Relatório 10

Programação de Alto Desempenho (HPC) usando GPU/CUDA –

Computação Paralela

Cristiano Lopes Moreira

Matrícula: 119103-0

Sumário

1. Introdução

A crescente demanda por sistemas de uso em tempo real em conjunto com os limites físicos que delimitam o crescimento da capacidade de processamento dos computadores vem desde 1955, pela iniciativa da IBM com o arquiteto de computadores Gene Amdahl, impulsionando desenvolvimento de tecnologias de Programação de Alto Desempenho (HPC) e possibilitando a interconexão de múltiplos computadores e processadores com suas unidades de memórias para dividir uma tarefa e compartilhar a capacidade de cada unidade de processamento.

Dentre as técnicas HPC desenvolvidas encontra-se a técnica de computação paralela com placa GPU (graphics processing unit) que que possuem milhares de processadores simplificados com grande desempenho em tarefas paralelas.

O objetivo deste trabalho é implementar a tecnologia de computação paralela utilizando GPU/CUDA, verificar e quantificar seus ganhos e perdas na tarefa de cálculo numérico de integral pelo método de Monte Carlo.

2. Desenvolvimento teórico

Como já afirmado por Amdahl (1967), a décadas os profetas expressaram a alegação de que a organização de um único computador atingiu seus limites, e que avanços verdadeiramente significativos podem ser feitos apenas pela interconexão de uma multiplicidade de computadores de maneira a permitir uma solução cooperativa.

Geralmente, métodos de computação paralela têm sido usados para aumentar o desempenho do sistema ou aumentar sua disponibilidade, porém a busca por esta interconexão trilha os desafios de conflitos no uso de recursos simultâneos como memória, discos, i/o. Três pontos são limitantes no progresso da computação com multiprocessadores: custo, facilidade de aplicação e desempenho (RODGERS, 1985).

No que tange a custos, apartado do elevado custo de memória cache (tipicamente integrada direto na CPU), que reduz os problemas de conflitos e colisões no uso de memória entre os processadores; o crescimento da oferta dos elementos de hardware nos anos entre 2017 e 2019 reduziu os valores dos elementos computacionais que não se mostra mais um grande limitante para a computação com multiprocessadores.

No limitante de facilidade de aplicação é possível verificar que as práticas de design de sistemas de hardware e software que produzem um bom desempenho de processador monolítico quando aplicadas a multiprocessadores levam a gargalos de desempenho devido à largura de banda de memória e falta de recursos de serviços do sistema. O próprio sistema operacional gera algumas dependências de serialização, o que limita o desempenho da adição de múltiplos processadores, sendo o ponto mais importante a interação física do sistema de interconexão com o sistema de memória, já que os processadores compartilham estes recursos.

Uma das técnicas utilizadas para contornar o desafio do compartilhamento dos recursos de memória é a arquitetura de endereçamento virtual de memória, mas além de adicionar uma complexidade na gestão de tabelas de indexação de memória, cria uma latência adicional e consequente Overhead de gerenciamento do paralelismo (podendo variar de acordo com o tamanho do problema e número de núcleos utilizados) que culminando em uma redução do desempenho da aplicação [4] e [5].

Outra abordagem, ainda na parte de aplicação, é a definição de arquiteturas para uso da memória e do gerenciamento das aplicações:

Estrutura de conexão de memória por

- Multiprocessadores com barramento compartilhado no tempo
- Multiprocessadores com chave de barra cruzada
- Multiprocessadores com memória multiportas

Gestão das aplicações por

- Supervisor separado em cada processador
- Sistema operacional mestre-escravo
- Sistema operacional de supervisor flutuante

De todas as formas os sistemas de multiprocessadores compartilham (memoria, I/O, devices, interrupções de sistemas etc.) e recursos como memória e I/O são atribuídos dinamicamente a processos, e não permanentemente conectados aos processadores (RODGERS, 1985).

Nos estudos realizados por Amdahl (1967) ele já observava que a fração computacional responsável por gerenciamento de memória corresponde cerca de 40%; esse fato somado as instruções que são necessariamente sequencias, como I/O, geram um gargalo a todo o processamento e coloca um limite superior na aceleração

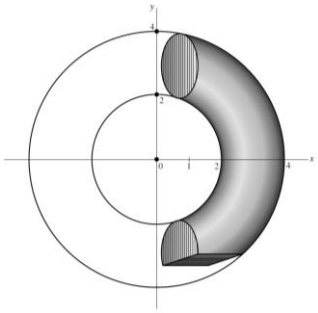
Aceleração $$S = \frac{1}{r_s + \frac{r_p}{n}} \quad (1)$$

Onde $r_s+r_p=1$ (2) e r_s representa a proporção da porção sequencial em um programa.

Levando (2) em (1) $$S = \frac{1}{(1 - r_p) + \frac{r_p}{n}} \quad (3)$$

2.1. Descrição do problema:

Deseja-se calcular o volume do Toroide seccionado, proposto pelo por Press et al. (2007), utilizado a metodologia de Monte Carlo implementada com computação paralela utilizando GPU, e avaliar a eficiência do paralelismo em cudablocks.



O objeto é definido pelas seguintes equações [3]:

Toroide $$z^2 + \left(\sqrt[2]{x^2 - x^2} - 3\right)^2 \leq 1 \quad (4)$$

Seccionamento $$x \geq 1 \text{ e } y \geq 1 \quad (5)$$

2.2. Algoritmo de Monte Carlo com GPU:

O algoritmo de integração numérica pelo método de Monte Carlo propõe sortear variáveis de sua equação ou ambiente, realizar a interação dessas variáveis sorteadas com o ambiente e/ou função que se deseja obter o resultado estatístico, anotar os resultados e repetir N vezes até que suas amostras sejam suficientes para representar o resultado. A característica essencial do processo é que evitamos lidar com múltiplas integrações e passamos a gerar amostras em uma distribuição uniforme.

Sua integração utilizando GPU é realizada pela divisão da tarefa de gerar múltiplos sorteios e cálculos entre os diversos cuda cores e, na sequência, centralizar as informações geradas na CPU que concluirá o cálculo da integral utilizando todos os dados.

Para realizar a utilização da GPU é necessário alocar memória no dispositivo(GPU) e realizar apontamentos dos valores para as rotinas em cuda.

3. Proposta de implementação

Pseudocódigo:

```
Aloca memória device cuda

integralMonteCarloCuda()
        inicializa a semente randômica
        loop no número de eventos
                gera um número aleatório x,y,z entre limites
                calcula f(x,y,z)
                calcula somatoriaFxyz
                calcula somatoriaFxyz ˆ2
        fim loop

        Se thread ==0
                calcula reduz threads em total
                retorna total

reduz cudablocks
calcula integral
```

4. Experimentação e Resultados

Foram executados 5.000.000.000 números aleatórios sendo distribuída a tarefa entre cuda cores paralelos a GPU da seguinte forma: 1 acrescido em 10 até 100 cuda blocks, de 100 acrescido de 1000 até 100 cuda blocks, e de 10000 acrescido de 10000 até 50000 cuda blocks, 100000 e 1000000 cuda blocks; como observação inicial do resultado comparado com a aplicação executada sem nenhuma técnica de paralelismo.

Ambiente:

Processoador model name Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz 48 cores

centos:centos:7

2 GPUs Tesla V100-PCIE 5120 Cuda Cores

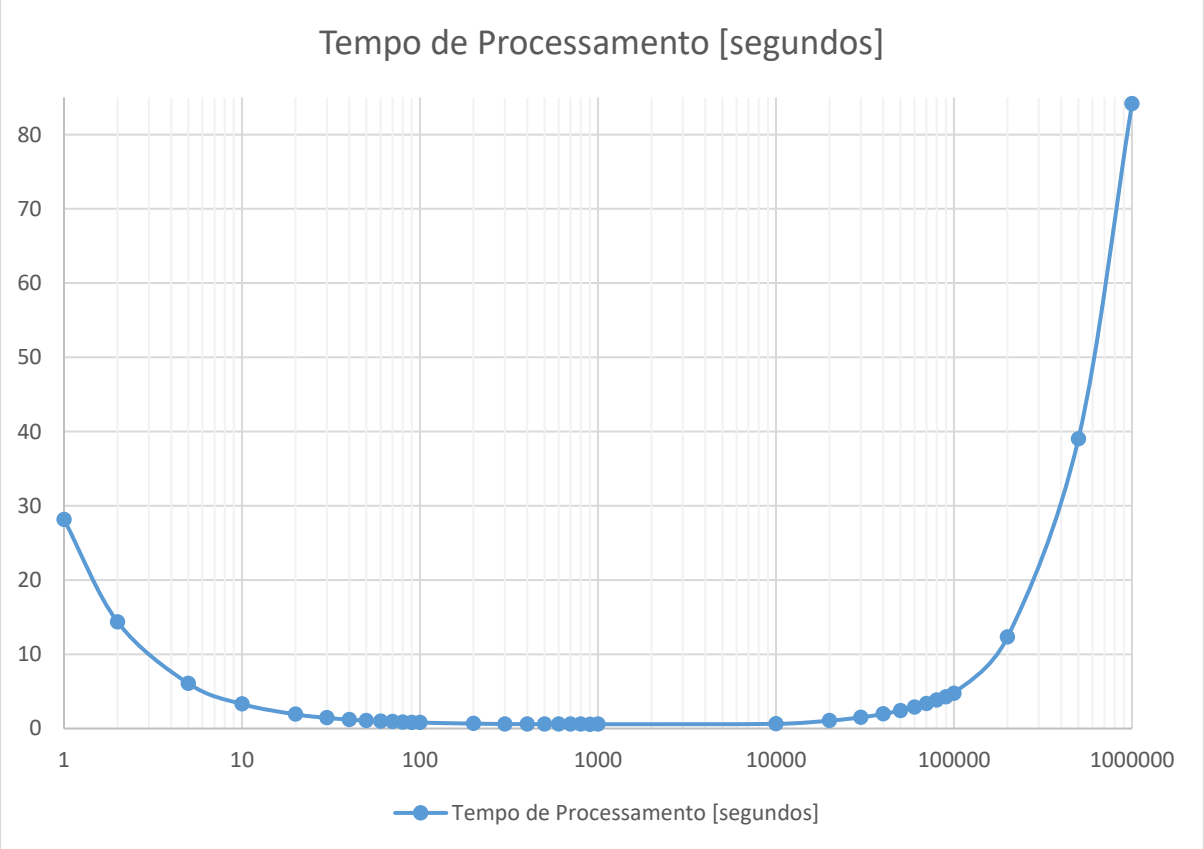
Resultado do cálculo do volume do Toroide pelo método de Monte Carlo

Monte Carlo	
N	Toroide
50.000.000.000	22.097343188760000032289099181070923805236816406 25

Tempo de execução por CudaBlocks

	5.000.000.000 Sorteios										
	Número da Execução - Resultado [segundos]										
Processos	1	2	3	4	5	6	7	8	9	10	Média
1	28.2	28.13	28.13	28.16	28.17	28.2	28.15	28.17	28.19	28.16	**28.166**
2	14.42	14.42	14.43	14.34	14.31	14.32	14.34	14.35	14.34	14.35	**14.362**
5	6.14	6.08	6.05	6.06	6.06	6.04	6.05	6.06	6.06	6.08	**6.068**
10	3.31	3.34	3.28	3.28	3.29	3.28	3.3	3.27	3.3	3.3	**3.295**
20	1.99	1.91	1.9	1.9	1.89	1.92	1.91	1.89	1.89	1.93	**1.913**
30	1.45	1.43	1.44	1.43	1.43	1.43	1.45	1.43	1.43	1.43	**1.435**
40	1.2	1.26	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	**1.206**
50	1.07	1.06	1.07	1.07	1.07	1.04	1.07	1.07	1.07	1.07	**1.066**
60	0.98	0.97	1.02	0.97	0.97	0.97	0.97	0.97	0.97	0.97	**0.976**
70	0.99	0.89	0.9	0.9	0.91	0.91	0.91	0.96	0.91	0.9	**0.918**
80	0.84	0.86	0.85	0.85	0.86	0.86	0.86	0.88	0.85	0.86	**0.857**
90	0.82	0.81	0.84	0.81	0.88	0.82	0.82	0.83	0.83	0.82	**0.828**
100	0.8	0.78	0.79	0.8	0.82	0.79	0.79	0.79	0.79	0.78	**0.793**
200	0.66	0.73	0.66	0.65	0.64	0.65	0.65	0.65	0.64	0.65	**0.658**
300	0.6	0.63	0.63	0.6	0.6	0.6	0.6	0.62	0.6	0.59	**0.607**
400	0.58	0.59	0.61	0.63	0.59	0.59	0.59	0.59	0.58	0.61	**0.596**
500	0.58	0.65	0.6	0.58	0.57	0.57	0.58	0.58	0.58	0.57	**0.586**
600	0.56	0.57	0.57	0.6	0.57	0.57	0.59	0.58	0.55	0.56	**0.572**
700	0.56	0.58	0.59	0.58	0.57	0.59	0.59	0.58	0.57	0.57	**0.578**
800	0.6	0.59	0.57	0.59	0.59	0.56	0.56	0.56	0.57	0.56	**0.575**
900	0.6	0.57	0.55	0.58	0.55	0.57	0.56	0.58	0.56	0.57	**0.569**
1000	0.61	0.61	0.56	0.58	0.58	0.56	0.57	0.58	0.56	0.57	**0.578**
10000	0.69	0.62	0.61	0.62	0.61	0.62	0.62	0.62	0.62	0.65	**0.628**
20000	1.05	1.05	1.05	1.07	1.05	1.09	1.06	1.06	1.07	1.05	**1.06**

30000	1.5	1.51	1.5	1.5	1.5	1.5	1.5	1.51	1.51	1.5	**1.503**
40000	1.97	1.97	1.96	1.97	1.95	1.98	1.96	1.97	1.95	1.97	**1.965**
50000	2.47	2.41	2.41	2.4	2.41	2.42	2.38	2.44	2.44	2.44	**2.422**
60000	2.85	2.96	2.88	2.88	2.88	2.86	2.86	2.88	2.86	2.89	**2.88**
70000	3.41	3.33	3.33	3.35	3.33	3.35	3.35	3.32	3.34	3.33	**3.344**
80000	3.91	3.82	3.79	3.79	3.81	3.79	3.82	3.8	3.81	3.81	**3.815**
90000	4.34	4.26	4.26	4.28	4.26	4.26	4.26	4.27	4.25	4.27	**4.271**
100000	4.87	4.75	4.73	4.72	4.75	4.73	4.75	4.73	4.74	4.76	**4.753**
200000	12.33	12.32	12.28	12.37	12.38	12.31	12.31	12.29	12.28	12.36	**12.32**
500000	39.03	39.03	39	38.96	39.04	38.95	39.03	38.97	38.97	38.97	**38.99**
1000000	84.04	84.9	84.06	84.05	84.05	84.09	84.09	84.13	84.1	84.09	**84.16**



5. Conclusão

O acréscimo de cuda blocks paralelos até o máximo do número de cuda cores da GPU acelera a execução da tarefa computacional, porém não proporcional à quantidade de cores adicionadas, é observada reduções do acréscimo de aceleração a cada processador adicionado em confirmação da lei de Amdahl, o que mostra que mesmo GPU respeita a lei de Amdahl.

Para acréscimos de cuda blocks acima do número de cuda cores da GPU observa-se que o comportamento é contrário, desacelerando a execução da tarefa computacional de forma exponencial com a quantidade de processos adicionados, o que confirma os custos de Overhead mensurados nos trabalhos de Höfinger e Haunschmid (2017) e Oliveira et al. (2018).

6. Referências

[1] AMDAHL, Gene M.. Validity of the single processor approach to achieving large scale computing capabilities. In: AFIPS SPRING JOINT COMPUTER CONF, 67., 1967, Atlantic City. **(Spring) Proceedings of the April 18-20**. Reston: Afips Press, 1967. v. 30, p. 483 - 485

[2] RODGERS, David P.. Improvements in Multiprocessor System Design. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 12., 1985, Boston. **ISCA '85 Proceedings**. Los Alamitos: Ieee Computer Society Press, 1985. p. 225 - 231.

[3] PRESS, William H. et al. **Numerical Recipes**: The Art of Scientific Computing. 3. ed. Cambridge, Massachusetts: Cambridge University Press, 2007. Cap. 7. p. 397-401

[4] HÖFINGER, Siegfried; HAUNSCHMID, Ernst. **Modelling parallel overhead from simple run-time records**. 2017. The Journal of Supercomputing. Disponível em: <https://doi.org/10.1007/s11227-017-2023-9>. Acesso em: 31 mar. 2017.

[5] OLIVEIRA, Victor H. F. et al. Application Speedup Characterization: Modeling Parallelization Overhead and Variations of Problem Size and Number of Cores. In: ACM/SPEC INTERNATIONAL CONFERENCE ON PERFORMANCE

ENGINEERING, 18., 2018, Berlin. **Proceeding ICPE '18 Companion of the 2018**. New York: Acm, 2018. p. 43 - 44