Relatorio Final

João Cerqueira
A103944

Luís David
A103931

Filipe Lemos
A97717

I. Introdução

Muito utilizado em simulações Físicas, um problema de dinâmica de fluidos pode se tornar bastante pesado a nível computacional. Tendo em mãos um problema deste tipo e existindo a necessidade de adaptar algoritmos para se tornarem eficientes em máquinas modernas, ou implementar uso de aceleradores. Vamos partir do código numa versão sequencial/poucas otimizações e otimizar a vários níveis, explorando a arquitetura do CPU, multi-threading e chegando à aplicação em GPU.

A. *Algoritmo*

Neste algoritmo, uma substância (source) é introduzida numa 3D grid, e é aplicada uma força para se espalhar no espaço. A substância e a força são aplicadas no centro da grid com intensidades / direções em tempos específicos (timesteps), representados em events.txt.

É utilizado um solver baseado nas equações de Navier-Stokes para simular a dispersão do fluido. São chamadas funções que calculam a força(`vel_step`) e densidade (`dens_step`) em cada ponto e em cada timestep. Para isso, aplicam a substancia e força com `add_source`, e fazem difusão com diffuse e advect. A função `difuse`, quando chamada pela `dens_step`/`vel_step`, utiliza a `lin_solve` para somar o valor da substancia/força no ponto que é introduzida, ao adicioná-la com o ponto com os valores dos vizinhos de 1º grau que estão na grid, para cada ponto. Substitui esse valor nesse ponto. A função advect utiliza a grid da força para calcular a substância / força resultante. Ambas usam `set_bnd` para atualizar as fronteiras.

No fim soma-se a densidade de todos os elementos para obter o resultado.

II. Explorando a arquitetura do CPU

A. *Analise e Otimizações ILP*

Criando o *call graph* (ver Anexo 1) são identificadas as funções mais problemáticas. `lin_solve` é o grande foco, usa 85.51% dos ciclos comparado aos 0.56% de advect, sendo que ambas fazem cálculos/acessos num bloco de loops aninhados. Apesar de `lin_solve` possuir mais 200 chamadas, a diferença de peso é muita, sugerindo problemas de localidade e/ou dependências na `lin_solve`.

Observando as instruções assembly, reparou-se em repetições provocadas pela expansão da macro `IX` usada repetidamente.

*1) **Dependencias**:* Obteve-se o grafo de dependências baseado nas instruções assembly responsáveis pelo cálculo de x[IX(i,j,k)] e assim verificou-se que o cálculo é realizado sequencialmente, apesar de ser possível uma soma em pares. Este modelo de soma aos pares é introduzido pelo nível de otimização *Ofast* do compilador *gcc*, removendo 2 ciclos de relógio a cada cálculo.

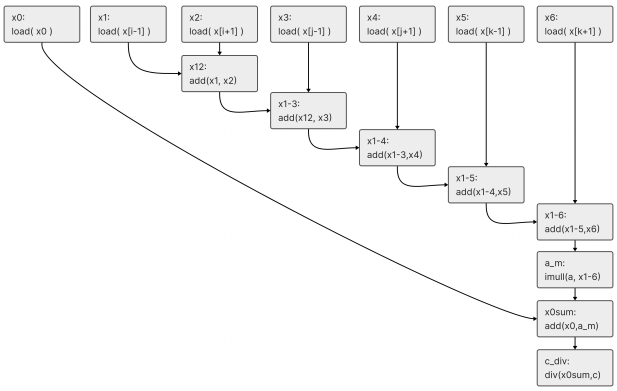


Fig. 1: Grafo Dependencias

*2) **Loop Unrooling**:* Com o loop unrolling (de 3), processa-se três cálculos de x de uma vez. Se guardar os resultados em variáveis pode-se paralelizar alguns cálculos e mais tarde armazenar no array em paralelo. Desta forma, estima-se que o código otimizado possa reduzir os ciclos, pelo menos, cerca de $\frac{2}{3}\#CCA$, sendo $\#CCA$ os ciclos de relógio anteriormente gastos a armazenar os valores no array. Também se reduz as instruções de controlo de ciclo.

B. *Analise e Otimizações Localidade*

O array 1D é uma matriz 3D de 42x42x42 floats. Cada plano ij tem 7kB. O tamanho da L1-d do nosso CPU é de 32kB. Da maneira que o array está ordenado, a cache L1 carrega 4

planos (completos). Ao iterar em k não se aproveita localidade. Mudando a ordem dos ciclos para k,j,i, sem unrolling, diminuise o número de cache misses por 1/4.

Para tirar partido de localidade temporal usa-se loop blocking, onde se divide a matriz 3D de (42x42x42)floats em blocos com um tamanho menor que o da cache L1 (32kB). Os blocos precisam de tamanho divisor de 42. Usamos um #block_size de 6 porque 7 é primo (impossibilidade de unroll) e blocos de 14 sobrecarregam a L1.

Isto permite-nos manter um número reduzido de misses e aumentar o CPI.

C. *SIMD*

De forma a tirar proveito da vetorização deve-se usar algumas flags na compilação (`-ftree-vectorize -msse4`). Na função `lin_solve` não se conseguiu encontrar uma forma de vetorizar. Porém na função `advect` existem dependências de controlo que podem ser removidas e usando uma flag como `-funroll-loops` permitimos a vetorização.

Otimização	Com flags para vetorização				Sem flags para vetorização			
	Tempo (s)	GFLOPs	#I (G)	#CC (G)	Tempo (s)	GFLOPs	#I (G)	#CC (G)
-O0	14.1	9.2	87.7	45.2	14.0	9.3	87.6	45.1
-O2	5.8	9.5	15.6	18.2	6.6	9.3	17.5	21.1
-O3	5.7	9.5	15.5	18.3	5.8	9.5	15.8	18.3

TABLE I: Tabela de desempenho com e sem flags para vetorização.

Uma versão como `-O3` já ativa a vetorização sem as flags e `-O0` não foi possível vetorizar o código. A versão `-O2` realiza algumas otimizações ao nível dos loops que ajudam a aplicar a vetorização. Com as flags, ela foi capaz de aplicar "vetorização de 128 bits".

III. Multi-threading

A. *Solver Red/Black*

Nesta fase foi utilizada uma versão modificada do algoritmo, mantendo a sua lógica, com um solver diferente. Este novo solver é uma versão red/black da função `lin_solve`, esta abordagem divide os cálculos em dois conjuntos de células, identificadas como red (`((1+i+j)%2 == 0)`) e black (`((i+j)%2 == 0)`). Esta divisão garante que cada célula seja atualizada utilizando apenas os valores mais recentes disponíveis, fazendo assim uma otimização comparada ao `lin_solve` anterior, de tal forma que assegura que nenhuma célula red seja diretamente vizinha de outra red, e o mesmo para as células black.

Para ambos os conjuntos de células, é atualizado o cálculo de `x[IX(i,j,k)]` de uma forma iterativa que considera os valores vizinhos. Antes da atualização do valor `x[IX(i,j,k)]`, guardamos o mesmo em uma variável `old_x`, calculamos a diferença absoluta entre o novo e o antigo valor e guardamos na variável `max_c`.

Estas iterações ocorre dentro do loop `while`, que continua até que a solução (`max_c`) convirja ou o número máximo de iterações seja atingido.

Este novo `lin_solve` resolve parte da redução de dependências, simplificando a lógica que a função anterior tinha.

B. *1ª Abordagem*

Numa primeira abordagem, procuramos paralelizar todos os loops onde as iterações são independentes umas das outras e onde não existe escrita em variáveis/posições acessadas em outras iterações. Apenas os loops `for` da função `lin_solve` apresentaram "data races", por isso aplicamos a diretiva `#pragma omp parallel for` nos loops das outras funções do código.

Nesta fase, identificamos que a localidade espacial dos dados estava impactando negativamente o desempenho. Para resolver isso, alteramos a ordem dos loops dimensionados em i,j,k para k,j,i. Essa mudança garante que o loop na dimensão i se torne mais interno, uma vez que as posições ao longo de i estão mais próximas na memória. Essa reorganização melhorou o acesso aos dados e reduziu a latência.

Após implementar essas otimizações, realizamos testes variando a quantidade de threads de 10 a 20. Observamos que o melhor desempenho foi alcançado com 15 threads, resultando em um tempo de execução de 8,253 segundos.

C. *2ª Abordagem*

Ao iniciar a função `lin_solve`, foram identificadas três variáveis que sofrem "race conditions". Entre elas, estão a variável `max_c` e outras duas que são usadas nos ciclos, com seus valores sendo alterados a cada iteração. Declarar essas variáveis fora da região paralela faz com que todas as threads utilizem os mesmos endereços de memória, criando a possibilidade de conflitos ("data races"), especialmente quando o objetivo é executar ciclos em paralelo.

A primeira etapa para resolver esse problema foi tornar essas variáveis, exceto `max_c`, privadas para cada thread utilizando a cláusula `private`. No entanto, uma solução mais eficiente foi declarar as variáveis localmente, apenas quando atribuindo seus valores, reduzindo o esforço do OpenMP para gerir o paralelismo (ver Anexo 2).

A variável `max_c`, por ser utilizada no controle do ciclo, requer atualizações consistentes entre threads ao final da região paralela. Como o objetivo é garantir que `max_c` contenha sempre a maior diferença entre os valores antigo e

novo de `x[IX(i,j,k)]`, utilizamos uma redução com a operação `max`. Essa abordagem assegura que o maior valor de `max_c` entre todas as threads seja atribuído à variável compartilhada fora da região paralela. A diretiva utilizada foi:

```
#pragma omp parallel for
reduction(max:max_c)
```

Aplicamos essa otimização ao longo da dimensão k nos dois ciclos `for` da função `lin_solve`. Com essa implementação, o melhor desempenho registado foi de 1,181 segundos utilizando 19 threads.

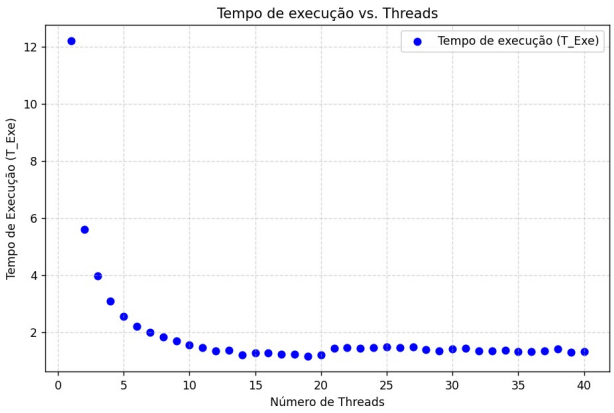


Fig. 2: Gráfico Texe/#Threads

D. Abordagem Falhada

Após a implementação da abordagem anterior, tentamos ampliar o paralelismo do código utilizando a diretiva `task` do OpenMP.

Funções como `advect`, `add_source` e `diffuse` são chamadas três vezes com três arrays diferentes dentro da função `vel_step`, o que as torna, em teoria, independentes entre si. Para reduzir o overhead associado ao paralelismo em funções menores, decidimos aplicar o paralelismo apenas à função `diffuse`, que é a mais ”pesada” das três.

Após realizar as alterações, executamos o código, mas não observamos diferenças significativas no desempenho. Pelo contrário, em média, o tempo de execução aumentou.

Utilizando ferramentas como `perf`, identificamos que o overhead causado pelo paralelismo superou os benefícios esperados. Essa situação pode ter ocorrido devido ao paralelismo aninhado, uma vez que a função `diffuse` faz chamadas para `lin_solve`, que já está paralelizada. Outra explicação, talvez mais plausível, é que a execução da função `diffuse` não representa uma parte significativa do tempo total da execução, oque reduz o impacto do paralelismo nessa região do código.

E. Resultados

Testamos o desempenho de um código variando o número de threads usadas e verificamos que o maior ganho de desempenho não ocorre ao usar o maior número de threads.

Observou-se que o melhor desempenho é obtido com cerca de 18 threads, sugerindo que o uso de hyper-threading (onde o processador cria threads lógicas adicionais) em números superiores a 20 threads pode levar a uma perda de eficiência. Isso ocorre porque, ao ultrapassar esse limite, há uma sobrecarga do processador, que começa a competir pelos mesmos recursos entre várias threads, resultando em latências e em um uso ineficiente da CPU. Além disso, com mais threads, a sincronização entre elas se torna mais desafiadora, o que pode aumentar a instabilidade, pois existe uma maior probabilidade de uma thread se atrasar em relação às outras.

Para reduzir a variação de resultados obtida nas medições, foi utilizado o método k-best. Este método tem como objetivo selecionar os melhores tempos de execução, descartando os valores atípicos e garantindo um resultado mais estável.

Para aplicar o k-best, o processo começa com a realização de um número mínimo de 10 execuções e um máximo de 20. Durante essas execuções, a condição predominante é que pelo menos três dos tempos de execução devem estar dentro de uma margem de 5% do valor mínimo de execução observado. Caso essa condição seja atendida, o valor final é calculado como a média entre o tempo mínimo e os tempos dentro dessa margem de 5%. Isso ajuda a suavizar as flutuações nos tempos de execução e a fornecer um resultado que seja mais representativo do desempenho real.

Se a condição de pelo menos três tempos dentro da margem de 5% não for atendida, o número de execuções é aumentado progressivamente até que a condição seja satisfeita. Esse processo de ajuste permite garantir que os tempos considerados para o cálculo de média sejam suficientemente consistentes e precisos.

Ao final, a média dos tempos de execução dentro da margem é tomada como o valor final, o que resulta em um tempo de execução mais confiável e representativo.

Com o uso do método k-best obteve-se um tempo médio de

1.246 segundos.

IV. Aplicação em GPU

Uma das grandes vantagens do uso de GPUs é a capacidade de explorar paralelismo em massa. Enquanto a abordagem tradicional de multi-threading com OpenMP é limitada ao número de núcleos do processador e está sujeita a um overhead significativo, as GPUs oferecem uma alternativa mais escalável. Na GPU, o trabalho pode ser dividido em tarefas menores e distribuído entre threads que operam em unidades de processamento (Processing Units - PUs) mais simples e leves que os núcleos de uma CPU. Essa simplicidade no design das PUs permite à GPU alcançar uma densidade muito maior de threads ativas.

Diferente da CPU, em que o hardware é responsável por diversas otimizações para execução eficiente, na GPU o controle dessas otimizações recai sobre o programador. Cabe a ele organizar e equilibrar o trabalho, gerir memória e explorar as características específicas da arquitetura GPU.

Neste contexto, o CUDA (Compute Unified Device Architecture) surge como uma plataforma e modelo de programação desenvolvidos pela NVIDIA que facilita a criação de aplicações paralelas para GPUs.

A. *Implementação*

Para correr o código na GPU é crucial diminuir cópias de memoria entre CPU e GPU, para tal o melhor é alocar os arrays na GPU e fazer todo o processamento de dados lá. Quando obtivermos os arrays finais na GPU devemos então fazer a cópia para a CPU para serem utilizados. Depois de criar alocação na GPU com `CudaMalloc` devemos criar um kernel para todas as funções que armazenam dados.

1) ***`add_source`****:* Este kernel foi implementado para adicionar a função diretamente na GPU, em vez de usar um loop na CPU, que seria menos eficiente devido à execução sequencial, a computação foi distribuída em threads na GPU, permitindo que cada thread atualize um índice específico do vetor de forma simultânea. O tamanho do bloco foi definido como 512 threads. Apesar de o bloco suportar 1024 threads 512 foi melhor porque permitimos uma ocupação de SM com mais blocos. Como os acessos são contínuos podemos usar mais threads por bloco que a busca de dados à memoria global não é prejudicada.

2) ***`set_bnd`****:* A função divide o processo em três kernels (loop1, loop2 e loop3). Fizemos isto para garantir que são executados sequencialmente porque os conjuntos de loops têm dependências entre si, isto também permite dividir com blocos e threads diferentes.

As dimensões foram escolhidas tendo em conta os pares dg, tg e dy, ty que preenchem o array ao mesmo tempo que melhoram a localidade dos acessos à memoria global. Apesar da GPU não possuir cache, os acesso à memoria global são otimizados se as posições que o wrap acede são continuas, logo usar num bloco mais threads em x onde as posições são continuas é melhor.

No loop2 foram usadas dimensoes diferentes pois ele acede a posições em j e k logo nunca a posições continuas na memoria após alguns testes verificamos que usar 4 threads para j e 4 para k era a melhor abordagem. Esta divisão garante que otimizemos o poder de processamento paralelo da GPU.

3) ***`lin_solve`****:* A função mais prejudicial do código possui um verificação de convergência que assim como em multi-threading precisa de ser reduzida entre as threads que possuem a variável local. Esta redução precisa ser implementada manualmente pois não existe uma função CUDA que faça a redução.

a) ***aplicação GPU****:* Criando então 2 kernels um para as posições red outro para as black e implementando reduções neles obtemos no final o resultado do array x e um array com o numero de blocos usados (só podemos partilhar dados para a redução dentro do mesmo bloco o que resulta na redução a uma posição por bloco) que precisa ser reduzido a um único valor.

Para finalizar a redução são usados 2 kernel extra. Como o nosso numero de blocos é maior que 1024 precisa-se de mais que um kernel para a redução. Cria-se então um kernel `reduce_to_block` que faz a redução desde o array que se obteve com os kernel `red/black` até um outro array mais pequeno que pode ser processado por um unico bloco no kernel `reduce_max` e assim obter o valor final reduzido com a operação `max`. Usamos um algoritmo de redução eficiente, mas que requer o numero de threads por bloco seja potencia de 2.

Os kernel `red/black` juntos processam toda a grelha mas cada um só processa metade das posições em i (x) usando um mapeamento de índices para nao deixar threads ociosas (`unsigned int i = 2 * (threadIdx.x + blockIdx.x * blockDim.x) + 1 + (k + j) % 2;`) quanto a j,k é suficiente usar o mapeamento normal +1 (`threadIdx.z + blockIdx.z * blockDim.z + 1;`) apos os mapeamentos utilizamos o calculo de x como na versão sem CUDA e após o calculo adicionamos `change` na shared memory para fazer a redução dentro do bloco.

b) ***shared memory****:* Após a construção do kernel é interessante explorar otimização ao nível da shared memory uma vez que são feitos muitos acessos na memoria global para um único armazenamento. Para aplicar shared memory vamos analisar uma janela (4x4x4), que para facilitar a notação será o primeiro bloco da grelha de blocos onde os ids locais

coincidem com os ids globais (ver Anexo 3). Percebe-se olhando para a janela quais as posições a serem processadas.

De forma a perceber os acessos ao array x desenhamos um esquema que mostra as posições acedidas para cada calculo, que são os vizinhos, ou seja (i-1,j,k);(i+1,j,k);(i,j-1,k);(i,j+1,k);(i,j,k-1);(i,j,k+1).Realizamos um esquema (ver Anexo 4) para vizualizar os acessos, marcamos posições a varias cores para perceber como proceder à adição de valores na shared memory.

Verificamos que uma das posições é repetida em muitos acessos e foi marcada a verde (corresponde ao vizinho i-1). Marcamos a azul-claro todas as posições que já foram adicionadas a verde, isto ocupou uma boa parte dos acessos. Adicionando apenas as posições verdes ficamos com aproximadamente 73% das posições necessárias na shared memory, apenas precisamos agora de lidar com as fronteiras.

Observou-se mais posições que se repetiam e não tinham sido marcadas, após marcar a laranja uma das posições acedidas pelo último calculo em i (vizinho i+1) e amarelo onde estas se podiam reutilizar ficamos sem mais posições repetidas por colorir.

Procurou-se padrões e pintamos posições com varias outras cores. Nos primeiros cálculos em k=1 precisamos adicionar as posições coloridas a vermelho (vizinhos k-1). Nos últimos cálculos para k=4 (no caso) precisamos adicionar as posições a preto (vizinhos k+1). Sempre que j=1 precisamos adicionar as posições a roxo (vizinhos j-1). Sempre que j=4 (no caso) precisamos adicionar as posições a branco (vizinhos j+1). Após isto ficamos com 100% do elementos na shared memory realizando apenas 37.5% dos acessos à memoria global.

Precisamos agora arranjar uma forma de mapear o array da shared memory. Após varias tentativas/erro chegamos a um array continuo onde a posição (0,0,0) é o índice 0, a posição (2,0,0) é o indice 1 e assim por diante a forma de mapear as 3D (i,j,k) no array é: `(i>>1) + (j*(blockDim.x+1))+ (k*(blockDim.x + 1)*(blockDim.y + 2))` (ver Anexo 5)

4) ***advect****:* . Cada thread processa uma ponto específico usando (i,j,k) o que permite cálculos independentes e simultâneos. O kernel usa o mesmo algoritmo que na versão sem CUDA para calcular os valores nas posições intermediárias, mas sem os loops, pois paralelizamos as dimensões.

5) ***project****:* Em par com o algoritmo anterior, a função project foi dividida em duas partes, e paralelizamos cada uma separadamente, pois a kernel project2 tem dependência de dados com a project1.

Tentamos implementar o uso de shared memory no kernel project2 com o objetivo de otimizar o desempenho, reduzindo acessos repetitivos à global memory. Apesar de a implementação estar correta em termos de funcionalidade, o tempo de execução piorou, passando de 1,34% para 2,43%. Embora a shared memory tenha sido corretamente utilizada, fatores como o uso frequente de __syncthreads() e os cálculos de fronteira, ou até mesmo o fato de que, realizamos três stores diretamente na global memory para utilizar u,v e w, sejam o suficiente para tornar o ganho do desempenho inviável para essa função. Por isso, a otimização pretendida não foi alcançada

B. Otimizações globais

Ao alocar o array na GPU primeiramente foram usadas as dimensões da grade M,N,O, mas observou-se que estas dimensões nem sempre são divisíveis por blockDim.x, blockDim.y, blockDim.z. Se alocarmos diretamente o array com as blockDims observamos melhorias provavelmente pela forma como os dados ficam agora alinhados garantindo uma posição para cada thread.

A nivel de compilador estudamos a possibilidade de reduzir o máximo de registos usados pelo kernel. Usando a opção `-v` do compilador *ptxas* obtemos informação da compilação dos kernels sendo 40 o numero máximo de registos usados em um kernel. Como usamos 128 threads por bloco somos capazes de ter 16 blocos na mesma SM (2048 max threads). Porem com 40 registos por thread somos limitados a 12 blocos. O ideal seria 32 registos por thread ($2048 * 32 = 65536$).Vamos usar `-maxrregcount=32`, mesmo tendo removido algumas variáveis desnecessárias do kernel e ter verificado que o ptxas compilou naturalmente com no máximo 32 registos. Desta forma garantimos com segurança a maior ocupação da GPU.

C. Análise e testes

Quando aumentamos o tamanho dos arrays aumentamos o erro na soma final (*total_density*) devido à utilização de floats precisão simples. Para prevenir o erro decidimos usar doubles na soma final do array densidade.

Para o `SIZE 168` o código foi testado várias vezes o tempo de execução oscilou muito entre 12.8-14.5s (ver Anexo 6). Como se conseguiu obter várias vezes resultados entre os 12.8-13s assumimos um tempo de execução de 12.854s, que foi o menor obtido.

1) ***CLUSTER (Tesla K20)****:* De forma a comparar a escabilidade das versões OMP e CUDA do nosso algoritmo definimos um conjunto de *size inputs*. Como usamos dimensão de bloco (32,2,2) decidimos que era bom usarmos múltiplos de 32 e 2. A partir de `SIZE 168` (padrão) decidimos usar o próximo múltiplo de 32 e ir aumentando de 32 em 32 até chegar a 2×168.

Na análise anterior do código OpenMP foi usada a versão 11.2.0 do gcc ao contrário desta análise com 9.3.0. Por isso que não consideramos o tempo anteriormente obtido para `SIZE 84`.

SIZE	CUDA	OMP	total_density
84	3.073s	2.218s	140881
168	12.854s	18.553s	160890
192	19.927s	29.751	160148
224	34.228s	54.958s	159017
256	49.351s	1m5.604s	158214
288	1m15.255s	>2m	157062
320	1m33.86s	>2m	154304
336	1m54.492s	>2m	151843

TABLE II: Resultados

Fig. 3: Scability

Como esperado a nosso código OMP não escala tao bem como a aplicação em GPU. Uma das causas é o aumento do trabalho por thread gerando possiveis problemas nos acessos à memória e overhead associado ao scheduling entre outras funções OMP. Enquanto na GPU conseguimos sempre manter a mesma organização do bloco e o padrao escolhido de acessos à memoria, mesmo que as otimizações a nivel hardware sejam piores (sem uso automatico de cache). O overhead em GPU também é menor devido à simplicidade e foco dos componentes, possuindo muitos registos, um bom escalonador e PUs simples.

Na GPU aumentamos o numero de blocos, mantendo as 128 threads por bloco. Devido ao suporte de 2048 threads por SM conseguimos ter aproximadamente 16 blocos na mesma SM com 13 SMs conseguimos 208 blocos ”ao mesmo tempo” na GPU (em condições ideais, não dependendo de shared memory e registos).

NOTA: não são executados exatamente todos ao mesmo tempo(SM limitada a 193 CUDA cores) mas estão a ser abordados ao mesmo tempo de maneira a realizar o schedule das tarefas pelos cores

Enquanto nas threads CPU o hardware/openMP é mais complexo ficamos sujeitos a maior overhead, além do overhead de scheduling que vai ser maior tendo em conta o maximo de blocos que conseguimos executar em simultaneo.

Para `SIZE 84` a versão OMP tira vantagem, muito devido ao não preenchimento total da GPU ficando apenas com 28 blocos para processar. Nesta situação OMP lida melhor com o trabalho pois ainda existe pouco trabalho a distribuir e o overhead é pequeno tirando vantagem das otimizações a nivel de memoria presentes no CPU.

D. GeForce RTX 3050 TI

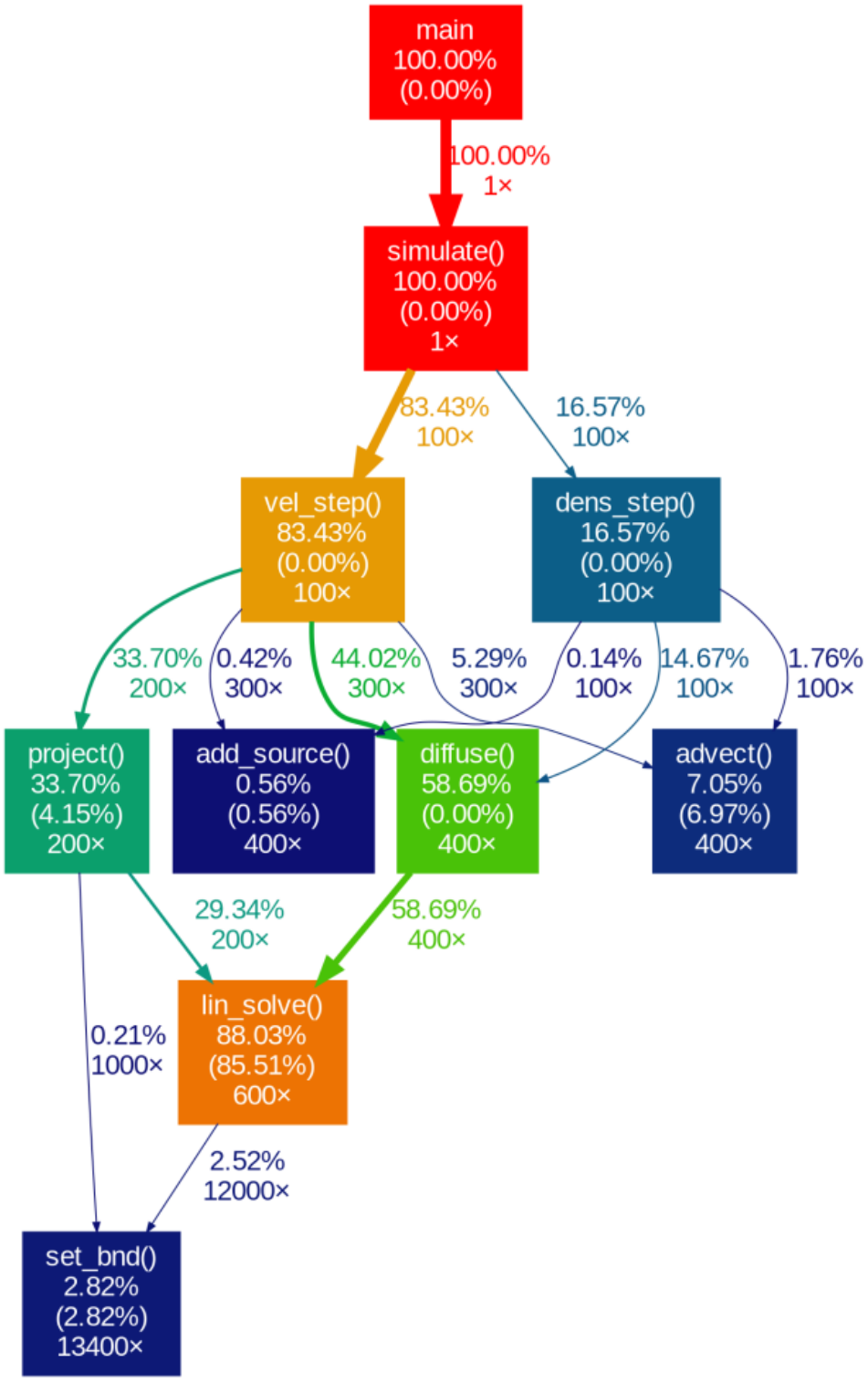
Através da tabela verificamos que o código está otimizado também para uso em outras GPUs.

SIZE	CUDA	total_density
84	2,3s	140881
168	7,47s	160890
192	10,63s	160148
224	16,76s	159017
256	24,88s	158214
288	36,52s	157062
320	51,36s	154304
336	58,7s	151843

TABLE III: Resultados

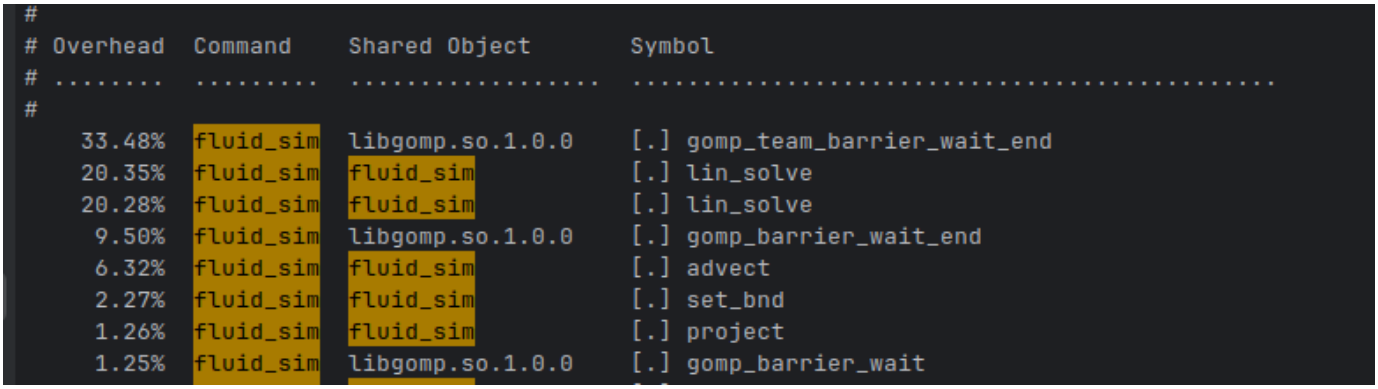
ANEXO 1

Call Graph



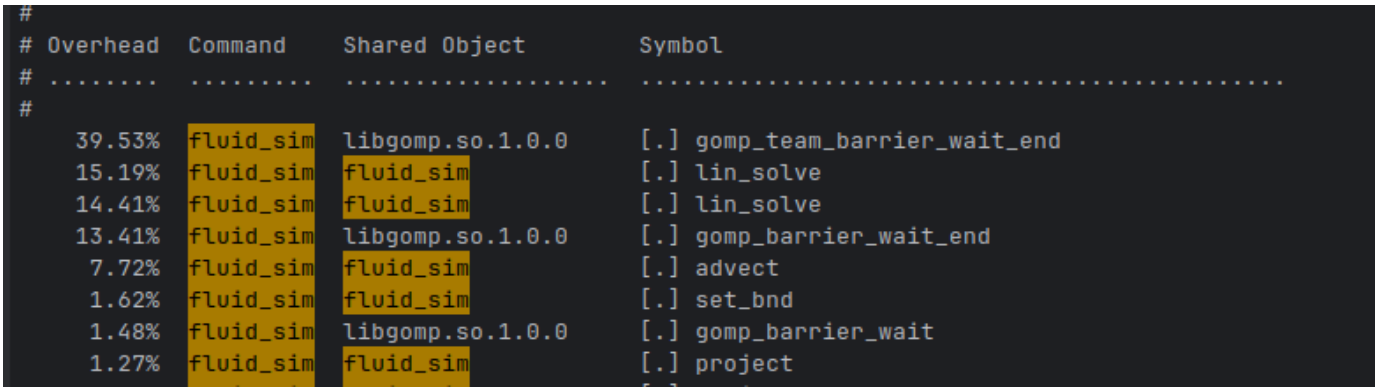
ANEXO 2

Comparação entre o overhead das funções openmp usando private e declarando as variaveis no momento em que lhe atribuimos valores:



```
#
# Overhead  Command    Shared Object       Symbol
# ........  .........  ..................  ..........................................
#
    33.48%  fluid_sim  libgomp.so.1.0.0    [.] gomp_team_barrier_wait_end
    20.35%  fluid_sim  fluid_sim           [.] lin_solve
    20.28%  fluid_sim  fluid_sim           [.] lin_solve
     9.50%  fluid_sim  libgomp.so.1.0.0    [.] gomp_barrier_wait_end
     6.32%  fluid_sim  fluid_sim           [.] advect
     2.27%  fluid_sim  fluid_sim           [.] set_bnd
     1.26%  fluid_sim  fluid_sim           [.] project
     1.25%  fluid_sim  libgomp.so.1.0.0    [.] gomp_barrier_wait
```

Fig. 4: Sem a clause `private`



```
#
# Overhead  Command    Shared Object       Symbol
# ........  .........  ..................  ..........................................
#
    39.53%  fluid_sim  libgomp.so.1.0.0    [.] gomp_team_barrier_wait_end
    15.19%  fluid_sim  fluid_sim           [.] lin_solve
    14.41%  fluid_sim  fluid_sim           [.] lin_solve
    13.41%  fluid_sim  libgomp.so.1.0.0    [.] gomp_barrier_wait_end
     7.72%  fluid_sim  fluid_sim           [.] advect
     1.62%  fluid_sim  fluid_sim           [.] set_bnd
     1.48%  fluid_sim  libgomp.so.1.0.0    [.] gomp_barrier_wait
     1.27%  fluid_sim  fluid_sim           [.] project
```

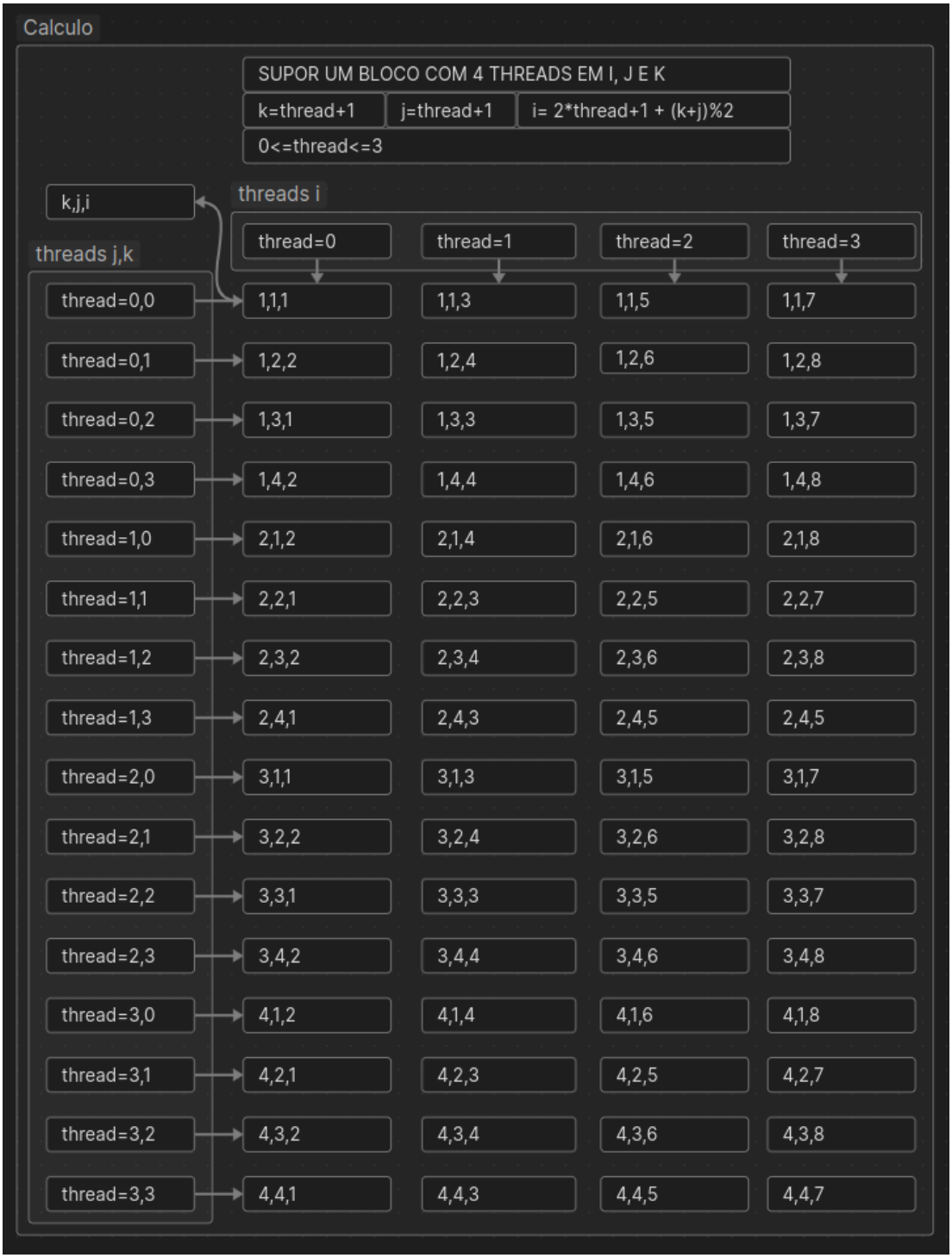
Fig. 5: Com a clause `private`

Na coluna ”Shared Object” identificamos os objetos referentes ao openmp (libgomp.so.1.0.0), na primeira coluna podemos ver o overhead. Verificamos uma redução de aproximadamente 10% no overhead do openmp apenas ao remover a clause private.

Nota: A função `lin_solve` consome agora menos percentagem de ciclos de relogio, mas não implica que consuma menos ciclos em valor absoluto. Os ciclos totais aumentaram devido ao overhead do openmp, então uma percentagem menor pode corresponder ao mesmo numero de ciclos.

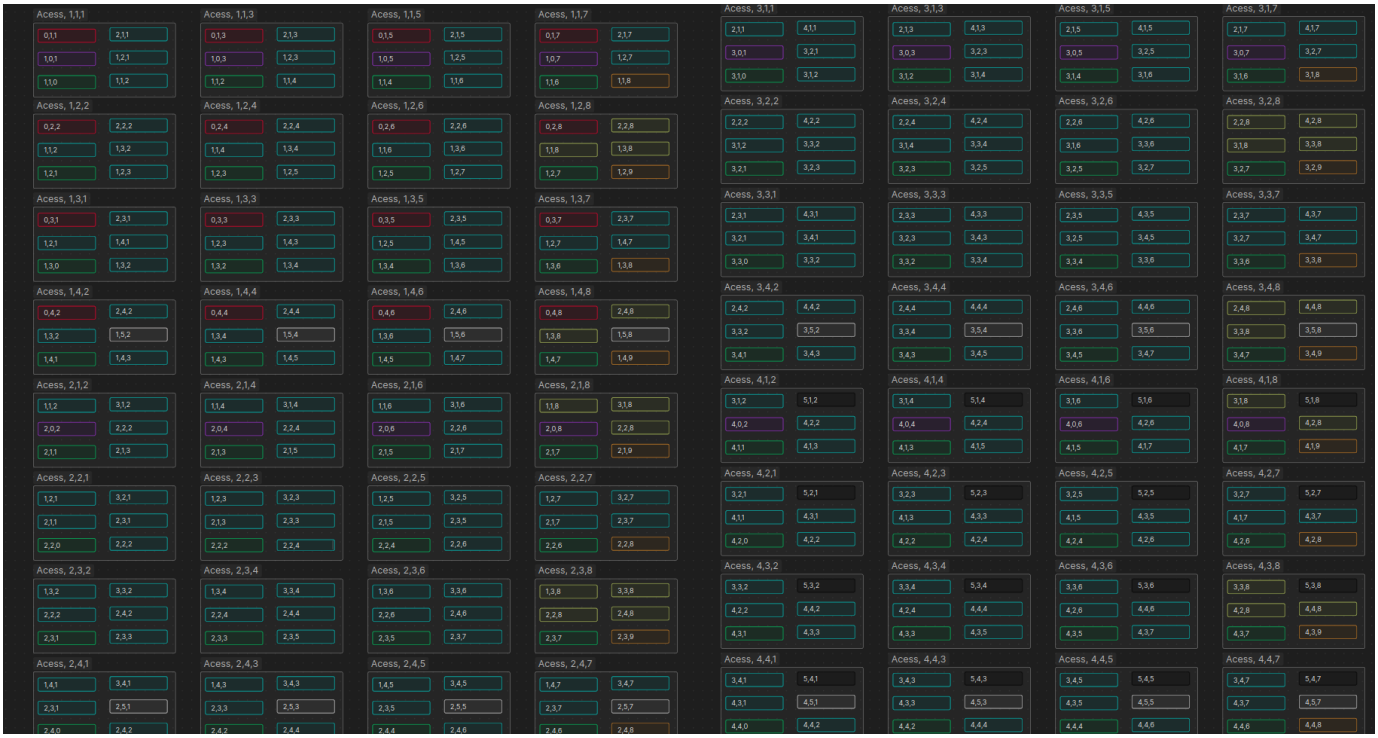
ANEXO 3

Janela 4x4 a ser processada



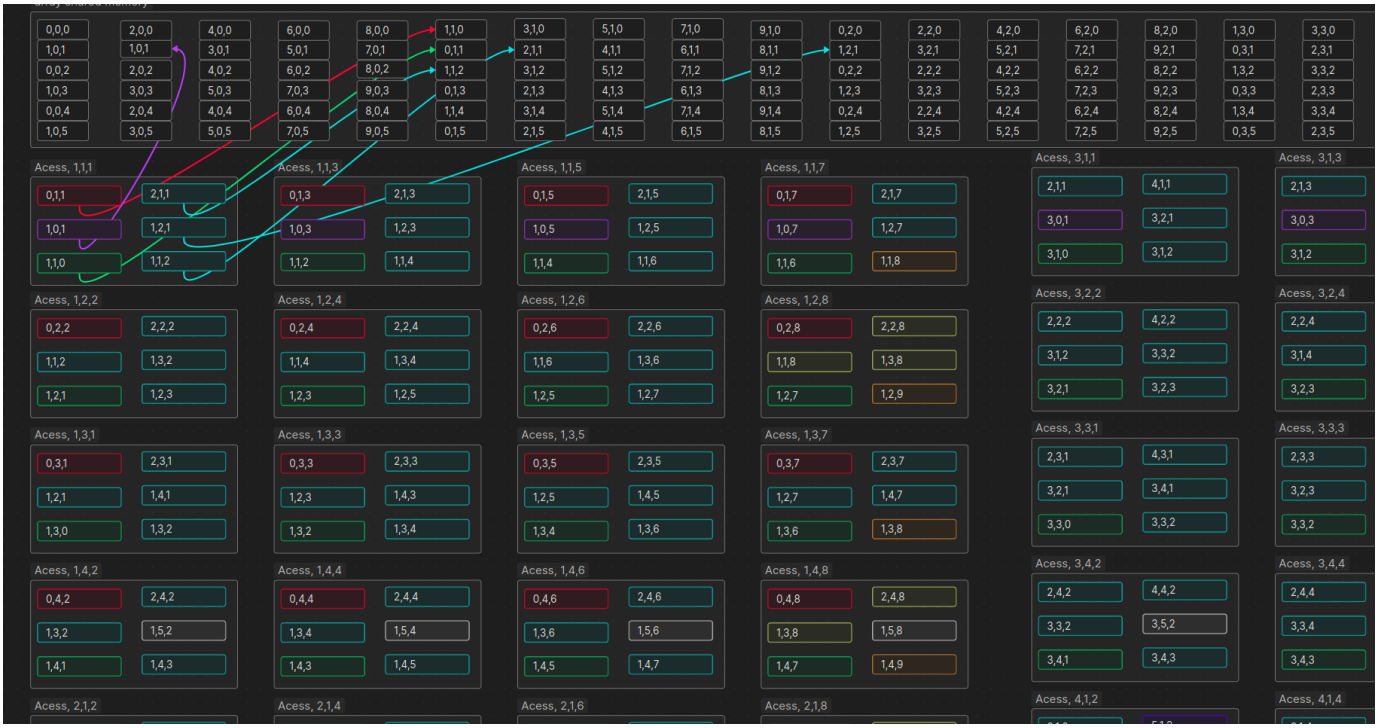
ANEXO 4

Acessos realizados por cada thread da janela.



ANEXO 5

Busca de dados à shared memory



ANEXO 6

Testes SIZE=168

```
Total density after 100 timesteps: 160890

real    0m14.551s
user    0m8.640s
sys     0m4.752s
Total density after 100 timesteps: 160890

real    0m14.441s
user    0m8.676s
sys     0m4.670s
Total density after 100 timesteps: 160890

real    1m14.184s
user    0m8.564s
sys     0m4.750s
Total density after 100 timesteps: 160890

real    0m12.905s
user    0m8.677s
sys     0m4.220s
Total density after 100 timesteps: 160890

real    0m42.911s
user    0m8.498s
sys     0m4.402s
Total density after 100 timesteps: 160890

real    0m12.895s
user    0m8.730s
sys     0m4.163s
~
```

```
Total density after 100 timesteps: 160890

real    0m12.854s
user    0m8.502s
sys     0m4.283s
~
~
~
~
~
~
~
```

NOTA: tempos superiores a 14.551s devido a suspensão de trabalho pelo slurm