**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 [34]. 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 [34], utilizando o conceito de *threads*, porém sem que o programador tenha que trabalhar diretamente com elas [36]. 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 os *cores*. Dessa forma, muitas aplicações podem tirar proveito desse padrão com pequenas modificações no código [37]. No OpenMP, a paralelização é realizada com múltiplas *threads* dentro de um mesmo processo. As *threads* são 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 as outras *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 uma única *thread* que executa sozinha todas as instruções até encontrar uma região paralela, que é identificada por uma diretiva OpenMP [34]. Ao chegar nessa região, um grupo de *threads* é alocado e juntas executam o código paralelizado. Ao finalizar a execução do paralelismo as *threads* são sincronizadas e a partir desse ponto somente uma *thread* (inicial) 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 [35].

Foi implementado algoritmo utilizando a linguagem Python, onde utilizou-se inicialmente apenas o processamento de CPU, posteriormente foi implementado o algoritmo utilizando a GPU, através da biblioteca *Numba* (utiliza API CUDA)em Python, estes códigos foram disponibilizados no github (https://github.com/bierley123/pythonapicuda/blob/master/PythonCuda.py)*.*

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).

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tamanho Matriz | Linguagem Python  tradicional | Linguagem Python  com Gpu | Diferença  Tradicional - Paralelo | Percentual  % |
| 100 x 100 | 0.1274585723876953 | 0.02337193489074707 | 0.10408663749694824 | 44,35 |
| 500 x 500 | 4.8812413215637210 | 1.18661522865295400 | 3.69462609291076660 | 311,36 |
| 1000 x 1000 | 23.7649521827697750 | 7.53870677947998050 | 16.22624540328979500 | 215,24 |
| 1500 x 1500 | 59.0981595516204800 | 23.89720773696899400 | 35.20095181465149000 | 147,30 |
| 2000 x 2000 | 120.9714651107788100 | 52.77848839759827000 | 68.19297671318054000 | 129,20 |
| 2500 x 2500 | 182.7194447517395000 | 97.07695484161377000 | 85.64248991012573000 | 88,22 |
| 3000 x 3000 | 288.8846502304077000 | 176.87205576896667000 | 112.01259446144104000 | 63,33 |
| 3500 x 3500 | 441.4162542819977000 | 266.10677719116210000 | 175.30947709083557000 | 65,88 |
| 4000 x 4000 | 619.8696622848511000 | 408.52408576011660000 | 211.34557652473450000 | 51,73 |
| 4500 x 4500 | 831.1901185512543000 | 560.63767409324650000 | 270.55244445800780000 | 48,26 |
| 5000 x 5000 | 1111.0693798065186000 | 761.98827505111700000 | 349.08110475540160000 | 45,81 |
| 5500 x 5500 | 1411.9509959220886000 | 908.88023877143860000 | 503.07075715065000000 | 55,35 |
| 6000 x 6000 | 1809.7701418399810000 | 1426.73860120077332000 | 383.03154063224790000 | 26,85 |
| 6500 x 6500 | 2146.5558035373690000 | 1642.31782841682430000 | 504.23797512054443000 | 30,70 |
| 7000 x 7000 | 2559.5322494506836000 | 1789.08699560165400000 | 770.44525384902950000 | 43,06 |
| 7500 x 7500 | 3040.4100060462950000 | 2208.05205607414250000 | 832.35794997215270000 | 37,70 |
| 8000 x 8000 | 6632.7703993320465000 | 3265.03331494331000004 | 3367.73708438873250000 | 103,14 |
| **Média** | **1252.0577872220208000** | **799.8081910469953** |  | **56,54** |

No gráfico 1 temos a curva do custo computacional, com o respectivo comportamento e speedup de cada uso da linguagem tradicional e paralela.

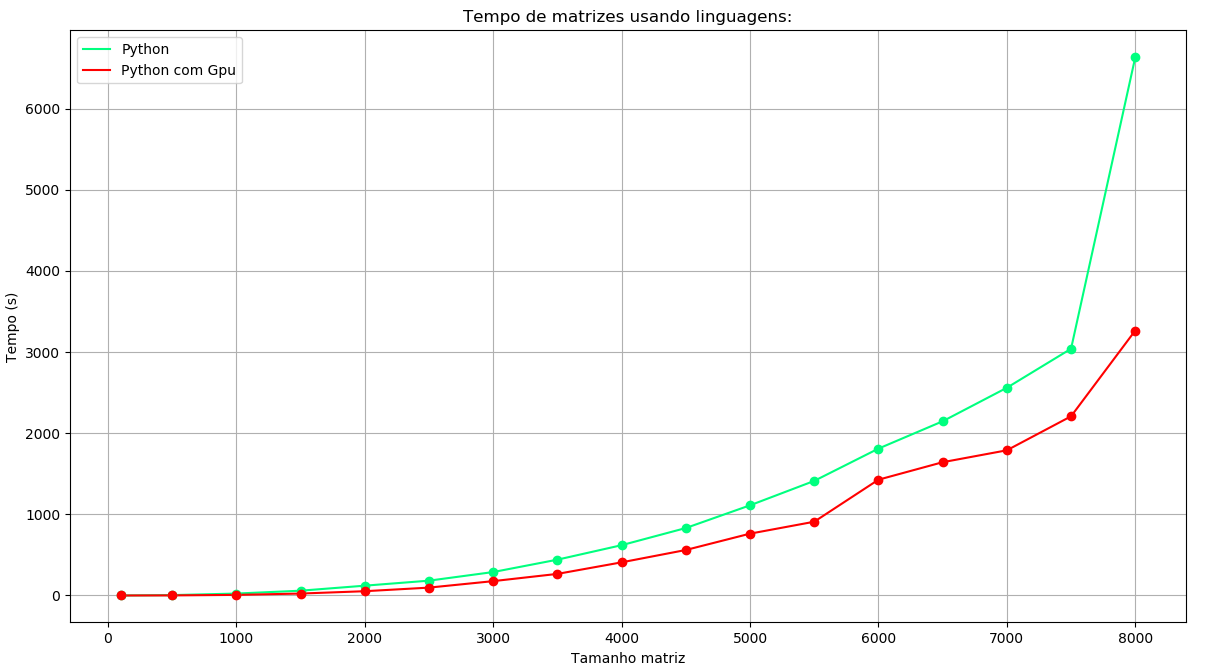
****

Figura 3. 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.

As principais memórias de uma GPU são descritas a seguir:

• Memória global: é a memória principal da GPU. Pode ser acessada por todas as threads/cores, porém possui alta latência e baixo throughput.

• Memória compartilhada: como já mencionado, é a memória dedicada de cada SM que possui baixa latência. Somente threads de um mesmo bloco pode acessá-la.

• Memória local: possui este nome pois é a memória específica de uma thread.

Das memórias descritas, a CPU tem acesso somente a memória global. A distribuição de threads e hierárquia de memória é ilustrada pela Figura 3

Uma opção à plataforma é a OpenCL (10) 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.

10. AMD. OpenCL Zone – Accelerated Parallel Processing. AMD Corporation, 2016. Disponível em: <http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OCL_Programming_Guide-2013-06-21.pdf/>. Acesso em 25 de novembro de 2019

Implementar uma função especial chamada Kernel. Essa função será executada dentro da GPU e utilizará os índices das threads para operar nos dados.

• Definir a quantidade de grids e blocos definindo assim o número total de threads que serão executados na GPU. Esses parâmetro dependem da arquitetura da placa e quanto mais o programador é familiar com a mesma, mais proveito ele tira da plataforma.

• Enviar os dados a serem executados da memória da CPU para memória global da GPU. Após a conclusão da execução da GPU o caminho inverso deve ser realizado.

4.1 Introdução

Em sistemas de manufatura e automação, os componentes (controladores, máquinas, supervisórios, etc.) e usuários do sistema podem estar conectados de diversas formas, estes dispositivos sendo atrelado ao gêmeo digital podem ser uma máquina, produto, ferramenta, robô, veículo, etc. Aplicações externas, que utilizam ou são utilizadas pelo gêmeo digital, podendo ser equipamentos externos, softwares ou até mesmo outros gêmeos digitais. O gêmeo digital desconectado não possui uma conexão direta com o dispositivo, seja porque este não possui capacidade de comunicação ou porque não é relevante ou exequível implementar tal conexão. Por exemplo, em (SIERLA et al., 2018) é proposto um gêmeo digital para simulação do processo de manufatura de um dispositivo, sendo que nenhuma informação tem origem no próprio dispositivo.

No âmbito da automação industrial, muitos equipamentos implementam algum tipo de comunicação usando protocolos industriais, possibilitando assim, ao menos em princípio, a implantação do gêmeo digital no modo conectado. Outras tecnologias que podem ser citadas como possíveis bases para implantação são frameworks para internet das coisas (por exemplo, Fiware1) e sistemas SCADA.

Por exemplo, quando um sensor é ativado no domínio físico, os valores são comunicados ao gêmeo digital através de uma interface de comunicação, que então altera o atributo apropriado do sensor na hierarquia da instância. Alterações estruturais na hierarquia da instância também podem ser feitas em tempo de execução, permitindo por exemplo, que o modelo de uma máquina seja expandido de forma dinâmica ao se adicionar uma ferramenta a ela.

Estas informações são especialmente importantes no âmbito da Indústria 4.0, onde sistemas heterogêneos precisam se adaptar e responder a novas demandas em cenários não vistos de acordo com as suas características. Neste contexto, uma aplicação inteligente, por exemplo, pode ter conhecimento através do modelo sobre o tipo de cada componente, e, se necessário, adaptar o seu comportamento de acordo com estas informações.

4.2. Lacuna a ser explorada

Após uma profunda visão geral da lógica de funcionamento e das ferramentas utilizadas para a implementação de componentes do gêmeo digital apresentados neste trabalho, percebemos que a agilidade na implantação de um sistema é crucial no contexto da Industria 4.0, sendo que a automatização deste processo pode 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.

4.3. Avaliar o desempenho da programação paralela

Existem pesquisas que avaliam o desempenho das principais APIs de programação paralela, através da execução de algoritmos em CPU e GPU. Para que a avaliação de desempenho fosse possível, primeiramente foram instaladas em um computador as APIs CUDA (GPU), OpenCL (GPU), OpenACC (GPU) e OpenMP (CPU) (CORRÊA; SULZBACH, 2017).

Foram selecionados para execução em cada API os algoritmos SRAD V1, HotSpot e PathFinder, que fazem parte do benchmark Rodinia. Os speedups dos testes mostraram que a GPU obteve um desempenho superior em relação a CPU na maioria dos algoritmos. Acredita-se que esse resultado ocorreu pelo fato da GPU possuir mais núcleos se comparada a CPU, apesar da frequência dos núcleos ser inferior) (CORRÊA; SULZBACH, 2017). Os resultados do artigo mostraram que os processos paralelos têm maior ganho computacional conforme a complexidade dos algoritmos ou mesmo do processo computacional. Veja Figura 22, temos o algoritmo SRAD V1 que tem como objetivo reduzir pequenas manchas com difusão anisotrópica é um método de difusão para aplicações de imagem de ultrassom e radar, baseados em equações diferenciais parciais (PDE).



Figura 22 – Speedup do algoritmo SRAD V1 (CORRÊA; SULZBACH, 2017)

Na Veja Figura 23, temos o algoritmo HotSpot que é uma ferramenta amplamente utilizada para estimar a temperatura do processador baseado em uma planta em arquitetura e medições de potência simulados.



Figura 23 – Speedup do algoritmo HotSpot (CORRÊA; SULZBACH, 2017)

Na Veja Figura 24, temos o algoritmo PathFinder que é um algoritmo utilizado para encontrar a rota mais curta entre dois pontos. Utiliza programação dinâmica para encontrar um caminho em uma grade 2-D, a partir da linha de fundo para a fila superior com os menores pesos acumulados, onde cada passo do caminho se move para frente ou diagonalmente em frente.



Figura 24 – *Speedup* do algoritmo PathFinder (CORRÊA; SULZBACH, 2017)

O artigo conclui que os resultados, com exceção do algoritmo HotSpot, que a execução na GPU através de (CUDA, OpenCL, OpenACC) comprovam a teoria de que quanto mais núcleos se tem a disposição para paralelizar o programa, melhor será o desempenho. Por fim ao executar todos esses procedimentos, computados e comparados, conclui-se que a programação paralela é melhor para implementar diversos tipos de sistemas complexos e pode trazer diversos benefícios aos sistemas tradicionais.

4.4. Onde aplicar a metodologia na usina hidroelétrica de Jirau

A metodologia proposta permite criar um modelo digital desde um simples produto manufaturado, passando pelos equipamentos industriais e chegando no nível de digitalizar a fábrica inteira. Nas aplicações da usina hidroelétrica de Jirau podemos usar gêmeo digital para uso de simulações e modelagem. Explorando o uso de uma GPU 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 (KIRK and HUW, 2010). 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 (GASTER al., 2012). Dentre esta gama de possibilidades usamos a linguagem CUDA.

CUDA é uma plataforma de computação paralela e um modelo de programação criados pela NVIDIA em 2006. Seu objetivo é possibilitar ganhos significativos de desempenho computacional aproveitando os recursos das unidades de processamento gráfico (GPU). Através da API CUDA pode-se enviar código C, C++ e Fortran diretamente à GPU, sem necessitar de uma nova linguagem de compilação (NVIDIA, 2015b). A tecnologia CUDA é de abordagem proprietária, concebida para permitir acesso direto ao hardware gráfico específico da NVIDIA.

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.

Dessa maneira é fácil observar que multiplicar matrizes é um bom exemplo de computação paralela. Cada elemento de C é computado de maneira independente, logo, pode ser paralelizado.

**3. Multiplicação de matrizes**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**A. Multiplicando matrizes na GPU.** A primeira abordagem para paralelização da multiplicação de matrizes utilizando uma GPU é disparar diversas threads fazendo com que cada uma calcule um elemento da matriz resultante C.

Com isso o algoritmo faz muitos acessos à m emória global, que possui alta latência e baixo throughput.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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 6 (3). 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.

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.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**B. Multiplicando matrizes na CPU 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 paralelizado.

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.

**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.

Que fique claro que 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.

**B.1. CPU × CPU + OpenMP**

a partir da multiplicação das matrizes de ordem 320×320 já se obtém um pequeno ganho de 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.

**B.2. 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.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Multiplicação para precisão dupla.**

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

**C.1. 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).

**C.2. GPU × CPU + OpenMP.**

A diferença de tempo de execução, mostrado no gráfico da Figura 20(b), ainda é bem discrepante quando a ordem 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.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**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 a ordem 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.

**4. Experimentos**

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.

Passos a seguir:

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 (11), e em GFLOPS

11. John L Hennessy and David A Patterson. Computer architecture: a quantitative approach, chapter 8, pages 29–32. Elsevier, 2 edition, 2011.

Pode ser observado que na medida que as matrizes aumentam, as configurações de blocos 8 × 8 e 16 × 16 diminuem a capacidade de cálculo em relação ao tamanho 32 × 32.