**Capítulo 4**

**4.1 – Introdução**

Uma outra maneira simples de se obter paralelismo utilizando apenas os núcleos de uma CPU é utilizando multi-processamento aberto (*Open Multi-Processing*, OpenMP), uma interface de programação paralela de memória compartilhada para arquitetura de múltiplos processadores [37]. Sendo assim, neste trabalho a computação paralela utilizando GPU e OpenMP é realizada para computar a multiplicação de matrizes e seus resultados são comparados com versão sequencial utilizando a CPU.

**4.2 – OpenMP**

OpenMP é uma especificação que fornece um modelo de programação paralela com compartilhamento de memória. Essa API é composta por um conjunto de diretivas que são adicionadas as linguagens C/C++ e Fortran [37], utilizando o conceito de *threads*, porém sem que o programador tenha que trabalhar diretamente com elas [39]. Esse conjunto de diretivas quando acionado e adequadamente configurado cria blocos de paralelização e distribui o processamento entre os núcleos disponíveis. O programador não necessita se preocupar em criar *threads* e dividir as tarefas manualmente no código fonte. O OpenMP se encarrega de fazer isso em alto nível.

O OpenMP não é uma linguagem de programação. Ele representa um padrão que define como os compiladores devem gerar códigos paralelos através da incorporação nos programas sequenciais de diretivas que indicam como o trabalho será dividido entre *cores*. Dessa forma, muitas aplicações podem tirar proveito desse padrão com pequenas modificações no código [37]. No OpenMP, a paralelização é realizada com múltiplos threadsdentro de um mesmo processo. Os threadssão responsáveis por dividir o processo em duas ou mais tarefas que poderão ser executadas simultaneamente. Diferente dos processos em que cada um possui seu próprio espaço de memória, cada *thread* compartilha o mesmo endereço de memória com os outros *threads* do mesmo processo, porém cada *thread* tem a sua própria pilha de execução [37].

O modelo de programação do OpenMP é conhecido por *fork-join*, onde um programa inicia com um único *thread* que executa sozinha todas as instruções até encontrar uma região paralela, que é identificada por uma diretiva OpenMP [38]. Ao chegar nessa região, um grupo de *threads* é alocado e juntas executam o código paralelizado. Ao finalizar a execução do paralelismo os *threads* são sincronizados e a partir desse ponto somente um *thread* é que segue com a execução do código sequencial. O *fork-join* pode ocorrer diversas vezes e é dependente do número de regiões paralelas que o programa possui [39].

**4.3 – OpenACC**

**4.4 – OpenCL**

Uma opção à plataforma é a OpenCL [40] que permite codificar tanto para GPUs NVIDIA quanto para ATI/AMD. O desenvolvimento de códigos em CUDA podem ser realizados utilizando C/C++ juntamente com alguns comandos específicos da plataforma. De maneira geral, programadores com alguma experiência em C/C++ não encontram dificuldades com a linguagem.

**4.5 – Ganho de eficiência no envio de dados**

A agilidade em um sistema é crucial no contexto da automatização de processos, podendo permitir grandes ganhos de eficiência, principalmente na implantação de sistemas complexos e que necessitem constantemente ser reconfigurados para se adaptar a novas exigências do mercado. Uma limitação no uso da metodologia destaca-se a impossibilidade de comunicação em tempo de execução no envio de dados entre o gêmeo digital e o simulador para o robô, tais processo podem ser paralelizados e com isso o ganho computacional viabiliza a utilização do gêmeo digital.

**4.6 – Onde aplicar a metodologia na usina hidroelétrica de Jirau**

Para obter ganho computacional nas aplicações da usina hidroelétrica de Jirau, e explorar o uso gêmeo digital em simulações e modelagem, utilizando os recursos disponíveis na GPU e empregando a arquitetura CUDA para paralelização de algumas etapas envolvidas na simulação e paralelização de processos do Gêmeo digital.

Para tirar o máximo de proveito dos processadores atuais, o software deve ser projetado para executar processos em paralelo. Essa heterogeneidade faz com que o processo de desenvolvimento eficiente de software apresente muitas ferramentas e arquiteturas, resultando em uma série de desafios para a comunidade da programação. Dentre esta gama de possibilidades usamos a linguagem CUDA [41].

Para avaliar o desempenho de cada API estamos realizando execuções de cada algoritmo. O tempo dessas execuções foram computados, sendo a média o valor utilizado, e desse tempo de execução obteve-se o intervalo de confiança das amostras, e também o *speedup* que cada algoritmo obteve em relação a execução sequencial de cada API.

**4.7 – Experimento**

Para iniciar a implementação utilizando GPU utilizamos uma função chamada Kernel. Essa função será executada dentro da GPU e utilizará os índices dos *threads* para tratar os dados. Definindo a quantidade de grids, blocos e o número total de threads que serão executados na GPU. Esses parâmetros dependem da arquitetura da placa, sendo assim a familiaridade e o conhecimento do programador sendo mais amplo tem-se um proveito maior da plataforma. O processo e o envio dos dados da memória da CPU para serem executados na memória global da GPU. Após a conclusão da execução da GPU o caminho inverso deve ser realizado.

Foi implementado algoritmo utilizando a linguagem C++, onde utilizou-se inicialmente apenas o processamento de CPU, posteriormente foi implementado o algoritmo utilizando a GPU, através do acesso API CUDA, estes códigos foram disponibilizados no github (https://github.com/bierley123/mestrado)*.*

Os experimentos foram executados em uma máquina com sistema operacional Linux, distribuição Xubuntu, com processador intel core i7, 2.5 GHz, 2 núcleos, 3 MB de memória cache e 6GB de memória RAM e uma placa gráfica NVIDIA Geforce 940M.

* A multiplicação será realizada considerando a implementação por ladrilho sequencial, paralelizada em OpenMP e em GPU.
* As abordagens serão aplicadas para precisão simples (float) e dupla (double).
* A multiplicação será executada em três diferentes configurações: 1 vez, 100 vezes, e 1000 vezes. As duas últimas paralelizadas com OpenMP.
* As matrizes e os ladrilhos serão ajustados para máximo desempenho.
* O desempenho será medido em termos de tempo de execução, *speedup*, através da lei de Amdahl [30], e em GFLOPS.

**4.7.1 – Ladrilhamento**

**4.7.2 – Multiplicação de matrizes na GPU**

A primeira abordagem para paralelização da multiplicação de matrizes utilizando uma GPU, após definir a matriz A e B, disparar diversas threads fazendo com que cada uma calcule um elemento da matriz resultante C.

A segunda abordagem de multiplicação de matrizes na GPU tem como objetivo utilizar a memória compartilhada de cada bloco de threads visando reduzir o número de acessos a memória global do dispositivo. Para isso as matrizes são subdivididas em pequenos blocos como mostrado na Figura 7. Essa técnica é conhecida como multiplicação por ladrilhamento e diferentemente da abordagem anterior, no qual toda linha de A e coluna B era multiplicada de uma só vez gerando assim um elemento de C, no ladrilhamento tem-se submatrizes de A e B que vão gerar um valor parcial de um elemento de C. Quando todos os ladrilhos forem processados o valor final de cada um dos elementos da matriz C será o somatório de cada elemento parcial obtido pela multiplicação de cada uma das linhas e colunas submatrizes A e B, respectivamente.

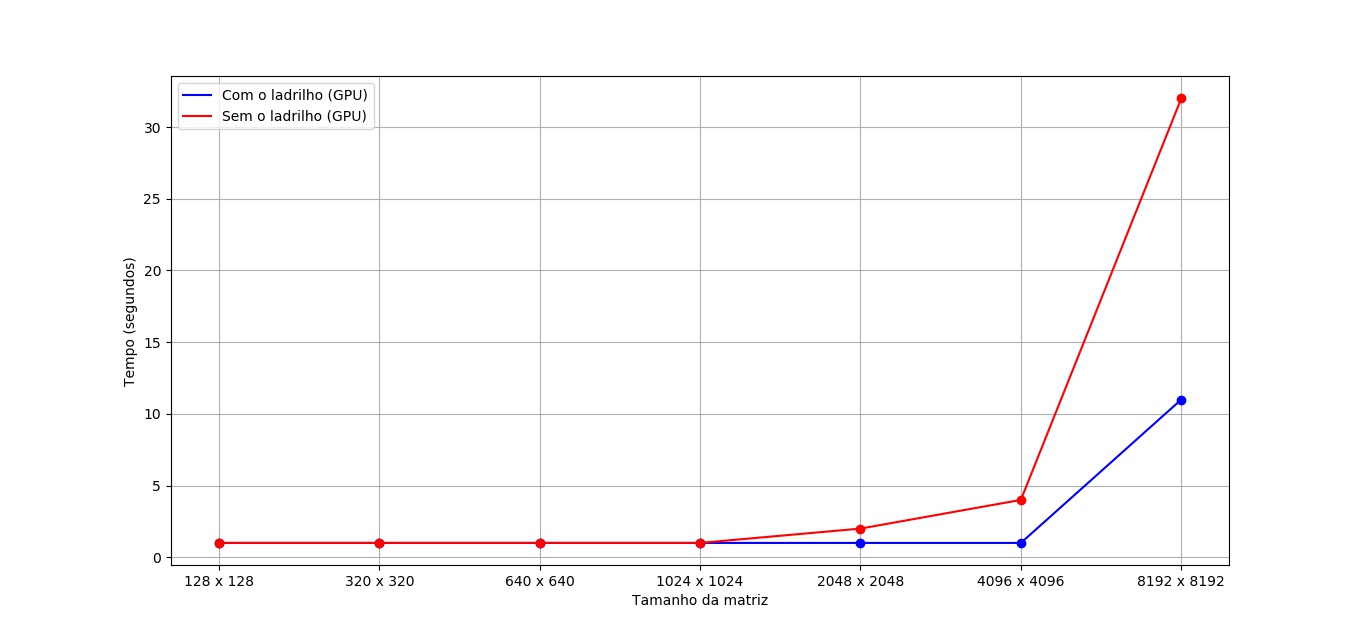


Figura 7. Multiplicação de matrizes com e sem ladrinho (GPU).

O objetivo de principal de se usar o ladrilhamento em GPU é carregar cada submatriz na memória compartilhada do bloco de threads. O caso ideal seria alocar toda matriz dentro de uma memória compartilhada. Porém, como já mencionado, essa memória é de tamanho reduzido quando comparada com a memória global. Dessa forma, a ideia é carregar na memória compartilhada apenas submatrizes de A e B, na qual cada bloco de threads pode compartilhar seus dados de maneira rápida. Com isso, um requisito de projeto é que o ladrilho caiba dentro de um bloco de threads da GPU. O valor do tamanho do bloco varia de acordo com o dispositivo e cabe ao programador conhecer a arquitetura para melhor aproveitá-la.

**4.7.3 – Multiplicação de matrizes na GPU com OpenMP**

Na CPU esses índices são obtidos por loops de controle. O objetivo principal do ladrilhamento também é o mesmo: tirar proveito da arquitetura de CPU através da memória cache de acesso rápido.

Para realizar paralelização por meio da CPU é utilizado a OpenMP. A API dá suporte as linguagens C/C++ e Fortran, e basicamente o que deve ser incluído ao código original são diretiva em trechos de códigos que devem ser paralelizados. Como mostrado na Figura 8.

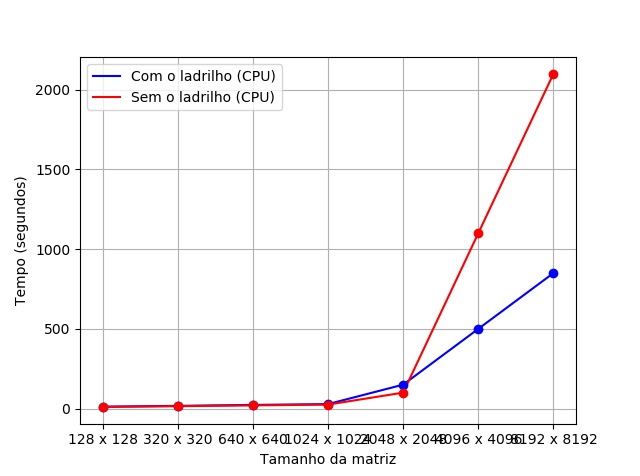


Figura 8. Multiplicação de matrizes com e sem ladrinho (CPU).

A simples anotação indica para paralelizar o loop na sequência, compartilhar as variáveis n, x e y e manter i privado. Dessa maneira simples já é possível obter ganho de desempenho utilizando os núcles da CPU. O número de threads disparadas como default é uma por núcleo do processador, todavia o programador pode alterar esse valor por meio da variável de ambiente OMP\_NUM\_THREAD.

**4.7.4 – Multiplicação para precisão simples**

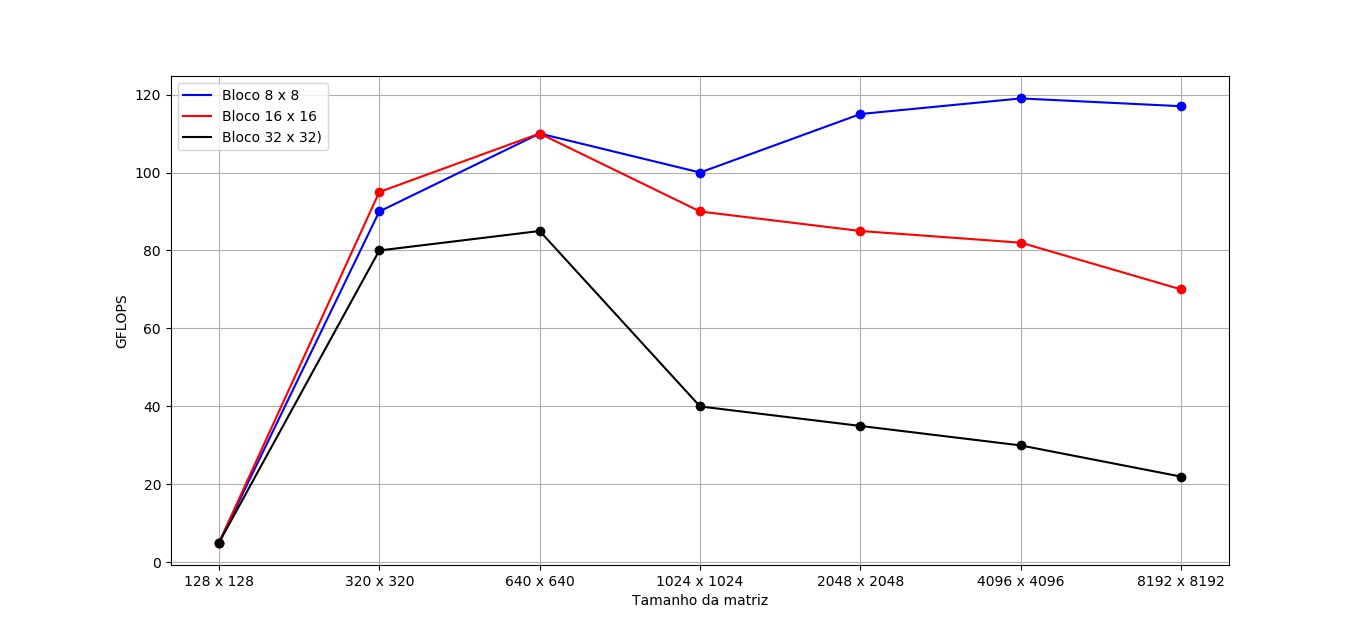
Para obter eficiência máxima, as matrizes escolhidas possuem número de linhas e colunas divisíveis pelo tamanho máximo de ladrilho, ou seja, um bloco de 32 × 32. Dessa forma um grid de blocos se encaixa perfeitamente nas dimensões da matriz, sem necessidade de verificações na função de kernel da GPU, o que ocorreria se o ladrilho não fosse divisível. Isso não é uma limitação da GPU, é simplesmente um artifício para verificar performance máxima. Com esses valores tanto para precisão simples, quanto para precisão dupla, as matrizes cabem dentro da memória global. Além disso, como a CPU utilizada possui apenas dois núcleos, o número de threads disparadas será igual a quatro, duas por núcleo. Como mostrado na Figura 9.

Figura 9. Processamento em Blocos de tamanho 8, 16 e 32.

**4.7.5 – CPU × CPU + OpenMP**

A partir da multiplicação das matrizes de ordem 320×320 já se obtém um pequeno ganho de processamento. De maneira geral a paralelização via OpenMP obtém um ganho em relação a sequencial quando as matrizes multiplicadas alcançam a ordem de 1024 × 1024. Devido a facilidade de se incluir a paralelização, ainda assim é recomendável o uso da API para matrizes grandes.

**4.7.6 – CPU + OpenMP × GPU**

A partir desta configuração a proporção dispara para mais de 200x, deixando claro que a GPU efetua muito mais cálculos por segundo do que a CPU e a CPU+OpenMP. De maneira geral, baseado nos gráficos apresentados nas Figuras 11 a 16, é possível notar a diferença de desempenho da GPU para CPU e CPU+OpenMP. A capacidade de processamento é extremamente maior sendo muito vantajoso seu uso, principalmente quando a multiplicação de matrizes atinge a ordem de 1024 × 1024, no qual o crescimento da curva de tempo de execução para CPU e CPU+OpenMP é muito alto

**4.7.7 – Multiplicação para precisão dupla.**

Nesta segunda parte de experimentos serão analisados os resultados para matrizes de precisão dupla (double).

**4.7.8 – CPU × CPU + OpenMP**

O processamento é bem próximo ao da precisão simples. A diferença de tempo de execução também segue a mesma linha, sendo perceptível a partir da ordem 640 × 640, como mostrado na Figura 17(b).

**4.7.9 – GPU × CPU + OpenMP**

A diferença de tempo de execução, mostrado no gráfico da Figura 20(b), ainda é bem discrepante quando as ordens das matrizes multiplicadas aumentam. O *speedup*, mostrado na Figura 20(c), também diminui bastante em relação a precisão simples, porém ainda é possível observar o ganho em relação a GPU.

**4.7.10 – Comparação entre precisão simples e dupla**

Na comparação de speedup em relação a CPU, ilustrada na Figura 23(c), é possível perceber que na medida que as ordens das matrizes aumentam, o *speedup* da precisão simples se distancia da precisão dupla. O processamento da precisão simples cresce substancialmente quando comparada com a dupla.

A Tabela 2 mostra o cada tamanho de cada matriz utilizada e seus respectivos tempos. As mesmas matrizes são processadas tanto em programação tradicional usando CPU quanto programação paralela usando GPU (utiliza API CUDA).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tamanho Matriz | Linguagem C++  Tradicional | Linguagem C++  com Gpu | Diferença  Tradicional - Paralelo | Percentual (%) |
| 100 x 100 | 0.1274 | 0.0233 | 0.1040 | 44,35 |
| 500 x 500 | 4.8812 | 1.1866 | 3.6946 | 311,36 |
| 1000 x 1000 | 23.7649 | 7.5387 | 16.2262 | 215,24 |
| 1500 x 1500 | 59.0981 | 23.8972 | 35.2009 | 147,30 |
| 2000 x 2000 | 120.9714 | 52.7784 | 68.1929 | 129,20 |
| 2500 x 2500 | 182.7194 | 97.0769 | 85.6424 | 88,22 |
| 3000 x 3000 | 288.8846 | 176.8720 | 112.0125 | 63,33 |
| 3500 x 3500 | 441.4162 | 266.1067 | 175.3094 | 65,88 |
| 4000 x 4000 | 619.8696 | 408.5240 | 211.3455 | 51,73 |
| 4500 x 4500 | 831.1901 | 560.6376 | 270.5524 | 48,26 |
| 5000 x 5000 | 1111.0693 | 761.9882 | 349.0811 | 45,81 |
| 5500 x 5500 | 1411.9509 | 908.8802 | 503.0707 | 55,35 |
| 6000 x 6000 | 1809.7701 | 1426.7386 | 383.0315 | 26,85 |
| 6500 x 6500 | 2146.5558 | 1642.3178 | 504.2379 | 30,70 |
| 7000 x 7000 | 2559.5322 | 1789.0869 | 770.4452 | 43,06 |
| 7500 x 7500 | 3040.4100 | 2208.0520 | 832.3579 | 37,70 |
| 8000 x 8000 | 6632.7703 | 3265.0333 | 3367.7370 | 103,14 |
| **Média** | **1252.0577** | **799.8081** |  | **56,54** |

**Tabela 2:** Tempo processamento CPU x GPU em segundos.

Na Figura 6 mostra a curva do custo computacional, com o respectivo comportamento e *speedup* de cada uso da linguagem tradicional e paralela.

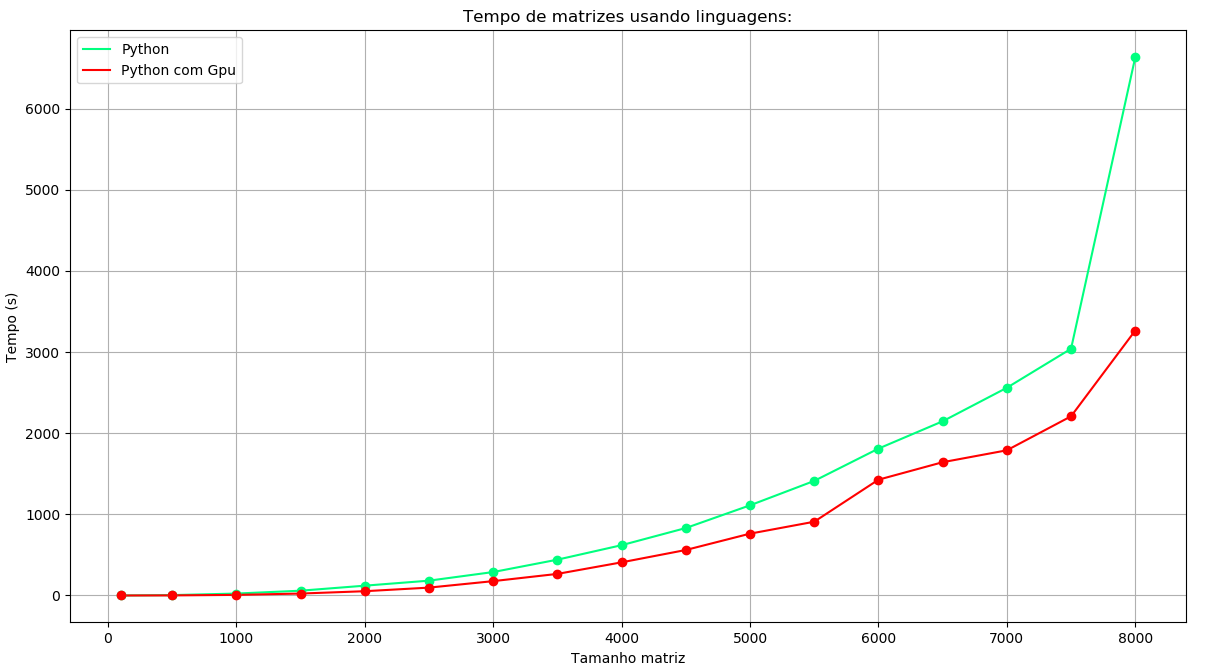
****

Figura 10. Custo computacional do algoritmo.

Utilizando as funções CUDA para realizar as operações na geração das matrizes, o tempo médio do cálculo destas matrizes foi reduzido 56,54% vezes mais rápido. Os resultados mostraram que mesmo realizando uma pequena parte do processamento na GPU a combinação da linguagem interpretada com o processamento na GPU oferece uma alternativa bastante conveniente para o desenvolvimento de algoritmos numéricos.