Otimização do Desempenho de uma Simulação de Dinâmica de Fluidos

1st Augusto Campos
Universidade do Minho
Braga, Portugal
pg57510@uminho.pt

2st João Rodrigues
Universidade do Minho
Braga, Portugal
pg52687@uminho.pt

3st Tiago Silva
Universidade do Minho
Braga, Portugal
pg57617@uminho.pt

***Abstract*—Este trabalho foca-se na otimização do desempenho de uma simulação de dinâmica de fluidos através de melhorias progressivas ao longo de três fases. Na primeira fase, utilizámos ferramentas de análise de desempenho para identificar partes do código críticas em desempenho e aplicámos técnicas como melhorias na localidade de dados e vetorização, o que nos levou a reduções significantes no tempo de execução. Na segunda fase, explorámos a paralelização com OpenMP, aproveitando a memória partilhada para acelerar o processamento e melhorar a escalabilidade. Finalmente, na terceira fase, implementámos uma solução baseada em CUDA para explorar o paralelismo oferecido por GPUs.**

***Index Terms*—Otimização de desempenho, profiling, OpenMP, CUDA, vetorização, paralelismo, memória partilhada, GPUs**

I. Fase 1: Otimização Sequencial

A. Introdução

Nesta primeira fase, o foco foi na análise detalhada do código original e na aplicação de otimizações sequenciais, utilizando técnicas ensinadas ao longo da disciplina. Inicialmente, utilizamos ferramentas de *profiling* como o *gprof* para identificar as áreas mais críticas em termos de custo computacional. Com base nesses resultados, implementamos melhorias como a otimização da localidade de dados e vetorização, com o objetivo de reduzir o tempo de execução sem comprometer a precisão.

B. Metodologia

A nossa abordagem foi dividida em três fases principais: análise do código, otimização do mesmo e verificação do desempenho após as alterações. Este é um processo iterativo que aplicamos no desenvolvimento de cada versão da implementação.

1) Análise de Desempenho Inicial: A análise do desempenho inicial foi realizada utilizando o *gprof*, uma ferramenta de *profiling* que nos permitiu identificar as partes do código mais críticas em termos de tempo de execução.

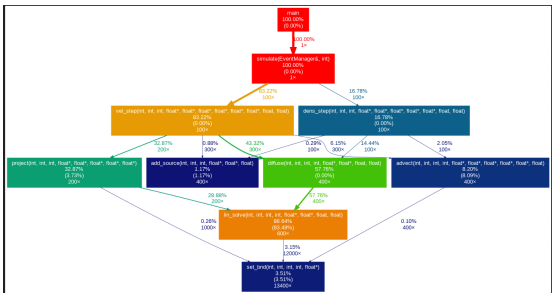


Fig. 1. Resultados gprof do código não otimizado

2) Técnicas de Otimização: Com base nas informações adquiridas, aplicamos as seguintes técnicas de otimização:

- Macro Uma das principais estratégias adotadas para otimizar o desempenho foi a modificação de uma macro. A alteração consistiu na troca da posição dos índices utilizados no cálculo do acesso à matriz. Esta mudança melhora a localidade dos dados em memória, resultando num uso mais eficiente do cache da CPU e, consequentemente, melhorando o desempenho.
- Processamento por Blocos O processamento por blocos foi implementado para dividir as operações de cálculo em grupos, permitindo uma melhor utilização da hierarquia de memória e das capacidades de paralelismo da *CPU*. Optamos pelo processamento em blocos de tamanho 4, em vez de blocos maiores, por uma questão de balanço entre a utilização eficiente do cache e a sobrecarga de gerenciamento de blocos maiores. Blocos de tamanho 4 permitiram uma maior flexibilidade no uso do cache L1 e L2, além de reduzirem *cache misses* devido ao tamanho limitado da cache.
- Cálculo prévio dos índices e remoção de divisões em vírgula flutuante Na função lin_solve utilizamos a macro uma única vez para pré calcular o índice do elemento *IX(i,j,k)* e aceder aos seus "vizinhos" de forma mais direta através do seus endereços. Também substituímos a divisão existente nessa função por uma multiplicação pelo inverso pré calculado, eliminando as divisões em vírgula flutuante, já que estas são mais caras em termos de ciclos de CPU. Desta maneira temos apenas uma divisão fora dos ciclos e uma operação menos custosa a ser repetida inúmeras vezes dentro dos ciclos.

- Mudanças no Makefile Outra abordagem que contribuiu para a otimização do tempo de execução do programa foi a modificação do *Makefile*. As mudanças incluem a adição de diversas flags, que têm como objetivo melhorar o desempenho da aplicação. Neste processo, testamos diversas combinações de *flags*. Após estes testes, descobrimos que a combinação que resultou no melhor desempenho em termos de tempo de execução foi a seguinte:
  - `-funroll-loops`: Permite reduzir o número de iterações dos *loops*, que aumentou a velocidade de execução, especialmente para loops que são executados muitas vezes.
  - `-Ofast`: Ativa todas as otimizações que não comprometem a conformidade com o padrão, resultando em um desempenho significativamente mais rápido.
  - `-march=native`: Faz com que o compilador gere código otimizado para a arquitetura específica do processador em que o programa está a ser compilado, aproveitando ao máximo as características disponíveis.
  - `-ftree-vectorize`: Habilita a vetorização de loops, permitindo que operações sejam realizadas em paralelo, utilizando as capacidades *Single Instruction Multiple Data* do hardware.

C. Resultados

Os testes realizados demonstraram uma redução significativa no tempo total de execução da simulação após a aplicação das otimizações. No código original, o tempo médio de execução era de 27,5 segundos. Com as melhorias implementadas na Versão 2, que incluiu a alteração da macro, remoção da divisão em virgula flutuante e a adição de flags que otimizam a execução, esse tempo foi reduzido para 5,2 segundos.

Após a adição de *tiling*, realizamos testes com um block size de 16, que apresentou um tempo de execução de 4,2 segundos. No entanto, a análise revelou que a taxa de cache miss aumentou para o dobro, o que nos levou a concluir que, possivelmente, o block não estava completamente na L1 data cache. Por isso, diminuímos o block size para 4, onde conseguimos o menor valor percentual até então de cache misses, que foi de 2,49%.

Além disso, para assegurar que as otimizações não comprometeram a precisão dos resultados, realizamos uma comparação entre os valores finais das simulações originais e otimizadas. Os resultados mostraram uma diferença mínima de apenas 0,1 unidades.

TABLE I
COMPARAÇÃO DE DESEMPENHO ENTRE CÓDIGO ORIGINAL, VERSÃO 2, OTIMIZADO E BLOCK SIZE 16

Métrica	Original	Versão 2	Block Size 16	Otimizado
Instruções	166,406,023,725	15,069,383,874	16,603,273,185	22,080,727,424
Ciclos	88,198,612,312	16,942,673,874	13,530,605,936	11,641,036,958
Branch Misses	24,699,511	1,484,162	1,764,700	2,552,430
L1D Loads	70,394,703,031	6,361,206,948	6,965,087,527	8,928,285,506
L1D Misses	2,305,922,907 (3.28%)	207,088,649 (3.26%)	455,000,643 (6.53%)	222,694,550 (2.49%)
Tempo (s)	27.446 ± 0.279	5.2302 ± 0.0549	4.2069 ± 0.0651	3.7677 ± 0.0168

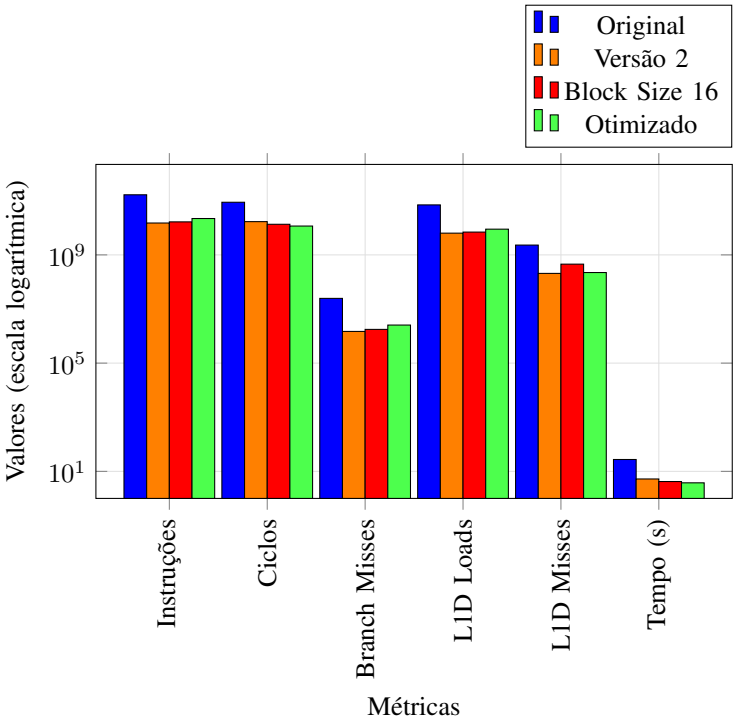


Fig. 2. Comparação de Desempenho entre Código Original, Versão 2, Block Size 16 e Otimizado

II. Fase 2: Paralelização com OpenMP

A. Introdução

O objetivo desta fase foi aplicar técnicas de paralelismo para otimizar um programa, reduzindo o seu tempo de execução. Para atingir este objetivo, utilizámos ferramentas de *profiling* para identificar os pontos críticos do programa, onde o custo computacional é mais elevado. De seguida, implementámos técnicas de paralelismo e avaliámos o desempenho da solução adotada por meio de análises de escalabilidade e eficiência.

B. Identificação dos hotspots do programa

Através da ferramenta de *profiling perf*, foi possível identificar que a função *lin_solve* é responsável pela maior parte do tempo de execução do programa. Por este motivo, esta função foi escolhida como o principal foco de paralelização, uma vez que melhorar o seu desempenho pode trazer uma melhoria significativa no tempo de execução do programa.

C. Implementação de técnicas de paralelismo

Nesta implementação, optamos por paralelizar as operações usando loops paralelizados com OpenMP, aproveitando instruções como *#pragma omp parallel for* e *collapse* para distribuir uniformemente o trabalho entre as threads. Também utilizamos *reduction* para computar valores acumulados (neste caso o *max_c* em *lin_solve*) de forma eficiente em paralelo e *simd* para explorar o paralelismo a nível de vetor em algumas operações internas. O uso de barreiras (*#pragma omp barrier*) foi necessário em pontos específicos para sincronizar as threads entre fases distintas da computação, garantindo a consistência dos dados.

Outra abordagem possível seria o uso de tasks, em que o trabalho seria dividido dinamicamente em tarefas de tempo de execução imprevisível, permitindo um agendamento mais adaptável.

A escolha dos loops paralelizados foi motivada pela natureza do problema e pela estrutura regular dos dados. Neste código, a matriz permite uma divisão estática eficiente das iterações. Ao usar loops paralelizados com *collapse* e *reduction*, garantimos que cada thread executa blocos contínuos de trabalho, reduzindo o overhead associado ao *scheduling* dinâmico e aproveitando melhor a cache.

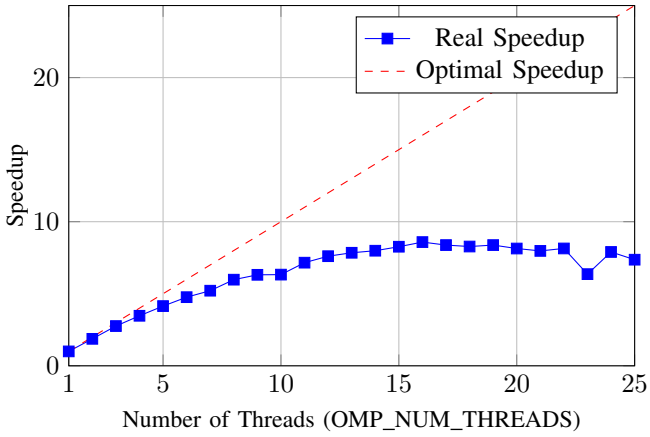
Por outro lado, a abordagem baseada em tasks, embora mais flexível, introduz maior overhead devido ao custo de criação, gestão e sincronização das tarefas. Essa técnica seria mais vantajosa em problemas com carga de trabalho altamente irregular ou dinâmica, o que não é o caso aqui.

Portanto, a nossa abordagem é superior neste contexto mantendo o código eficiente, simples e escalável em sistemas com múltiplos núcleos, enquanto minimiza o overhead e aproveita a estrutura regular da matriz para uma execução paralela previsível e balanceada.

D. Análise de performance

Como podemos ver pelo gráfico da Figura 2, a redução do tempo de execução é significativa até cerca de 16 threads, demonstrando uma boa escalabilidade inicial. Para valores superiores, os ganhos tornam-se marginais, e surgem oscilações no tempo de execução.

Estas oscilações podem ser explicadas devido ao overhead de comunicação entre threads, conflitos de cache e barreiras de sincronização, que afetam a eficiência à medida que mais threads são utilizadas. Apesar disso, o desempenho geral mostra que a paralelização foi bem sucedida, proporcionando uma redução substancial no tempo de execução.



III. Phase 3: GPU Acceleration with CUDA

A. Introdução

Nesta última fase, implementámos a simulação de dinâmica de fluidos com CUDA, focando em explorar o paralelismo das GPUs para melhorar o desempenho. Foram paralelizadas funções críticas, aproveitando a hierarquia de grids e blocks das GPUs. Além disso, otimizámos a gestão de memória e a sincronização de threads para reduzir latências.

B. Arquitetura e Planeamento

A implementação em CUDA foi estruturada para maximizar a eficiência e escalabilidade da simulação de dinâmica de fluidos, explorando o paralelismo massivo das GPUs. O planejamento focou nos seguintes pontos principais:

- **Identificação de Componentes Críticos:** Foram priorizadas funções como lin_solve, diffuse e advect para paralelização, por serem responsáveis pela maior parte do tempo de execução.
- **Estratégia de Paralelismo:** O espaço tridimensional da simulação foi mapeado para um grid 3D de threads CUDA, garantindo paralelismo em larga escala, com cada thread responsável por um ponto do grid.
- **Gestão de Memória:** Matrizes principais foram alocadas na memória global da GPU usando cudaMalloc. Sempre que possível, utilizou-se memória compartilhada para reduzir latências e otimizar acessos frequentes.
- **Configuração de Grids e Blocks:** Configuraram-se grids e blocks para explorar ao máximo os multiprocessadores da GPU, otimizando o número de threads por bloco.
- **Sincronização e Otimizações:** Sincronizações com __syncthreads() foram usadas para consistência de dados. Operações foram agrupadas para minimizar latências e reduzir a comunicação CPU-GPU.

C. Implementação

A implementação da simulação com CUDA concentrou-se em adaptar as funções críticas para aproveitar o paralelismo das GPUs. O principal foco foi a conversão das funções mais computacionalmente intensivas, como `lin_solve`, `diffuse` e `advect`, além de estratégias eficientes de gestão de memória e sincronização para garantir a consistência dos dados.

1) Paralelização de Funções Críticas: A função `lin_solve` foi o ponto central de otimização, pois resolve sistemas lineares iterativamente em um grid tridimensional. Na implementação CUDA, utilizou-se a estratégia de pontos alternados (*red-black Gauss-Seidel*) com dois kernels separados:

- **Kernel para Pontos Vermelhos:** Processa os pontos do grid onde $(i + j + k)\%2 == 0$.
- **Kernel para Pontos Pretos:** Processa os pontos do grid onde $(i + j + k + 1)\%2 == 0$.

Cada kernel foi projetado para operar num ponto específico do grid tridimensional.

Exemplo de Kernel CUDA para Pontos Vermelhos:

```
__global__ void red_dot_kernel(int M, int N, int O,
    float *x, float *x0, float a, float c, float *
    max_changes)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x +
        1;
    int j = blockIdx.y * blockDim.y + threadIdx.y +
        1;
    int k = blockIdx.z * blockDim.z + threadIdx.z +
        1;

    if (i <= M && j <= N && k <= O && (i + j + k) %
        2 == 0)
    {
        float old_x = x[IX(i, j, k)];
        x[IX(i, j, k)] = (x0[IX(i, j, k)] +
                          a * (x[IX(i - 1, j, k)] +
                               x[IX(i + 1, j, k)] +
                               x[IX(i, j - 1, k)] +
                                    x[IX(i, j + 1, k)
                                    ] +
                               x[IX(i, j, k - 1)] +
                                    x[IX(i, j, k + 1)
                                    ])) / c;

        float change = fabs(x[IX(i, j, k)] - old_x);
        max_changes[IX(i, j, k)] = change;
    }
}
```

Listing 1. Kernel CUDA para Pontos Vermelhos

Após processar os pontos vermelhos, os pontos pretos são atualizados por um kernel semelhante. A alternância entre os dois kernels continua até que a solução converja ou o número máximo de iterações seja atingido.

Execução Alternada dos Kernels:

```
red_dot_kernel<<<numBlocks, threadsPerBlock>>>(M, N,
     O, x, x0, a, c, d_max_changes);
cudaDeviceSynchronize();
black_dot_kernel<<<numBlocks, threadsPerBlock>>>(M,
    N, O, x, x0, a, c, d_max_changes);
cudaDeviceSynchronize();
```

Listing 2. Execução Alternada dos Kernels

Esta abordagem evita dependências de leitura-escrita entre threads e garante consistência no cálculo.

2) Exemplo de Kernel Simples: Um exemplo de kernel básico implementado é o `set_bnd_kernel`. Este kernel demonstra como o CUDA distribui cálculos por threads para processar um grid tridimensional. O kernel é responsável apenas por realizar os cálculos em paralelo para cada ponto do grid e sincronizar no final.

Exemplo de Kernel simples em CUDA:

```
__global__ void set_bnd_kernel(int M, int N, int O,
    int b, float *x) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    if (idx > 0 && idx <= M && idy > 0 && idy <= N)
        {
        x[IX(idx, idy, 0)] = b == 3 ? -x[IX(idx, idy
            , 1)] : x[IX(idx, idy, 1)];
        x[IX(idx, idy, O + 1)] = b == 3 ? -x[IX(idx,
             idy, O)] : x[IX(idx, idy, O)];
    }
    __syncthreads(); // Sincroniza o para
        garantir consist ncia
}
```

Listing 3. Exemplo de Kernel Simples em CUDA

3) Gestão de Memória: Os dados tridimensionais foram alocados na memória global da GPU para serem acessados pelos kernels CUDA. Após a execução, os resultados são transferidos de volta para a CPU.

Exemplo de Alocação e Transferência:

```
float *d_max_changes, *d_max_result;
int reduction_size = (M * N * O + 255) / 256;

cudaMalloc((void **)&d_max_changes, sizeof(float) *
    M * N * O);
cudaMalloc((void **)&d_max_result, sizeof(float) *
    reduction_size);
cudaMemset(d_max_changes, 0, sizeof(float) * M * N *
     O);
```

Listing 4. Alocação e Transferência de Memória

No final, a memória é libertada:

```
cudaFree(d_max_changes);
cudaFree(d_max_result);
```

Listing 5. Liberação de Memória

4) Definição de Grids e Blocks: Para o mapeamento tridimensional, os kernels foram estruturados com dimensões adequadas de threads por bloco e blocos por grid.

Configuração Utilizada no Código:

```
dim3 threadsPerBlock(16, 16, 4);
dim3 numBlocks((M + threadsPerBlock.x - 1) /
    threadsPerBlock.x,
                (N + threadsPerBlock.y - 1) /
                    threadsPerBlock.y,
                (O + threadsPerBlock.z - 1) /
                    threadsPerBlock.z);
```

Listing 6. Configuração de Grids e Blocks

Esta estrutura assegura que o domínio tridimensional seja subdividido em partes menores, permitindo que cada thread CUDA processe um ponto específico do grid tridimensional.

5) Sincronização e Otimizações: Sincronizações explícitas foram utilizadas para garantir consistência:

- **cudaDeviceSynchronize():** Garante que os kernels tenham concluído antes de avançar.
- **Redução de Máximos:** Foi implementado um kernel para calcular a mudança máxima no grid de forma eficiente, utilizando memória compartilhada para acelerar o processo.

Exemplo de Redução:

```
__global__ void reduction_kernel(float *max_changes,
     float *result, int size)
{
    extern __shared__ float shared_data[];

    int tid = threadIdx.x;
    int index = blockIdx.x * blockDim.x + tid;

    shared_data[tid] = (index < size) ? max_changes[
        index] : 0.0f;
    __syncthreads();

    for (int stride = blockDim.x / 2; stride > 0;
        stride >>= 1)
    {
        if (tid < stride)
        {
            shared_data[tid] = fmaxf(shared_data[tid
                ], shared_data[tid + stride]);
        }
        __syncthreads();
    }

    if (tid == 0)
    {
        result[blockIdx.x] = shared_data[0];
    }
}
```

Listing 7. Kernel para Redução de Máximos

D. Resultados de Desempenho

Os testes de desempenho foram realizados num sistema equipado com um processador AMD Ryzen 5 5600 (6 núcleos e 12 threads) e uma NVIDIA GTX 1060 com 6 GB de memória. As medições consideraram três versões distintas do código:

- **Fase 1:** Implementação sequencial otimizada.
- **Fase 2:** Implementação paralelizada com OpenMP.
- **Fase 3:** Implementação com CUDA.

TABLE II
COMPARAÇÃO DE DESEMPENHO ENTRE AS FASES 1, 2 E 3

Tamanho do Domínio	Fase 1 (s)	Fase 2 (s)	Speedup (F2)	Fase 3 (s)	Speedup (F3)
42^3	0.57	0.15	3.8	2.75	0.21
84^3	4.35	0.90	4.8	16.60	0.26
168^3	44.31	14.02	3.2	128.52	0.34

Os resultados, apresentados na Tabela II, mostram que a Fase 2 (OpenMP) trouxe ganhos significativos em comparação com a versão sequencial, alcançando speedups de até 4.8 vezes. Entretanto, a Fase 3 (CUDA) não conseguiu superar a versão paralelizada em OpenMP, apresentando tempos de execução consideravelmente superiores.

E. Escalabilidade e Eficiência

Embora esta fase tenha apresentado tempos de execução superiores à implementação sequencial em todos os tamanhos de domínio testados, é possível observar uma tendência positiva em termos de escalabilidade. Conforme o tamanho do domínio tridimensional aumenta, o speedup relativo da Fase 3 em relação à Fase 1 melhora gradualmente. Este comportamento reflete a capacidade do CUDA de lidar melhor com cargas de trabalho maiores.

Este comportamento sugere que, apesar de ainda ser ineficiente em termos absolutos, a Fase 3 demonstra um ganho de eficiência relativo conforme o tamanho do problema cresce. Esse padrão é esperado em implementações CUDA, onde o overhead inicial tem menor impacto proporcional em problemas de maior escala.

F. Comparação entre as diferentes abordagens

A implementação CUDA e a paralelização com OpenMP representam abordagens distintas para explorar o paralelismo, cada uma com seus benefícios, limitações e cenários ideais de aplicação. Casos em que Cada Abordagem é Mais Indicada:

- **OpenMP:**A abordagem com OpenMP é mais indicada para problemas de pequena a média escala. Nestes casos, o tempo de execução em CPU é significativamente menor devido ao reduzido overhead de comunicação e sincronização entre as threads. Além disso, é a solução ideal para situações em que o hardware é limitado, ou seja, quando não há acesso a uma GPU ou quando o sistema possui um processador com um número considerável de núcleos.
- **CUDA:**A abordagem com CUDA é mais indicada para problemas de grande escala. Nestes casos, o custo computacional elevado pode compensar o overhead inicial de transferência de dados entre a CPU e a GPU. Além disso, CUDA é altamente eficiente para cálculos paralelos, em que muitas operações independentes podem ser distribuídas entre os milhares de núcleos CUDA disponíveis. Sendo esta abordagem vantajosa em sistemas com hardware dedicado, como GPUs de alto desempenho.

IV. Conlusão

Este trabalho explorou a otimização de uma simulação de dinâmica de fluidos em três fases: sequencial, paralelizada com OpenMP e acelerada com CUDA. A implementação com OpenMP apresentou os melhores resultados, aproveitando eficientemente o paralelismo da CPU para domínios pequenos e médios. Já o CUDA não alcançou os ganhos esperados, devido ao overhead de transferência de dados e à baixa ocupação da GPU para os tamanhos testados, destacando a necessidade de problemas maiores ou hardware mais robusto para explorar seu potencial.

Em resumo, o OpenMP provou ser a abordagem mais eficiente no contexto deste estudo, enquanto o CUDA, apesar de promissor, exige condições específicas para justificar sua aplicação. Para trabalhos futuros, recomenda-se explorar domínios maiores e otimizações avançadas em CUDA para maximizar o desempenho.

V. ANEXOS

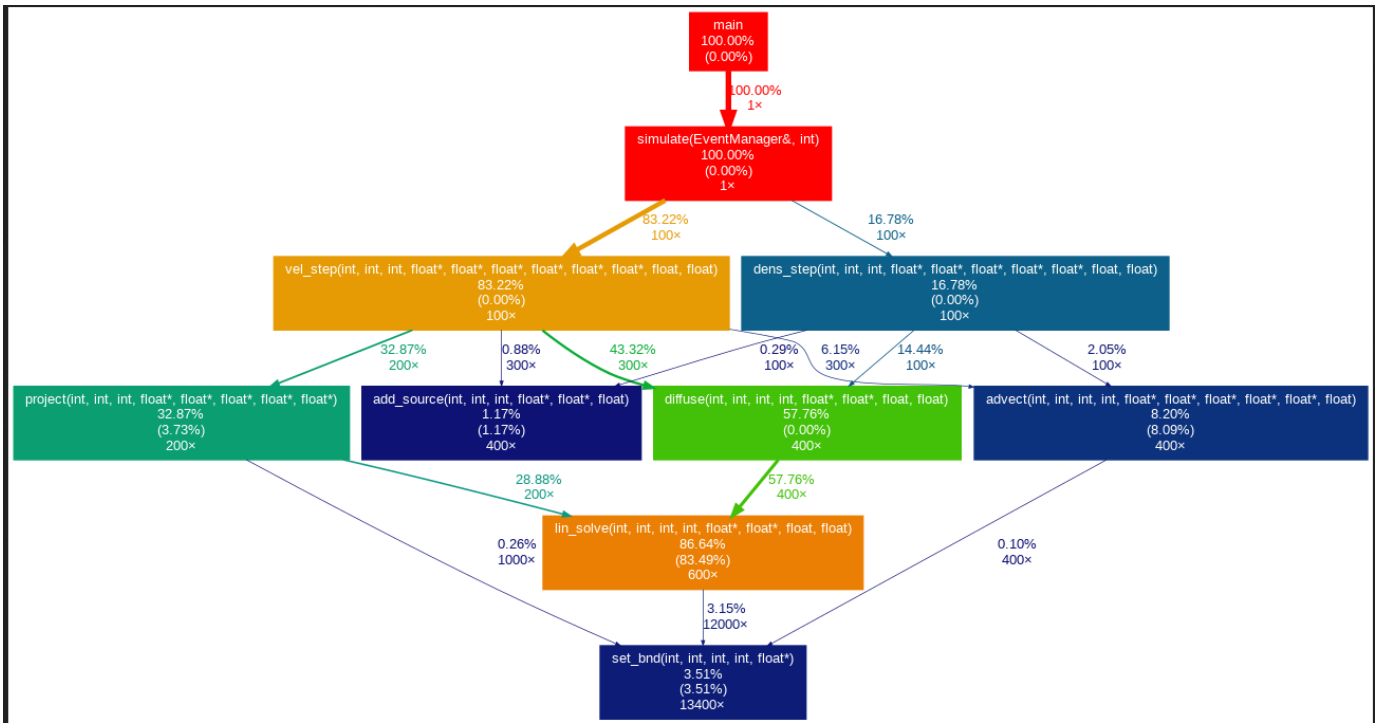


Fig. 1. Resultados gprof do código não otimizado

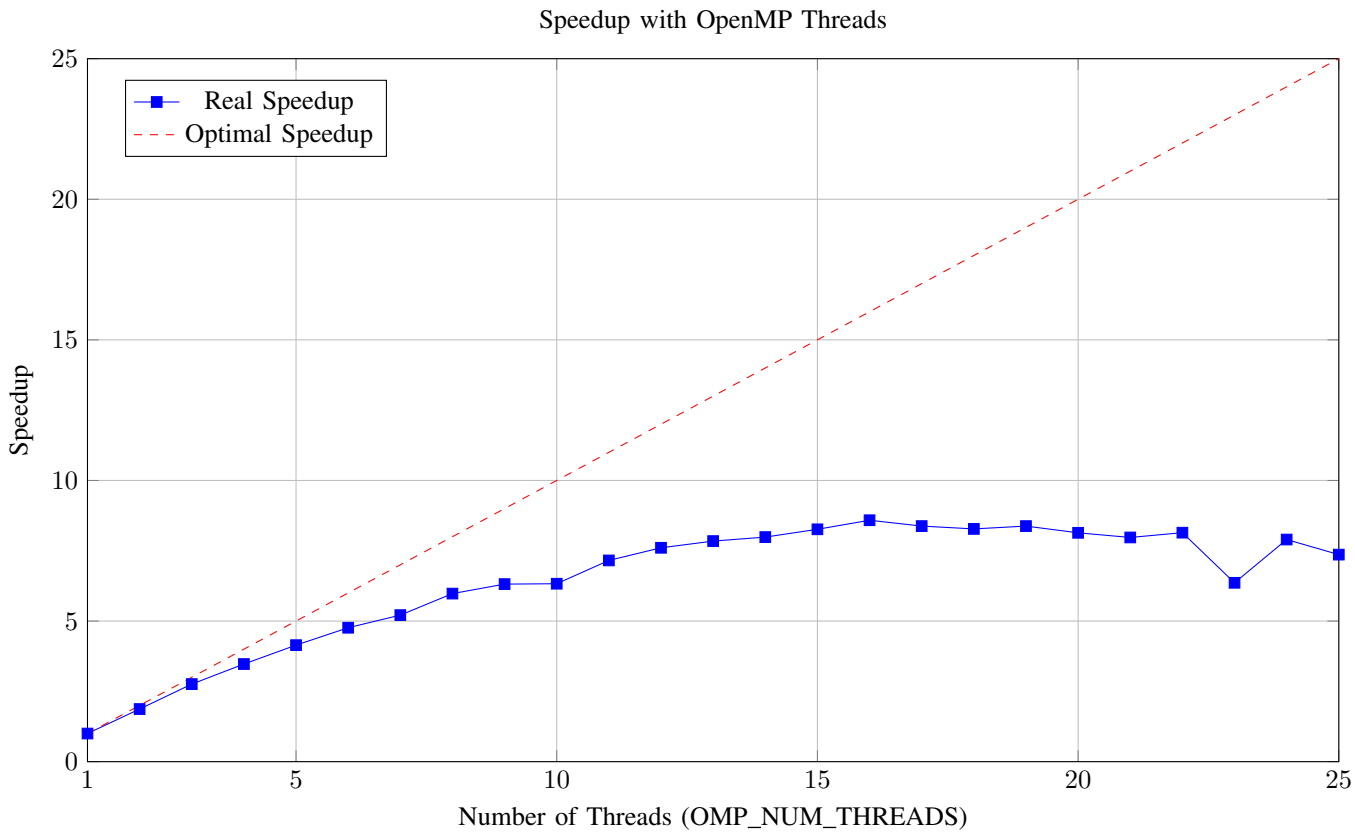


Fig. 2. Speedup with OpenMP Threads