Programação de Alto Desempenho

Atividade 7 — Programação CUDA

Lucas Santana Lellis — 69618
PPGCC — Instituto de Ciência e Tecnologia
Universidade Federal de São Paulo

I. Introdução

Nesta atividade foram realizados experimentos com programação CUDA para placas aceleradoras. Cada experimento foi realizado 3 vezes, e os resultados apresentados são a média dos resultados obtidos em cada um deles, sendo calculado o speedup em comparação à execução em CPU. As especificações da máquina utilizada estão disponíveis na TabelaI.

CPU	Intel i7 990X	**Cores**	6
Threads	12	**Clock**	3.47 GHz
Cache	12 MB	**RAM**	20 GB
SO	Ubuntu 14.04	**Kernel**	3.13.0
GPU	Tesla K40	**Threads**	2880
Mem.	12Gb GDDR5	**CUDA Cap.**	3.5
GCC	4.8.5	**NVCC**	V7.5.17

Tabela I: Especificações da Máquina

II. Exercício I — Difusão de calor

Neste exercício foi feita a paralelização em CUDA do algoritmo da difusão de calor em um domínio unidimensional através de diferenças finitas.

Foram implementados dois kernels, um para a transmissão de calor, chamado "Atualiza", e um para a operação de redução que retorna o valor máximo no vetor, chamado "Maximo", em uma versão alternativa, a inicializacao do vetor também é paralelizada pelo kernel "Inicializacao".

Alguns recursos do algoritmo original não foram implementados em CUDA, como o cálculo de x, que acumulava dx à cada iteração. Além disso, a operação de redução "Maximo" apenas retorna o maior valor, ignorando a posição do maior valor do vetor. De qualquer forma, em ambas implementações o resultado obtido é o mesmo: 99.8486.

No algoritmo em CUDA, foram obtidos os tempos da tabelaII, onde *HostToDevice* é o tempo de transmissão do vetor da memória local para a placa aceleradora, *Tempo kernels* é o tempo total da execução da etapa de transmissão de calor, em que são feitas chamadas sucessivas do kernel "Atualiza", *Tempo medio* é o tempo médio da execução dos kernels, *DeviceToHost* é o tempo de transmissão da saída da operação de redução para a CPU, e finalmente o *Tempo Total* é o tempo total da execução do algoritmo.

HostToDevice:	0.000165
Tempo kernels:	0.112622
Tempo medio:	0.000011
DeviceToHost:	0.012765
Tempo Total:	0.125966

Tabela II: Tempos obtidos na execução CUDA.

Agora, na Tabela III é feita a comparação entre a execução em CPU e em GPU. O speedup da operação foi de 10 vezes se comparado à execução em apenas uma thread na CPU.

CPU	1.34425
GPU	0.125966
CPU/GPU	10.671530

Tabela III: Comparação da execução em CUDA vs CPU.

III. Exercício II — nvprof e nvvp

Neste exercício utilizei o nvprof e nvvp para analisar a execução do exercício da seçãoII. Na Figura1 está a saída do nvvp, e na Figura2 um resumo da saída do nvprof, com os argumentos "nvprof –print-gpu-trace bin/ex01 ".

Uma análise mais profunda da Figura2 revela algumas informações interessantes: A transmissão dos dados iniciais para o dispositivo durou 136.39us, sendo enviados 781.25KB a uma taxa de 5.4629GB/s; Foram realizadas 50000 chamadas do kernel "Atualiza", cada uma durando em torno de 9us; A operação de redução "Maximo" em duas etapas, a primeira com 16us e a segunda com 4us; Por fim, a cópia do resultado da redução que durou 3us para enviar 8 bytes à memória principal.

IV. Exercício III — Numeros Aleatorios

Nesse exercício foi implementado um kernel em CUDA para preencher um vetor com números aleatórios do tipo float utilizando a função "cuRand". Neste exemplo, não foi necessário separar a etapa de atribuição das sementes da etapa de obtenção do número aleatório, uma vez que a utilização da função cuRand é realizada em apenas uma etapa em todo o código.

Foi feita uma implementação da mesma operação, utilizando a função "rand" para gerar um vetor de números aleatorios na CPU. A comparação entre os tempos de execução está na TabelaIV, e vemos que o speedup foi de 20 vezes com relação á execução em CPU com apenas uma thread.

CPU	0.010963
GPU	0.000534
CPU/GPU	20.529962

Tabela IV: Comparação da execução em CUDA vs CPU.

V. Exercício IV — Somatorio de Nros. Aleatorios

Utilizando o código do exercício da SeçãoIV, o exercício atual compreende no somatório dos valores de um vetor

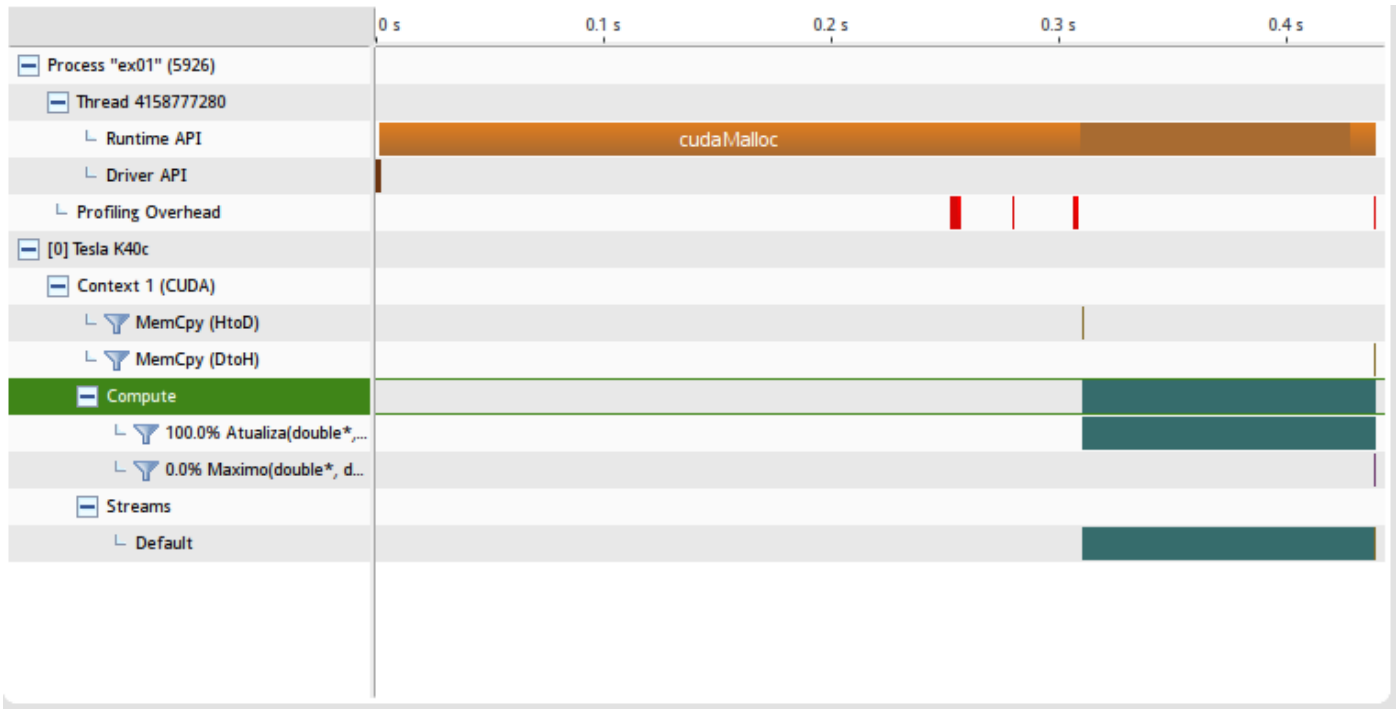

Figura 1: Saída do NVVP sobre o programa em CUDA da SeçãoII.

```
==7925== NVPROF is profiling process 7925, command: bin/ex01
Size: 99999, numBlks: 391, numThds: 256, mult: 100096
Inicio: qtde=100000, dt=1e-06, dx=1e-05, dx2=1e-10, kappa=4.5e-05, const=0.45
Iteracoes previstas: 10000
        HostToDevice: 0.00011664
        Inicializacao: 0.000440311
        Tempo kernels: 0.115724
        Tempo medio: 1.15712e-05
        DeviceToHost: 0.0b110544
        Tempo Total: 0.127257
Maior valor = 99.8486
==7925== Profiling application: bin/ex01
==7925== Profiling result:
   Start  Duration Grid Size Block Size ...     Size  Throughput  Name
403.55ms  136.39us         -          - ... 781.25KB  5.4629GB/s  [CUDA memcpy HtoD]
403.78ms  10.752us (391 1 1)  (256 1 1) ...        -           -  Atualiza(double*, double*, int) [185]
403.80ms  9.8250us (391 1 1)  (256 1 1) ...        -           -  Atualiza(double*, double*, int) [190]
[ ... ]
530.30ms  9.8240us (391 1 1)  (256 1 1) ...        -           -  Atualiza(double*, double*, int) [50180]
530.31ms  9.7920us (391 1 1)  (256 1 1) ...        -           -  Atualiza(double*, double*, int) [50185]
530.32ms  16.768us (391 1 1)  (256 1 1) ...        -           -  Maximo(double*, double*, int) [50190]
530.34ms  4.0000us   (1 1 1)  (256 1 1) ...        -           -  Maximo(double*, double*, int) [50195]
530.35ms  3.1040us         -          - ...       8B  2.4579MB/s  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread.
This number includes registers used internally by the CUDA driver and/or
tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
```

Figura 2: Saída do NVPROF sobre o programa em CUDA da SeçãoII.

preenchido com números aleatórios. Trata-se também de uma operação de redução em CUDA, realizada de forma semelhante à Seção III.

Um teste simples para somar um vetor de 1000000 posições com apenas o valor 1 foi suficiente para revelar uma deficiência nessa abordagem de redução utilizada, pois o valor obtido era muito inferior ao esperado. Logo, chegamos à conclusão que a utilização dessa função de redução para vetores com muitos elementos e blocos de tamanho fixo poderia causar resultados inconsistentes, e que para que a função de redução funcione corretamente é necessário que a chamada dos kernels seja feita com número de trheads superior ao número de blocos.

Assim, foi feita uma comparação do tempo de execução da mesma operação em CPU e em GPU (CUDA), e os resultados estão na TabelaV. Como pode-se ver, o speedup foi de 21 vezes com relação à execução em CPU em uma só thread.

CPU	0.000612
GPU	0.013140
CPU/GPU	21.470588235

Tabela V: Comparação da execução em CUDA vs CPU.

VI. Exercício V — Jogo da Vida

Finalmente, o último exercício propõe a comparação da implementação do jogo da vida em Cuda e OpenMP. A implementação em CUDA se deu atraves da implementação de um kernel que representa cada iteração do tabuleiro do jogo da vida, foi feito o uso da memória rápida da GPU, uma vez que há uma grande interação com os dados da memória, com uma vizinhança-8.

Ambos os programas foram testados com as mesmas condições iniciais, e após a execução do algoritmo em um tabuleiro de tamanho 1000×1000, obtemos o resultado da Figura 3.

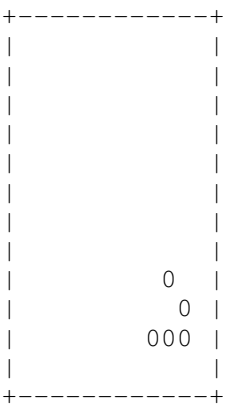

Figura 3: Resultado do teste em tabuleiro 1000×1000.

Foi feita então uma comparação dos tempos de execução do algoritmo paralelizado em 12 threads usando OpenMP com os tempos obtidos em CUDA na K40, disponível na TabelaVI.

Disp.	**Tempo**	**Speedup**
CPU (1 thread)		
CPU (12 threads)		
GPU (CUDA)		

Tabela VI: Comparação da execução em CUDA vs x CPU.