Universidade Federal de São Carlos
Departamento de Computação

Relatório 6 - Arquitetura de Alto Desempenho

Conceitos de GPUs

Alunos: **Lucas Mantovani Gomes**

Jonas Gabriel dos Santos Costa Fagundes

Lucas Gabriel Valenti

Professor: **Emerson Carlos Pedrini**

São Carlos, 23 de julho de 2025

Resumo

O presente relatório descreve o que foi desenvolvido na segunda atividade experimental da disciplina de arquitetura de alto desempenho. Na qual, o objetivo era estudar e explorar o potencial de paralelismo massivo presente nas unidades de processamento gráfico (GPUs). Na atividade prática implementamos uma função de mendelbrot no matlab, de forma serial e vetorizada utilizando diferentes funções de paralelização da ferramenta matlab, para mensurar o speed up e visualizar os fractais formados em um plano complexo.

1 História das GPUs

Ao contrário do que se imagina, as GPUs não surgiram a partir de processadores, mas evoluíram simultaneamente a estes. Quando os engenheiros perceberam que continuar aumentando o número de núcleos de um processador, traria mais custo e complexidade do que ganhos de desempenho, começaram a investir em extensões AVX, que teve relativo sucesso em aplicações multimídia. No entanto, por volta do final dos anos 90, os engenheiros da NVIDIA começaram a pensar em uma solução melhor do que CPUs para determinados problemas, como processamento gráfico. Surgiram então os primeiros aceleradores gráficos 3D não-programáveis - especializados em transformações geométricas, iluminação, texturas e rasterização. Logo mais, no início dos anos 2000, surgiram as primeiras General Purpose GPU (GPGPU), com as primeiras tecnologias de shaders para manipular vértices e texturas, e sombreamento de pixels.

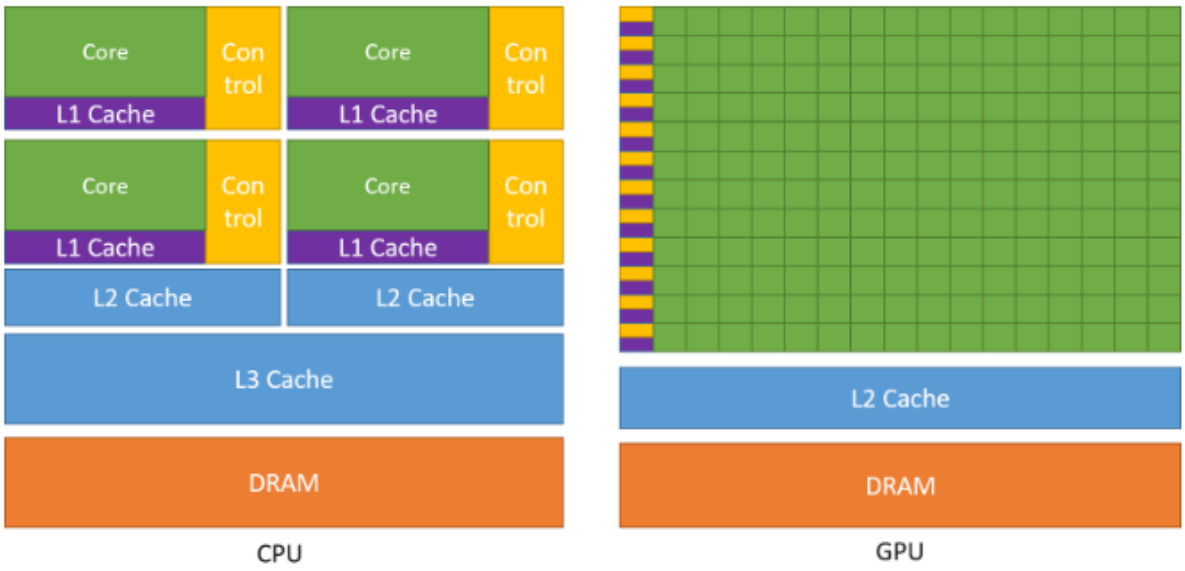

Figura 1: Diferença arquitetura CPU x GPU - (NVIDIA CORPORATION, 2025)

Em 2006 sugiu a GeForce 8800 com processamento paralelo otimizado através de uma arquitetura de shaders unificados e reprogramáveis, introduzindo pela primeira vez o CUDA. Uma abordagem baseada em C, que permite programar GPUs para um determinado propósito, como se fossem processadores multicore.

Para adaptar essa tecnologia para HPC em vez de processamento gráfico 3D, a NVIDIA lançou a linha Tesla. Mais barata e muitas vezes mais rápida que os processadores da

época. O que rápidamente tornou as GPUs muito mais atrativas para computação de alto desempenho, do que supercomputadores de empresas como Cray ou Fujitsu.

Impulsionada pelo crescente mercado de jogos, e pela acessibilidade para computação de alto desempenho, aumenta cada vez mais a quantidade de desenvolvedores CUDA, que nunca sequer tiveram contato com um Cray.

Em 2010, a NVIDIA lançou a arquitetura Fermi, com grandes avanços de desempenho, eficiência, paralelismo e produtividade de programação dos kernels, ao apresentar melhorias consideráveis na integração de CUDA com linguagem C, e a tecnologia de Streaming Multiprocessors(SM).

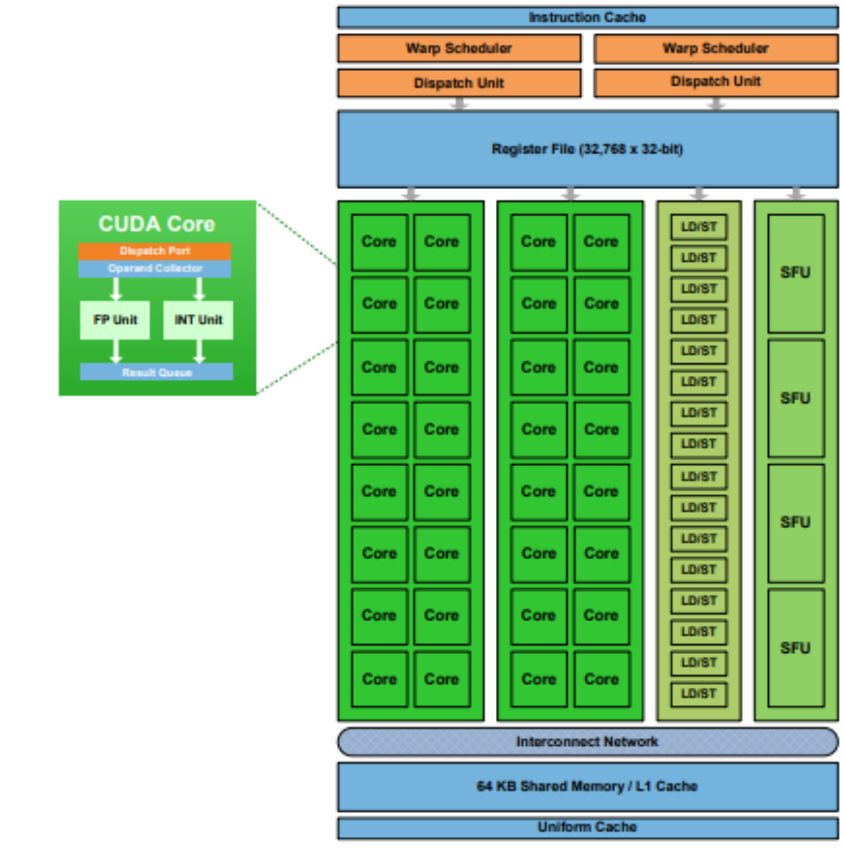

Figure 6. Each Fermi SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of configurable RAM, and thread control logic. Each core has both floating-point and integer execution units. (Source: NVIDIA)

Figura 2: Fermi SM - (GLASKOWSKY, 2010)

Dentro de cada SM, blocos de threads são divididos em warps de 32 threads e despachados para 32 núcleos CUDA. SMs também possuem Load/Store units para acelerar essas operações com caminhos críticos muito longos, e 4 Special Function Units, para

operações como Seno, Cosseno e Exponencial. Além de 64kb de memória RAM reconfigurável entre memória compartilhada entre todas as threads ou memória local L1 cache. Um total de 32 instruções podem ser despachadas simultaneamente por cada SM do nosso exemplo.

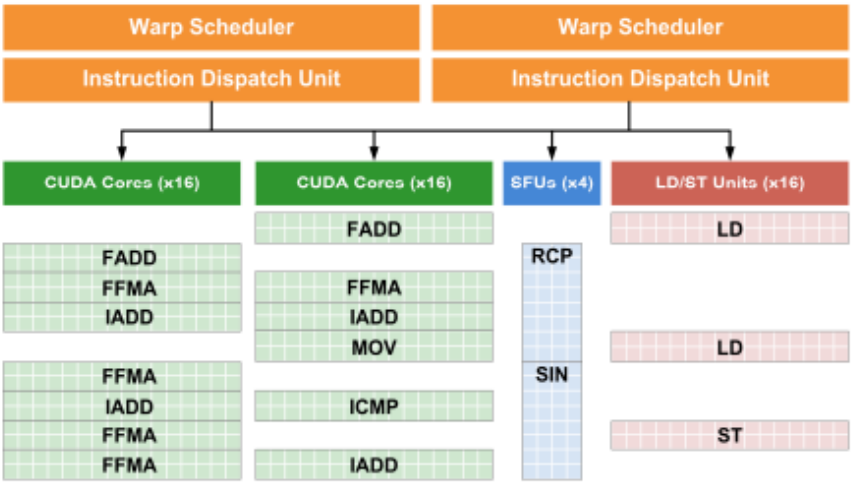

Figure 7. A total of 32 instructions from one or two warps can be dispatched in each cycle to any two of the four execution blocks within a Fermi SM: two blocks of 16 cores each, one block of four Special Function Units, and one block of 16 load/store units. This figure shows how instructions are issued to the execution blocks. (Source: NVIDIA)

Figura 3: Warp Scheduler - (GLASKOWSKY, 2010)

A arquitetura Fermi foi um divisor de águas ao introduzir recursos como memória cache unificada, suporte a ECC (Error Correction Code) e melhorias em controle de threads, o que tornou as GPUs ainda mais confiáveis e atraentes para aplicações científicas, simulações físicas, inteligência artificial e big data.

Nos anos seguintes, arquiteturas como Kepler, Maxwell, Pascal, Volta, Turing, Ampere e Hopper consolidaram ainda mais o papel das GPUs como pilares da computação paralela moderna. Cada geração trouxe avanços em eficiência energética, número de núcleos CUDA, interconexões de alta velocidade (como NVLink) e suporte a tecnologias emergentes como ray tracing em tempo real e Tensor Cores para aprendizado de máquina.

Hoje, as GPUs estão no centro de diversas revoluções tecnológicas — desde a renderização de gráficos hiper-realistas em tempo real até o treinamento de modelos massivos de inteligência artificial. A transição de aceleradores gráficos fixos para plataformas de computação programáveis demonstrou como uma arquitetura voltada para paralelismo massivo pode redefinir paradigmas de desempenho e escalabilidade.

Em resumo, a história das GPUs é marcada por uma transformação profunda: de simples controladores gráficos a motores de computação de propósito geral. Uma trajetória que continua em evolução, impulsionada pela demanda por mais desempenho, mais paralelismo e mais inteligência computacional.

2 Exercícios para Avaliação

2.1 tipos e características de memórias encontradas em GPUs;

As GPUs assim como as CPUs, típicamente possuem seus próprios espaços de memória DRAM, logo, é necessário alocar espaço na DRAM da GPU e trazer os dados do **host(CPU)**, para o **device(GPU)**, para que ele possa ser processado. Ao final do processamento, o conjunto de dados deve ser movido novamente da memória do device, para a memória do host.

Para tornar o acesso aos dados mais eficiente, existe uma hierarquia de memórias, na qual podemos citar os seguintes tipos: Possuem uma **memória global (memória de dispositivo)** de acesso lento e irrestrito, ou seja, compartilhada entre todas as threads e blocos de threads, onde são carregados os dados da CPU para a GPU. Porém as consultas nesta memória global (DRAM) são lentas, logo é necessário outro nível de memória mais rápida (cache), chamada **Parallel Data Cache (PDC)** e é compartilhada entre todas as threads de um mesmo bloco de cache - sua função é similar ao L1 Cache de uma CPU. Essa PDC ou **memória compartilhada**, pode ser acessada para operações atômicas por outros blocos de threads, formando uma **memória compartilhada distribuída**.

A PDC é tão rápida quanto os **registradores**, que são componentes de memórias privados de cada thread individual. As threadas também possuem um tipo de memória privada, chamada de **memória local**, um intervalo de endereços virtuais da memória global que elas podem utilizar para armazenar tipos de dados específicos que seriam muito grandes para armazenar em registradores. Também possuem, uma **memória constante**, uma ROM de baixa latência e alta largura de banda, que suporta múltiplos acessos simultâneos de diferentes threads.

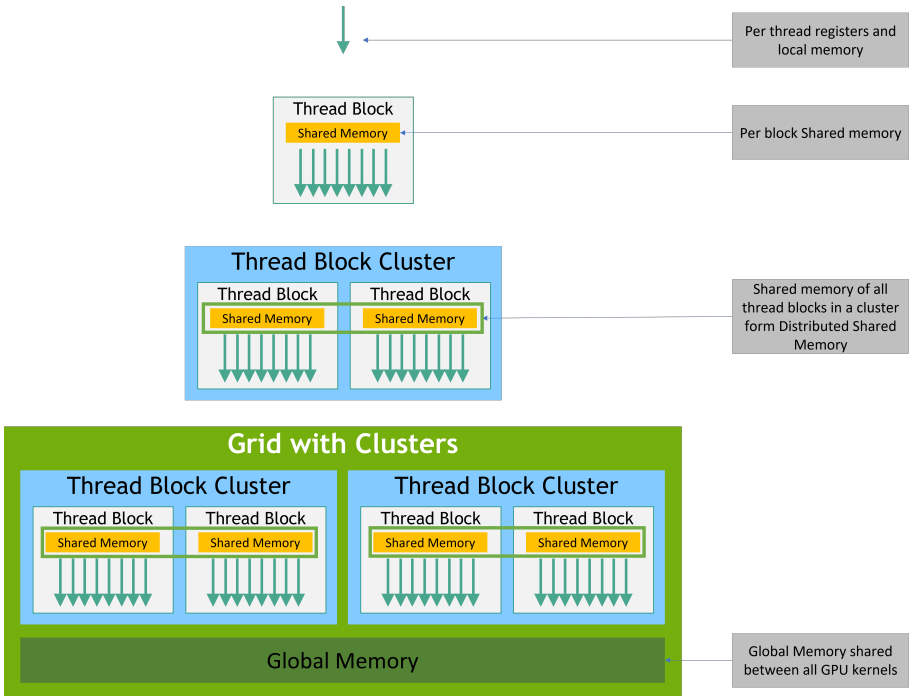

Figura 4: Hierarquia de Memórias de uma GPU - (NVIDIA CORPORATION, 2025)

2.2 Compare essas memórias com os tipos utilizados por CPUs

Nas CPUs temos tipos de memória semelhantes mas organizados em uma hierarquia diferente: Registradores nos núcleos da CPU, muito rápidos porém muito pequenos(KBs), feitos apenas para armazenar pequenos elementos que serão computados nas instruções. Em seguida temos 3 níveis de memória cache (L1, L2 e L3) cada qual mais lento e com maior capacidade que o anterior, funcionam para os núcleos do processador, de maneira similar à memória local das threads de uma GPU. Abaixo destes, existe a Memória Principal (RAM ou DRAM) muito mais lenta, porém muito maior (GBs), exerce a mesma função que a memória global das GPUs. Quando esta se esgota é utilizada a memória virtual, uma parte da memória secundária que é mapeada para acesso rápido da memória principal, a fim de exercer a mesma função. Em último nível temos a memória secundária, a única não-volátil, e a mais lenta entre todas.

2.3 Exercício slide 27

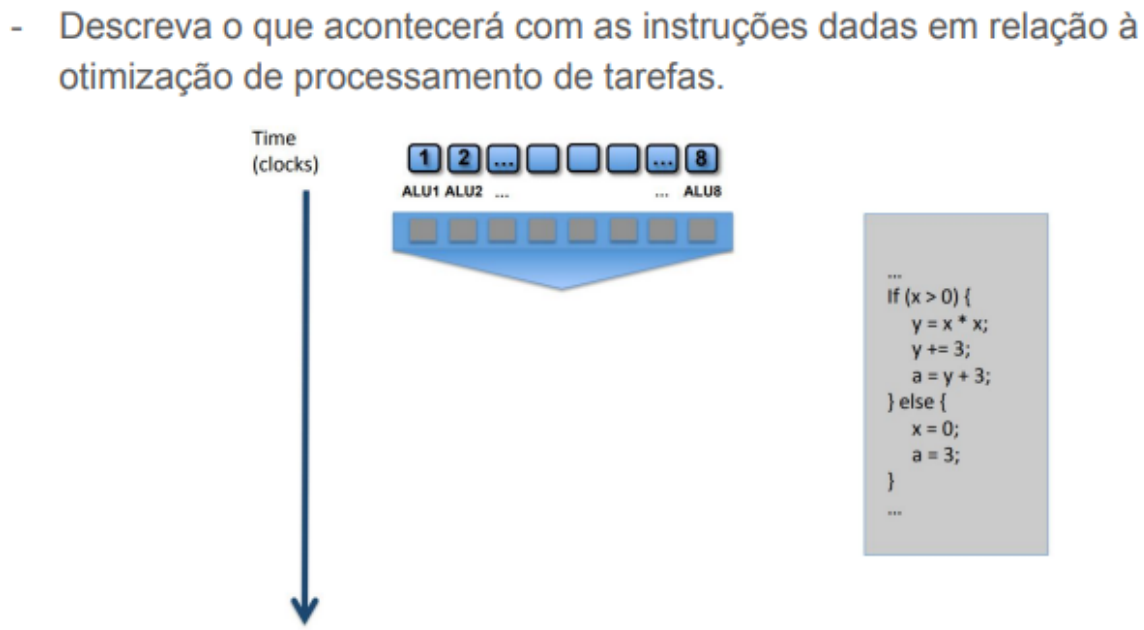

Figura 5: Slide 27 - (PEDRINI, 2025)

Essa estrutura faz com que algumas instruções dependam do resultado de outras, além de usar um comando "if-else" que divide o caminho que a execução pode seguir. Isso faz com que nem todas as unidades de processamento possam trabalhar ao mesmo tempo, já que algumas precisam esperar a conclusão de outras. Como resultado, o aproveitamento do processador não é o ideal e parte dos recursos acaba ficando parada, o que atrapalha a otimização do desempenho.

2.4 Pesquise sobre a família Pascal de GPUs e suas tecnologias

- Possui 64kb exclusivo para memoria compartilhada e memória cache/textura
- Maior capacidade de memória DRAM e maior banda de memória
- Nova tecnologia **NVlink** que permite que GPUs troquem informações entre si, via PCIexpress, ou GPU + CPU, tbm permite uma banda de 40gb/s de troca unidirecional. Com a finalidade de poder integrar uma ou mais GPUs a um processador, simulando aceleradores de alto desempenho.

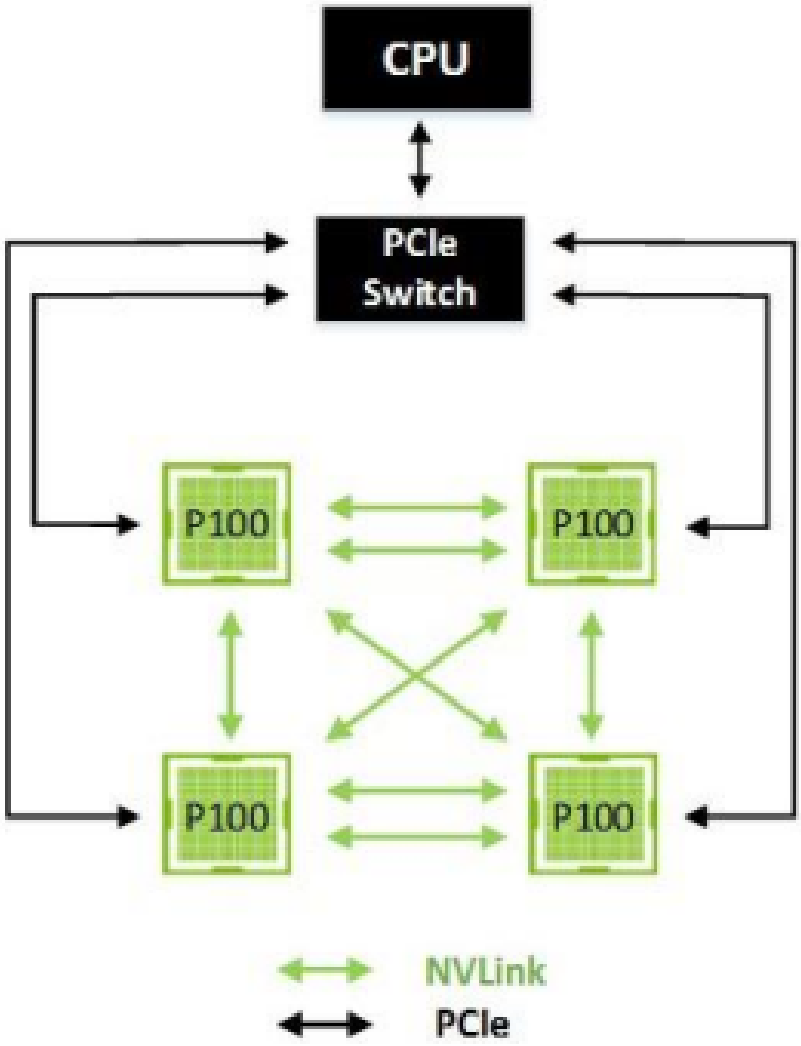

Figura 6: Topologia do NVlink -(NVIDIA CORPORATION, 2016)

- Memória unificada, por meio de endereços virtuais, para facilitar o acesso imediato aos dados pela GPU e pelo processador, sem concorrência, garantindo a coerência dos dados e facilitando o gerenciamento da memória por parte dos programadores, para que estes se preocupem apenas na paralelização das aplicações.

- Computação preemptiva com granularidade a nível de instrução, para prevenir que threads muito longas monopolizem o tempo de processamento da GPU.

Figura 7: Pascal GP100 SM Unit - (NVIDIA CORPORATION, 2016)

2.5 O padrão Infiniband de comunicação entre GPUs

Sendo as GPUs majoritariamente utilizadas em ambientes de computação de alta performance, a baixa latência e consistência na comunicação entre os dispositivos é crucial para atingir um desempenho otimizado. O padrão infiniband possui uma tecnologia de Remote Direct Memory Access (RDMA) que permite o acesso direto à memória por dispositivos como GPUs, sem interferência da CPU ou do sistema operacional. Sua arquitetura composta por Host Channel Adapters (HCAs), Target Channel Adapters (TCAs), Routers e Switches, permite a construção de uma malha de interconexão escalável para compor sistemas de computação de alto desempenho, com performance otimizada e simplifica a comunicação entre os dispositivos.

2.6 A importância da sincronização entre Threads e como ela ocorre nas GPUs

De acordo com (KIRK; HWU, 2016), em CUDA as threads são executadas em blocos de 32 threads, chamados warps. As threads nesses blocos podem ser sincronizadas através da função *syncthreads()* que estabelece uma Barreira de Sincronização, à qual todas as threads devem atingir antes de prosseguirem com as tarefas. No entanto, devem ser pré-

estabelecidas restrições para o uso dessa função, pois se utilizadas dentro de um bloco de decisão, pode ocorrer a seguinte situação: uma thread executa a barreira de sincronização no *if* e a outra no *else*, logo ambas ficarão eternamente esperando umas às outras.

Logo, a preempção é feita pelo próprio sistema CUDA, que aloca uma quantidade de recursos limitada ao bloco de threads e este deve realizar a tarefa apenas com este recurso, evitando assim longas espera.

2.7 Quais os papéis das memórias global e local no contexto de GPUs?

A memória global é uma DRAM própria da GPU para carregar os dados da CPU e ter acesso mais direto ao conjunto de dados que será processado, já a memória local, trata-se de um espaço privado para cada thread, na própria memória global, que tem a finalidade de armazenar quaisquer variáveis com qualificação de tipo de variável "automatic array", ou para variáveis muito grandes que não cabem nos registradores, ou para arrays com tamanho indeterminado.

3 Desenvolvimento Prático: Fractais de Mendelbrot

O experimento foi executado em uma placa de vídeo Nvidia GeForce RTX 4090 Laptop com 640 núcleos CUDA.

```
  Capabilities
        ComputeCapability: '8.9'
      MultiprocessorCount: 20
             ClockRateKHz: 2130000
        SingleDoubleRatio: 64

  Kernel Programming
       MaxThreadsPerBlock: 1024
         MaxShmemPerBlock: 49152 (49.15 KB)
       MaxThreadBlockSize: [1024 1024 64]
              MaxGridSize: [2.1475e+09 65535 65535]
                SIMDWidth: 32
           ToolkitVersion: 12.2000
```

Figura 8: gpuDevice

3.1 Exercício 1

Na primeira etapa da prática executamos uma função FFT no matlab para medir o speedup entre CPU e uma GPU. Código executado:

```
A1 = rand(3000,3000);
tic;
B1 = fft(A1);
time1 = toc;
A2 = gpuArray(A1);
tic;
B2 = fft(A2);
time2 = toc;
B2 = gather(B2);
class(B2);
speedup = time1/time2;
fprintf('O Speed up é %.4f\n', speedup);
```

Saída:

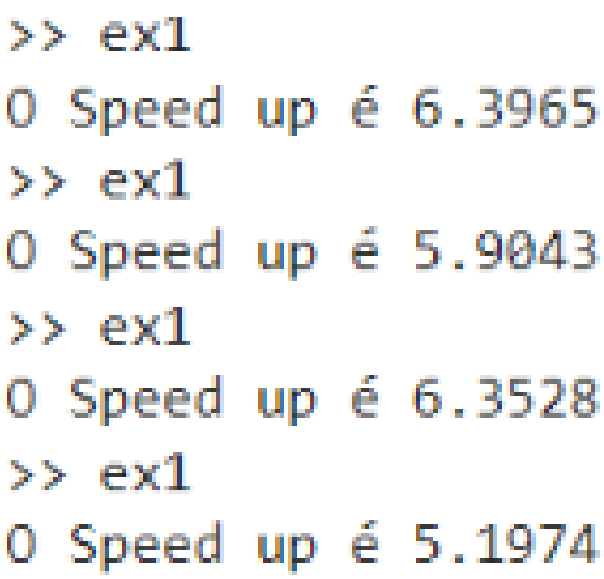

Figura 9: Saída Teste 1

Com as seguintes alterações, pudemos notar que o speed up descresce drásticamente quando consideramos o tempo que leva para trazer os dados para a GPU e devolver à CPU.

```
tic;
A2 = gpuArray(A1);
B2 = fft(A2);
B2 = gather(B2);
time2 = toc;
```

```
>> ex1
O Speed up é 0.1882
>> ex1
O Speed up é 0.2263
>> ex1
O Speed up é 0.1401
>> ex1
O Speed up é 0.1136
```

Figura 10: Teste 2

Em demais testes para entradas substancialmente maiores tivemos o mesmo resultado de speed up baixo.

3.2 Exercício 2

Este exercício consistia em implementar uma função no matlab para plotar o conjunto de Mendelbrot, utilizando uma execução serial, uma paralela – utilizando **gpuArray** –, e uma paralela – utilizando **arrayfun**, para calcular o speed up de cada uma.

3.2.1 Execução Serial

Executando este código:

```
    maxNumCompThreads(1);  %limita o uso de threads a um
N = 1000;              % tamanho do grid
max_iter = 500;        % número máximo de iterações
xlim = [-2, 1];        % intervalo no eixo real
ylim = [-1.5, 1.5];    % intervalo no eixo imaginário
% Criar o plano complexo
x = linspace(xlim(1), xlim(2), N);
y = linspace(ylim(1), ylim(2), N);
[X, Y] = meshgrid(x, y);
```

```
C = X + 1i * Y;        % número complexo c = x + i*y
Z = zeros(size(C));   % inicia Z como zero
M = zeros(size(C));   % matriz para armazenar número de iterações
% Loop principal (versão serial)
tic;
for k = 1:max_iter
   Z = Z.^2 + C;                    % iteração Z = Z^2 + c
   escaped = abs(Z) > 2 & M == 0;    % verifica escape
   M(escaped) = k;                   % salva a iteração em que escapou
end
% Mostrar imagem
time = toc;
imagesc(x, y, M); colormap(hot); axis image;
xlabel('Parte Real'); ylabel('Parte Imaginária');
title(['Conjunto de Mandelbrot (CPU) - Tempo: ', num2str(time), ' s']);
colorbar;
```

Obtivemos a seguinte imagem no plano complexo, em **17.33s**:

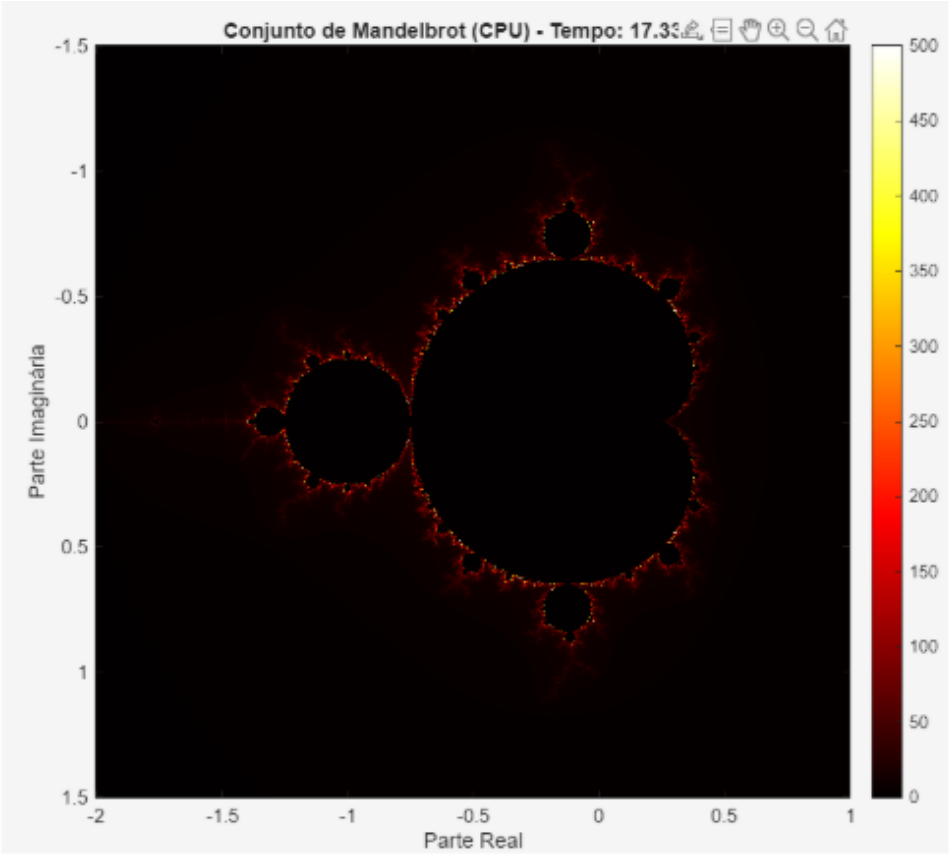

Figura 11: Plano Complexo - Serial

3.2.2 Execução paralaela: gpuArray

Utilizando o gpuArray, obtivemos a mesma imagem em apenas **0.026s**, configurando um *SpeedUp* de 640x mais rápido que a execução serial, ou **1.5s** quando consideramos o tempo de trazer os dados para a GPU e devolver à CPU (*SpeedUp* de 11.5x).

```
    N = 1000;                % tamanho do grid
max_iter = 500;          % número máximo de iterações
xlim = [-2, 1];          % intervalo no eixo real
ylim = [-1.5, 1.5];      % intervalo no eixo imaginário

% Criar o plano complexo
x = linspace(xlim(1), xlim(2), N);
y = linspace(ylim(1), ylim(2), N);
[X, Y] = meshgrid(x, y);
C = X + 1i * Y;          % número complexo c = x + i*y
```

```
% Transferir os dados para GPU
C_gpu = gpuArray(C);
Z_gpu = gpuArray.zeros(size(C_gpu));
M_gpu = gpuArray.zeros(size(C_gpu), 'uint16'); % matriz de iterações

% Medir tempo de execução na GPU
tic;
for k = 1:max_iter
    Z_gpu = Z_gpu.^2 + C_gpu;                    % iteração Z = Z^2 + c
    escaped = abs(Z_gpu) > 2 & M_gpu == 0;       % pontos que escaparam nesta iter
    M_gpu(escaped) = k;                          % salva a iteração de escape
end
gpu_time = toc;

% Trazer resultado de volta para CPU
M = gather(M_gpu);

% Mostrar imagem
imagesc(x, y, M); colormap(hot); axis image;
xlabel('Parte Real'); ylabel('Parte Imaginária');
title(['Conjunto de Mandelbrot (GPU) - Tempo: ', num2str(gpu_time), ' s']);
colorbar;
```

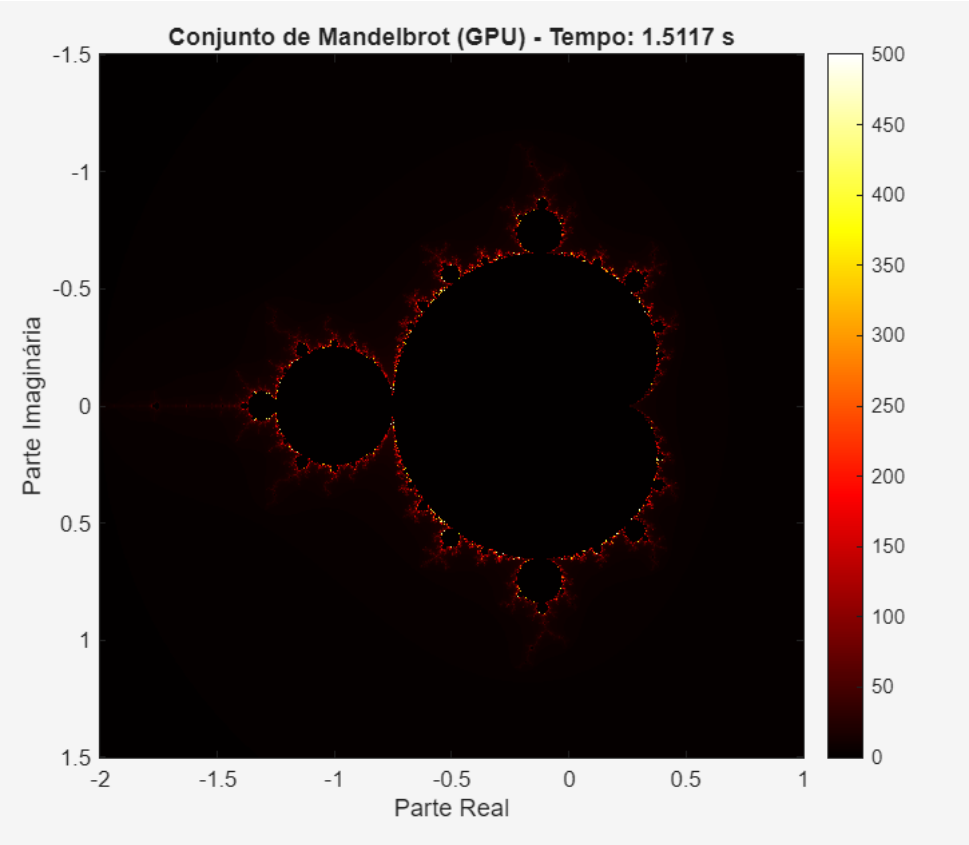

Figura 12: Plano Complexo - gpuArray

3.2.3 Execução Paralela - arrayfun

Utilizando arrayfun conseguimos uma imagem similar, no tempo de **3.8 segundos**, um *SpeedUp* de 4.5x em relação à implementação serial. Mas ainda sim, mais lenta que o gpuArray.

```
    N = 1000;% Tamanho da imagem
max_iter = 500;%Número de iterações
% Limites do plano complexo
xlim = [-2, 1];        % intervalo no eixo real
ylim = [-1.5, 1.5];    % intervalo no eixo imaginário
% Geração da grade de coordenadas
x = linspace(xlim(1), xlim(2), N);
y = linspace(ylim(1), ylim(2), N);
[X, Y] = meshgrid(x, y);
C = X + 1i * Y;   % número complexo c = x + i*y
```

```
% Função para verificar se pertence ao conjunto de Mandelbrot
mandelbrotFunc = @(c) mandelbrot_iter(c, max_iter);
% Aplica a função a cada elemento do array
tic;
M = arrayfun(mandelbrotFunc, C);
time = toc;
% Exibe a imagem do fractal
imagesc(x, y, M); colormap(hot); axis image;
xlabel('Parte Real'); ylabel('Parte Imaginária');
title(['Conjunto de Mandelbrot (Arrayfun) - Tempo: ', num2str(time), ' s']);
% ------------------------------
% Função para calcular Mandelbrot
function count = mandelbrot_iter(c, max_iter)
   z = 0;
   for k = 1:max_iter
       z = z^2 + c;
       if abs(z) > 2
           count = k;
           return;
       end
   end
   count = max_iter;
end
```

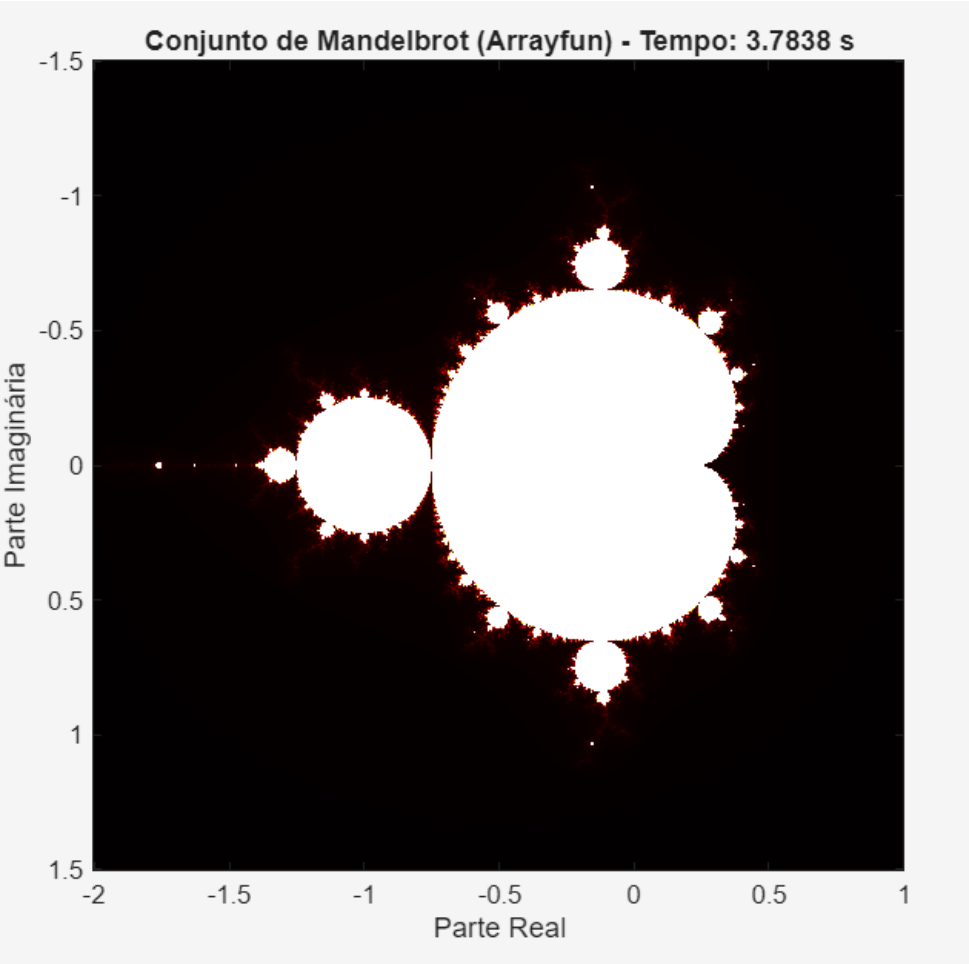

Figura 13: Plano Complexo - arrayfun

3.3 Conclusão

Os experimentos realizados demonstraram, na prática, o enorme potencial de aceleração que as GPUs oferecem em tarefas paralelizáveis, como a geração do conjunto de Mandelbrot e o cálculo de FFTs. Observou-se que, embora a GPU seja extremamente rápida na execução de operações matemáticas, o tempo total de execução pode ser fortemente afetado pelas transferências de dados entre CPU e GPU — um fator crítico que limita o speedup efetivo.

A comparação entre a execução serial, com 'gpuArray' e com 'arrayfun' mostrou que:

A versão com 'gpuArray', mantendo os dados na GPU, obteve o maior speedup, chegando a **640x** em relação à versão serial. Quando considerada a transferência de dados, o speedup caiu para 11,5x, evidenciando o custo da comunicação CPU e GPU. A versão com 'arrayfun' obteve desempenho intermediário, com speedup de 4,5x, mas ainda aquém do obtido com 'gpuArray'.

Além disso, os testes com ‘fft’ evidenciaram que **operações de uso intensivo de memória** e **funções já otimizadas na CPU** podem não se beneficiar tanto da GPU, especialmente se envolverem movimentação de dados entre os dispositivos.

Concluímos que, para aproveitar ao máximo o poder de paralelismo massivo das GPUs, é fundamental **minimizar as transferências de dados** e manter os dados e cálculos no dispositivo sempre que possível. Isso exige não apenas conhecimento das funções paralelas do MATLAB, mas também entendimento da **arquitetura de memória e execução das GPUs**, conforme estudado na parte teórica do trabalho.

Referências

GLASKOWSKY, Peter N. **NVIDIA's Fermi: The First Complete GPU Computing Architecture**. [S.l.], 2010. Acessado em julho de 2025.

KIRK, David B.; HWU, Wen-mei W. **Programming Massively Parallel Processors: A Hands-on Approach**. 3. ed. San Francisco: Morgan Kaufmann, 2016. ISBN 9780128119860.

NVIDIA CORPORATION. **CUDA C Programming Guide**. [S.l.], 2025. Acessado em julho de 2025.

_____. **Pascal Architecture Whitepaper**. [S.l.: s.n.], 2016. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`. Acessado em julho de 2025.

PEDRINI, Emerson Carlos. Conteúdo da Aula - Arquitetura de GPUs. Material didático em PDF fornecido na disciplina de Arquitetura de Alto Desempenho, Universidade Federal de São Carlos. [S.l.], 2025.