processadores, pelo que a programação

[1]Central Processing Unit
[2]Graphics Processing Unit
[3]General Purpose Graphics Processing Unit

para este é inerentemente paralela e difícil de aplicar a programas mais comuns e sequenciais. A tecnologia de GPGPU a ser utilizada é o CUDA[4], propriedade da NVIDIA, onde serão desenvolvidas e comparadas diferentes formas de executar o algoritmo, tirando partido de várias funcionalidades do dispositivo em questão: memória global, memória partilhada e memória de textura.

1.1 O problema

O problema apresentado tem como o objetivo calcular as diferenças entre duas imagens(esquerda e direita). Esta nova imagem atribui um valor a cada pixel proporcional à distância entre o pixel respetivo da imagem da esquerda e o correspondente da imagem à direita.

Cada imagem será modelada como uma matriz de inteiros de valores entre 0 a 255. Os valores correspondem ao cor/brilho do pixel, indicando o valor 0 preto e o valor 255 num pixel a cor branca.

Primeiro será determinado o custo de cada pixel na imagem à esquerda para diferentes valores de disparidade (até um determinado valor limite), comparando esta imagem com a imagem da direita, após isso, é feita uma agregação dos custos em cada ponto em várias direções e por último, a partir deste custo a imagem final pode ser calculada encontrando, para cada pixel, a melhor disparidade, ou seja, a que faz com que o custo final seja o mínimo possível.

É nos facultada uma uma versão do algoritmo a correr no CPU e o esqueleto de código para executar e comparar tempos do GPU.

1.2 Os resultados

Os resultados obtidos em cada tarefa deste trabalhos baseiam-se em tempos de execução de cada programa nas duas implementações possíveis (host e device). Foram calculadas médias destes tempos tendo por base 4 tentativas estáveis, isto é, tentativas cuja variação do tempo foi quase negligenciável. Com as medias obtidas foi calculada a razão entre o tempo médio de execução do host e o tempo médio de execução do device. Esta razão dá-nos o valor do *speedup*.

Foi usado o device de índice 0 do nikola.ieeta.pt. Testaram-se quatro imagens diferentes e dois valor possíveis de disp_rang (32 e 64).

Na secção 5 está descrito como é possível compilar e executar o código fonte.

[4]Compute Unified Device Architecture

2 Memória Global

Aqui foi implementado a passagem do código que corria em CPU para passar a usar ser executado no GPU usando a memória global do kernel. Assim sendo foi necessário remover os ciclos que percorriam os pixéis da imagem, pois deixaram de ser necessários, as atribuição de threads já faz com que cada pixel da imagem seja processado.

2.1 Tarefa 1 : *determine_costs()*

2.1.1 Implementação

De forma a implementar uma solução para o problema em CUDA, o primeiro passo foi arranjar a melhor configuração dos blocos e grelha ao preparar a execução dos kernels. Para esta tarefa a configuração que achámos conveniente é a seguinte:

```
int block_x = (disp_range<=32) ? 4 : 3;
int block_y = (disp_range<=32) ? 4 : 2;
int block_z = disp_range;

int grid_x = ceil((float)nx/block_x);
int grid_y = ceil((float)ny/block_y);

dim3 dimBlock(block_x,block_y, block_z);
dim3 dimGrid(grid_x,grid_y,1);
```

Numa primeira versão apenas tínhamos os block_x e block_y, posteriormente foi necessária a inclusão de um novo bloco (block_z) que modo que nos permitisse remover um ciclo existente na função determine_costs(). Mais a frente iremos explicar a necessidade da eliminação deste ciclo.

Calculadas as dimensões da grid e dos blocos é então possível passar ao desenvolvimento das soluções, tendo como mapeamento entre os pixéis da imagem e as threads do dispositivo as seguintes fórmulas:

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
int j = threadIdx.y + blockIdx.y * blockDim.y;
int d = threadIdx.z + blockIdx.z * blockDim.z;
```

Inicialmente a nossa implementação em kernel da função *determine_costs()* era a seguinte:

```
__global__ void devDetermine_costs(const int *left_image,
    const int *right_image, int *costs, const int nx,
    const int ny, const int disp_range) {

    int j = threadIdx.y + blockIdx.y * blockDim.y;
    MYASSERT(j < ny);
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    MYASSERT(i < nx);

    for ( int d = 0; d < disp_range; d++ ) {
        COSTS(i,j,d)=255;
        if (i >= d){
            COSTS(i,j,d) = abs( LEFT_IMAGE(i,j) - RIGHT_IMAGE(i-d,j) );
        }
    }
}
```

Posteriormente conclui-se que ao retirar o ciclo que itera em até *disp_range* o kernel seria executado mais rapidamente, tal como é possível constatar pelos resultados apresentados na próxima secção.

Temos portanto, a seguinte implementação final:

```
__global__ void devDetermine_costs(const int *left_image,
    const int *right_image, int *costs, const int nx,
    const int ny, const int disp_range) {

    int j = threadIdx.y + blockIdx.y * blockDim.y;
    MYASSERT(j < ny);
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    MYASSERT(i < nx);
    int d = threadIdx.z + blockIdx.z * blockDim.z;
    MYASSERT(d < disp_range);

    COSTS(i,j,d)=255;
    if (i >= d){
        COSTS(i,j,d) = abs( LEFT_IMAGE(i,j) - RIGHT_IMAGE(i-d,j) );
    }

}
```

2.1.2 Resultados

Com implementação primeiramente apresentada (com for disp_rang) obtivemos os seguintes resultados para a imagem default.

Task1 - Memória Global					
Imagem utilizada: *bull* (rbull + lbull)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5263,124512	5226,014648	**Amostras (ms)**	19880,460938	19514,443359
	5290,536133	5166,046875		20301,421875	19866,218750
	5272,479492	5166,014648		19975,470703	20266,220703
	5216,019043	5104,745117		20347,777344	20493,687500
Media (ms)	5267,802002	5166,030762	**Media (ms)**	20138,446289	20066,219727
Speedup	**1,019700**		**Speedup**	**1,003599**	

Figura 2: Speedups obtidos para disprange de 32 e 64 na imagem venus com a primeira implementação (com for)

Posteriormente, implementamos a segunda implementação (sem for disp_rang) e obtivemos os seguintes resultados para a imagem default.

Task1 - Memória Global					
Imagem utilizada: *bull* (rbull + lbull)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5119,915527	5029,683105	**Amostras (ms)**	20551,445312	20249,800781
	5188,753906	5037,687500		19722,380859	19491,730469
	5200,616699	5032,174316		19720,380859	19525,048828
	5192,601562	5036,449707		20337,343750	19774,751953
Media (ms)	5190,677734	5034,312012	**Media (ms)**	20029,862305	19649,900391
Speedup	**1,031060**		**Speedup**	**1,019337**	

Figura 3: Speedups obtidos para disprange de 32 e 64 na imagem bull

Concluímos portanto que se torna mais eficiente a segunda implementação, uma vez que os speedups obtidos na figura 3 são superiores aos da figura 2 (embora não muito relevante).

Os tempos médios e speedups para as restantes imagem com esta implementação são apresentados de seguida:

Task1 - Memória Global					
Imagem utilizada: *cones* (rcones + lcones)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5528,250488	5384,147461	**Amostras (ms)**	19670,035156	19529,615234
	5151,625977	5079,589844		19948,933594	19752,835938
	5133,340820	5079,803223		19920,527344	19795,427734
	5316,597656	5138,058105		20135,080078	19642,923828
Media (ms)	5234,111817	5108,930664	**Media (ms)**	19934,730469	19697,879883
Speedup	**1,024502**		**Speedup**	**1,012024**	

Figura 4: Speedups obtidos para disprange de 32 e 64 na imagem cones

Task1 - Memória Global					
Imagem utilizada: *teddy* (rteddy + lteddy)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5419,835449	5377,018555	**Amostras (ms)**	20908,429688	20107,496094
	5210,458496	5157,064941		19940,103516	19818,335938
	5230,944824	5151,528320		20453,316406	20219,257812
	5219,420410	5159,827637		21261,494141	20804,990234
Media (ms)	5225,182617	5158,446289	**Media (ms)**	20680,873047	20163,376953
Speedup	**1,012937**		**Speedup**	**1,025665**	

Figura 5: Speedups obtidos para disprange de 32 e 64 na imagem teddy

Pode-se concluir que o speedup médio para a nossa implementação da função determine_costs() em GPU é aproximadamente de 1,023827101. Por outro lado, a nossa implementação a ser executada com uma disprange de 32 e 64 tem um speedup aproximado de 1,023827101 e 1,0225008661, respetivamente.

Task1 – Memória Global					
Imagem utilizada: *venus* (rvenus + lvenus)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5310,101074	5169,916016	**Amostras (ms)**	20207,886719	19890,007812
	5064,492676	4996,901855		19781,576172	19307,798828
	5214,611328	5092,214844		20103,783203	19372,925781
	5108,086426	4993,333984		19737,341797	19330,626953
Media (ms)	5161,348877	5044,558350	**Media (ms)**	19942,679688	19351,776367
Speedup	**1,023152**		**Speedup**	**1,030535**	

Figura 6: Speedups obtidos para disprange de 32 e 64 na imagem venus

2.2 Tarefa 2: *iterate_direction()*

2.2.1 Implementação

Para esta tarefa foi criada uma função *void devIterate_direction* onde se encontra implementada todos os blocos e grids necessários. Para além disso, é nesta função onde são invocados todos os kernels existentes para esta tarefa: devIterate_direction_dirxpos, devIterate_direction_dirypos, devIterate_direction_dirxneg, devIterate_direction_diryneg. Para esta função foi utilizada a mesma estrutura da fornecida e implementada para o CPU. O blocos e grids utilizadas nesta tarefa são os seguintes:

```
int block_x = disp_range;
int block_y = 1;
dim3 dimBlock(block_x,block_y);
dim3 dimGridy(1,nx);
dim3 dimGridx(1,ny);
```

A invocação dos diferentes kernels é feito da seguinte forma:

```
devIterate_direction_dirxpos<<<dimGridx, dimBlock>>>
    (dirx,devLeft_image,devCosts,
    devAccumulated_costs, nx, ny, disp_range);

devIterate_direction_dirypos<<<dimGridy, dimBlock>>>
    (diry,devLeft_image,devCosts,
    devAccumulated_costs, nx, ny, disp_range);
```

```
devIterate_direction_dirxneg<<<dimGridx, dimBlock>>>
    (dirx,devLeft_image,devCosts,
    devAccumulated_costs, nx, ny, disp_range);

devIterate_direction_diryneg<<<dimGridy, dimBlock>>>
    (diry,devLeft_image,devCosts,
    devAccumulated_costs, nx, ny, disp_range);
```

A titulo de exemplo, a função devIterate_direction_dirxpos() encontra-se implementada de seguida:

```
__global__ void devIterate_direction_dirxpos(const int dirx,
    const int *left_image, const int* costs,
    int *accumulated_costs,  const int nx, const int ny,
    const int disp_range ) {

    const int WIDTH = nx;
    int j = blockIdx.y;
    int d = threadIdx.x;

    if(j < ny && d < disp_range){
    for ( int i = 0; i < WIDTH; i++ ) {
        if(i==0) {
            ACCUMULATED_COSTS(0,j,d) += COSTS(0,j,d);
        }
        else {
            devEvaluate_path( &ACCUMULATED_COSTS(i-dirx,j,0), &COSTS(i,j,0),
            abs(LEFT_IMAGE(i,j)-LEFT_IMAGE(i-dirx,j)),
            &ACCUMULATED_COSTS(i,j,0), nx, ny, disp_range);
        }
        __syncthreads();
    }
    }

}
```

2.2.2 Resultados

Task2 – Memória Global					
Imagem utilizada: *bull* (rbull + lbull)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5161,980957	453,772095	**Amostras (ms)**	20235,861328	979,065308
	5322,588867	453,626343		21026,621094	998,099365
	5095,584961	452,889038		21021,984375	998,127380
	5221,871094	457,866638		20252,115234	979,413391
Media (ms)	5191,926026	453,699219	**Media (ms)**	20637,049805	988,756378
Speedup	**11,443542**		**Speedup**	**20,871724**	

Figura 7: Speedups obtidos para disprange de 32 e 64 na imagem bull

Task2 – Memória Global					
Imagem utilizada: *cones* (rcones + lcones)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5304.345703	482.177460	**Amostras (ms)**	19576,445312	1000,617737
	5302,013672	483,940216		20291,218750	1000,797241
	5302,800781	478,675079		20179,826172	1002,388428
	5226,103027	479,152191		20473,210938	1010,726074
Media (ms)	5302,013672	479,152191	**Media (ms)**	20235,522461	1001,592835
Speedup	**11,065406**		**Speedup**	**20,203342**	

Figura 8: Speedups obtidos para disprange de 32 e 64 na imagem cones

Task2 – Memória Global					
Imagem utilizada: *venus* (rvenus + lvenus)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5060,730957	453,436249	**Amostras (ms)**	34898,796875	1053,982300
	5233,284668	454,549255		20612,046875	965,864868
	5023,804688	453,158661		20040,343750	965,535645
	5106,774414	453,271698		19364,658203	957,127930
Media (ms)	5083,752686	453,353974	**Media (ms)**	20326,195313	965,700257
Speedup	**11,213650**		**Speedup**	**21,048141**	

Figura 9: Speedups obtidos para disprange de 32 e 64 na imagem venus

Imagem utilizada: *teddy* (rteddy + lteddy)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5369,225098	485,829529	**Amostras (ms)**	19970,822266	1020,881287
	5254,530273	479,311096		31748,097656	1109,072144
	5364,057129	484,877747		25182,617188	1011,819824
	5313,092773	478,977325		20007,583984	1011,551453
Media (ms)	5338,574951	482,094422	**Media (ms)**	22595,100586	1016,350556
Speedup	**11,073712**		**Speedup**	**22,231602**	

Figura 10: Speedups obtidos para disprange de 32 e 64 na imagem teddy

Pode-se concluir que o speedup médio para a nossa implementação da função iterate_direction() em GPU é aproximadamente de 15,823441894. Por outro lado, a nossa implementação a ser executada com uma disprange de 32 e 64 tem um speedup aproximado de 11,143681067 e 20,959932365, respetivamente.

2.3 Tarefa 3 : *inplace_sum_views()*

2.3.1 Implementação

Nesta função calculamos a matriz de custos através da soma de duas matrizes de custos(im1 e im2). O resultado é acumado na matriz de curso im1. A função implementada está descrita abaixo:

```
__global__ void devInplace_sum_views( int * im1, const int * im2,
    const int nx, const int ny,
    const int disp_range ) {

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    MYASSERT(x < nx);

    int y = threadIdx.y + blockIdx.y * blockDim.y;
    MYASSERT(y < ny);

    int z = threadIdx.z + blockIdx.z * blockDim.z;
    MYASSERT(z < disp_range);

    int id = x + y * nx + z * nx * ny;
    im1[id] += im2[id];
}
```

2.3.2 Resultados

Task3 – Memória Global					
Imagem utilizada: *bull* (rbull + lbull)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5210,172852	387,460754	**Amostras (ms)**	20445,027344	854,422668
	5167,055664	387,056915		19714,447266	853,263611
	5145,578125	387,370026		19682,248047	852,108154
	5119,972656	386,837830		20669,414062	854,299622
Media (ms)	5156,316895	387,213471	**Media (ms)**	20079,737305	853,781617
Speedup	**13,316471**		**Speedup**	**23,518587**	

Figura 11: Speedups obtidos para disprange de 32 e 64 na imagem bull

Task3 – Memória Global					
Imagem utilizada: *cones* (rcones + lcones)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5359,023438	405,989990	**Amostras (ms)**	20126,531250	876,110352
	5128,068359	417,362732		19731,597656	878,653992
	5288,684570	405,465698		20069,748047	876,482727
	5253,674805	405,690277		19629,373047	876,706604
Media (ms)	5271,179688	405,840134	**Media (ms)**	19900,672852	876,594666
Speedup	**12,988315**		**Speedup**	**22,702252**	

Figura 12: Speedups obtidos para disprange de 32 e 64 na imagem cones

Task3 – Memória Global					
Imagem utilizada: *venus* (rvenus + lvenus)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5223,134766	387,140442	**Amostras (ms)**	20279,976562	840,213806
	5083.259766	386.722504		19563,488281	838,763245
	5289.880859	386.205963		19540,708984	839,834717
	5246.773926	386.394226		20350,009766	838,210449
Media (ms)	5223,134766	387,140442	**Media (ms)**	19921,732422	839,298981
Speedup	**13,491576**		**Speedup**	**23,736157**	

Figura 13: Speedups obtidos para disprange de 32 e 64 na imagem venus

Task3 – Memória Global					
Imagem utilizada: *teddy* (rteddy + lteddy)					
disp_range = 32			disp_range = 64		
	Host	**Device**		**Host**	**Device**
Amostras (ms)	5287,667480	410,756561	**Amostras (ms)**	20322,292969	888,614807
	5715,913574	418,937683		20127,496094	886,533875
	5611,432617	416,197876		20035,535156	886,280396
	5401,180664	420,595612		20007,736328	887,450867
Media (ms)	5506,306641	417,567780	**Media (ms)**	20081,515625	886,992371
Speedup	**13,186618**		**Speedup**	**22,640009**	

Figura 14: Speedups obtidos para disprange de 32 e 64 na imagem teddy

Pode-se concluir que o speedup médio para a nossa implementação da função inplace_sum_views() em GPU é aproximadamente de 18,065792795. Por outro lado, a nossa implementação a ser executada com uma disprange de 32 e 64 tem um speedup aproximado de 13,25154424 e 23,110419462, respetivamente.

2.4 Tarefa 4 : *create_disparity_view()*

2.4.1 Implementação

Finalmente os cálculos para a imagem final são determinados da seguinte forma:

```
__global__ void devCreate_disparity_view( const int *accumulated_costs ,
    int * disp_image, const int nx,
    const int ny, const int disp_range) {

    int j = threadIdx.y + blockIdx.y * blockDim.y;
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    if(i<nx && j< ny)
        DISP_IMAGE(i,j) = 4 *
            devFind_min_index( &ACCUMULATED_COSTS(i,j,0), disp_range );
}
```

2.4.2 Resultados

disp_range = 32		
	Host	**Device**
Amostras (ms)	5204,465332	364,415100
	5114,190430	363,392609
	5118,705078	363,293243
	5136,582520	364,121307
Media (ms)	5127,643799	363,756958
Speedup	**14,096346**	

Figura 15: Speedups obtidos para disprange de 32 na imagem bull

disp_range = 32		
	Host	**Device**
Amostras (ms)	5146,655273	349,118927
	9029,860352	349,720734
	8133,778809	355,081146
	5250,994629	349,796570
Media (ms)	6692,386719	349,758652
Speedup	**19,134299**	

Figura 16: Speedups obtidos para disprange de 32 na imagem cones

disp_range = 32		
	Host	**Device**
Amostras (ms)	5214,678223	345,280670
	5075,844727	345,919800
	5306,715332	345,922852
	5057,084961	345,164795
Media (ms)	5145,261475	345,600235
Speedup	**14,887899**	

Figura 17: Speedups obtidos para disprange de 32 na imagem venus

disp_range = 32		
	Host	**Device**
Amostras (ms)	5425,670898	380,958496
	5239,999512	380,399963
	5231,223633	380,216339
	5476,764160	382,828247
Media (ms)	5332,835205	380,679230
Speedup	**14,008737**	

Figura 18: Speedups obtidos para disprange de 32 na imagem teddy

Pode-se concluir que o speedup médio para a nossa implementação da função create_disparity_view() em GPU é aproximadamente de 14,492122425 (com disprange de 32).

3 Memória Partilhada

Esta versão utiliza a memória partilhada pelas threads de um bloco da grelha do device que é a memória para a qual são copiados a partir do host os dados da sub-imagem e os restantes pixeis, que não pertencem ao bloco, mas no entanto são necessários para a robustez e correcção do algoritmo de Harris.. Semelhante à versão anterior, a imagem resultante é armazenada na memória global. A memória partilhada é a memória que tem menos latência no acesso à memória, prevendo que a sua implementação será a mais rápida a devolver a imagem resultante.

Não nos foi possível a implementação desta versão para todas as tarefas devido à falta de tempo e às quebras de disponibilidade por parte do servidor.

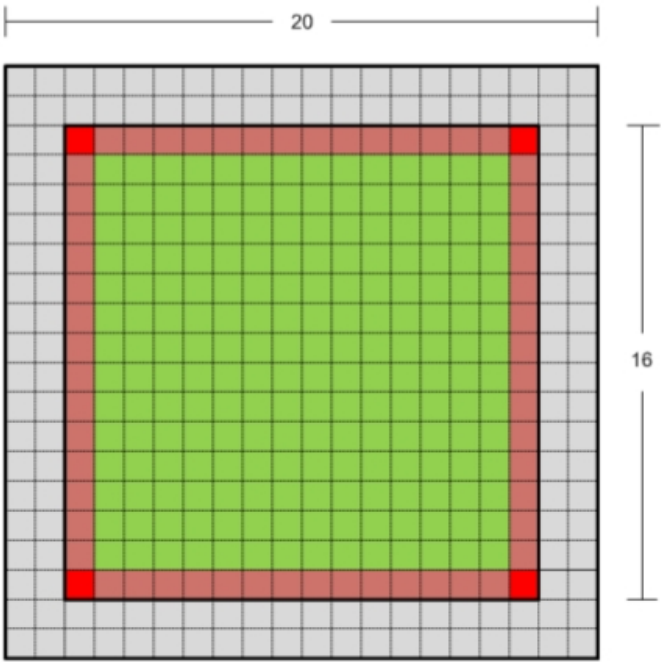

Figura 19: Representação da memória partilhada de um bloco

4 Texture Memory

Esta segunda versão foi desenvolvida tendo também por base a versão para o host disponibilizada. Esta versão utiliza a memória de texturas do device que é a memória para a qual são copiados a partir do host os dados da imagem. A imagem resultante é armazenada na memória global.

Ao utilizar-mos a memória de texturas, uma memória que apenas pode ser lida pelas threads, com uma menor latência comparando com a versão anterior(memória global) conseguimos tornar mais rápida a execucção do algoritmo.

Tal como foi dito anteriormente, não nos foi possível a implementação desta versão para as tarefas devido à falta de tempo e às quebras de disponibilidade por parte do servidor.

5 Compilação e execução

- Imagens disponíveis
  - *bull.pgm* (imagem direita e esquerda)
  - *cones.pgm* (imagem direita e esquerda)
  - *teddy.pgm* (imagem direita e esquerda)
  - *venus.pgm* (imagem direita e esquerda)
- Compilação do ficheiro *sgm.cu*

  ```
  make
  ```

- Execução do programa *sgm.cu*

  ```
  Usage: ./sgm [-h] [-d device] [-l leftimage] [-r rightimage]
  [-o dev_dispimage] [-t host_dispimage] [-p disprange]
  ```

  - device: device utilizado no servidor
  - leftimage: nome da imagem da esquerda
  - rightimage: nome da imagem da direita
  - dev_dispimage: nome da imagem onde será guardado o resultado executado na GPU
  - host_dispimage: nome da imagem onde será guardado o resultado executado na CPU
  - disprange: valor possível para o disprange;

6 Conclusões

Chegado ao final deste relatório, é nossa intenção efetuar uma retrospetiva da evolução do mesmo, tendo em conta os problemas com que nos deparámos, e principais conclusões retiradas.

Começando pelas tarefas 1 e 4, foi preciso perder algum tempo para perceber como tudo funcionava e encaixava como um todo, mas após essa pesquisa estas tarefas foram ultrapassadas com sucesso e sem dificuldades.

Relativamente às tarefas 3 e 4 deparámos-nos com alguns problemas para passar estas funções da linguagem C++ para CUDA mas conseguimos passado algum tempo e fizémos as otimizações que conseguimos de modo a obter o melhor speedup possível. Após testes intensivos tudo o foi implementado encontra-se totalmente funcional e com um speedup apartir da tarefa 2 bastante elevado na nossa opinião.

Na tabela seguinte encontra-se um tabela comparativa dos vários speedups obtidos nas várias tarefas para um disprang de 32. Conclui-se que em cada tarefa houve um aumento significativo do tempo de execução de cada programa, em especial na tarefa 2.

Tarefa 1	1,023827101
Tarefa 2	11,143681067
Tarefa 3	13,25154424
Tarefa 4	14,492122425

Tabela 1: Tabela comparativa speedups das diferentes tarefas

No final acabámos por não fazer uma implementação em memória partilhada nem com texturas, mas pensamos ter um trabalho com um algoritmo de memória global bastante eficiente e funcional.

Este trabalho permitiu-nos aprofundar as nossas competências adquiridas nas aulas práticas da unidade curricular de Arquitetura de Computadores Avançada e aprender um pouco mais sobre a tecnologia CUDA.

Referências

[1] Cuda Programming Guide. URL: http://elearning.ua.pt/pluginfile.php/232981/mod$_r esource/$

[2] CUDA C Best Practices. URL: http://elearning.ua.pt/pluginfile.php/232982/mod$_r esource/co$

[3] Real-time Stereo Vision: Optimizing Semi-Global Matchings. URL: http://elearning.ua.pt/pluginfile.php/287661/mod$_r esource/content/2/ImproveSGM.pdf$

[4] Using Shared Memory in CUDA C/C++. URL: http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/