

Turma 2, Série 6, EDDP em 2+1 dimensões com CUDA

Grupo 25, MEFT, 2013 /2014

Rui Marques 75969

Sofia Freitas 76025

23 de Janeiro de 2014

Resumo

Nesta série pretende-se resolver a equação de Gross-Pitaevskii fazendo vídeos do código em C++ e depois adaptado para programação em paralelo – OpenMP e CUDA. Para tal começamos por estudar a biblioteca de complexos de C++ e por criar uma nossa, na qual nos baseámos para efectuar os restantes programas da série.



Grupo 25, MEFT, 2013 / 2014 Rui Marques 75969 Sofia Freitas 76025

De notar que, por motivos de comodidade de compilação (devido ao uso de bibliotecas) criámos também um Makefile^a, que compila todos os programas presentes na série (excepto o de CUDA – g25s6c4a.cu) de uma só vez, incluindo os de OpenMP. Todas as funções relativas à bilbioteca de complexos estão definidas em *compz.h/compz.cpp*.

O comando a usar no terminal é simplesmente:

1 \$ make

A compilação tem o seguinte resultado esperado:

```
ruimarques@beta:~/Dropbox/FC\>> make
for fname in g25s6c1a ; do g++ -Wall -g -ggdb -O3 -lexif -lglut -fopenmp -c $fname.cpp ; done
for fname in g25s6c2a compz ; do g++ -Wall -g -ggdb -O3 -lexif -lglut -fopenmp -c $fname.cpp ; done
for fname in g25s6c3a compz ; do g++ -Wall -g -ggdb -O3 -lexif -lglut -fopenmp -c $fname.cpp ; done
for fname in g25s6c3a2 compz ; do g++ -Wall -g -ggdb -O3 -lexif -lglut -fopenmp -c $fname.cpp ; done
for fname in g25s6c3b compz ; do g++ -Wall -g -ggdb -O3 -lexif -lglut -fopenmp -c $fname.cpp ; done
for fname in g25s6c1b compz ; do g++ -Wall -g -ggdb -O3 -lexif -lglut -fopenmp -c $fname.cpp ; done
g++ g25s6c1a.o -lm -fopenmp -o p1a
g++ g25s6c2a.o compz.o -lm -fopenmp -o p2a
g++ g25s6c3a.o compz.o -lm -fopenmp -o p3a
g++ g25s6c3a2.o compz.o -lm -fopenmp -o p3a2
g++ g25s6c3b.o compz.o -lm -fopenmp -o p3b
g++ g25s6c1b.o compz.o -lm -fopenmp -o p1b
rm -f *~ *.o g25s6c1a
rm -f *~ *.o g25s6c2a compz
rm -f *~ *.o g25s6c3a compz
rm -f *~ *.o g25s6c3a2 compz
rm -f *~ *.o g25s6c3b compz
rm -f *~ *.o g25s6c1b compz
```

Figura 1: *Resultado esperado da compilação dos programas com recurso à Makefile. A compilação divide-se em 3 partes, divisão esta evidente na figura: 1 – compilação individual, 2 – linkagem, 3 – remoção de ficheiros auxiliares.*

O programa g25s6c4a.cu foi compilado no servidor disponibilizado da forma indicada por e-mail pelo professor da cadeira.

1 \$. cudaenv
2 \$ nvcc g25s6c4a.cu -o g25s6c4a

^aO código do Makefile, bem como todos os headers e códigos auxiliares apresentam-se em Anexo.

1 Contentores e computação paralela

1.a Vantagem de usar templates e contentores da Standard Template Library C++. Exemplificação para números complexos

Um contentor (container) é um objecto que armazena um conjunto de outros objectos – elementos do container. Os containers são implementados como templates de classes, o que permite uma muito maior flexibilidade dos tipos suportados como elementos.

Nos containers verifica-se uma gestão do espaço reservado aos elementos, e a estes estão associadas funções que permitem o acesso – propriedades semelhantes às dos ponteiros.

Os containers têm várias funções membro em comum, partilhando funcionalidades. A decisão de que tipo de container utilizar passa, portanto, não só pela sua complexidade, mas também pela sua eficiência no código em questão.

De notar que os adaptadores de contentores não são classes por si só, nem constituídas apenas por containers. São classes que possibilitam uma interface específica com base num objecto de um dos containers da classe – por exemplo *list* – para gerir os elementos. A programação é feita de tal modo que os elementos do container podem ser acedidos pelos membros do adaptador independentemente da classe de contentor usada.

Nesta alínea é-nos então pedido que desenvolvamos um código que exemplifique o uso da STL para números complexos, incluindo templates e containers.

Mostramos, de seguida, um print do terminal com o código em funcionamento, para a parte dos complexos.

```

complexo c1
real(c1) = 1
im(c1) = -2
abs(1,-2) = 2.23607
exp(1,-2) = (-1.1312,-2.47173)
conj(1,-2) = (1,2)
log(1,-2) = (0.804719,-1.10715)
norma(1,-2) = 5
sqrt(1,-2) = (1.27202,-0.786151)

complexo c2
real(c2) = 3
im(c2) = 5
Abs(3,5) = 5.83095
Exp(3,5) = (5.69751,-19.2605)
Conj(3,5) = (3,-5)
log(3,5) = (1.76318,1.03038)
Norma(3,5) = 34
sqrt(3,5) = (2.1013,1.18974)

soma: (1,-2) + (3,5) = (4,3)
diferenca: (1,-2) - (3,5) = (-2,-7)
            (3,5) - (1,-2) = (2,7)
produto: (1,-2) * (3,5) = (13,-1)
divisao: (1,-2) / (3,5) = (-0.205882,-0.323529)
            (3,5) / (1,-2) = (-1.4,2.2)

```

Figura 2: Resultado da implementação do código g25s6c1a.cpp – complexos.

Relativamente aos containers, escolhemos usar um container tipo list, com o seguinte output no terminal:

```

Uso de containers: LIST
list1: 0 1 2 3 4 5
list2: 5 4 3 2 1 0

list1: 0 1 2 3 4 5
list2: 4 3 2 1 0 5

list1: 0 1 2 3 4 4 3 2 1 0 5 5
list2:

```

Figura 3: Resultado da implementação do código g25s6c1a.cpp – list.

Trata-se de um código relativamente simples que cria duas listas, inserindo valores em ambas, que são impressos. Posteriormente, o código faz com que o primeiro elemento de cada lista passe para o início; e depois há a junção das duas listas na primeira.

Aqui mostramos a definição do container em si:

```

12 void printLists (const list<int>& l1, const list<int>& l2)
13 {
14     cout << "list1:" ;
15     copy (l1.begin(), l1.end(), ostream_iterator<int>(cout, " "))
16     ;
17     cout << endl << "list2:" ;
18     copy (l2.begin(), l2.end(), ostream_iterator<int>(cout, " "))
19     ;
20     cout << endl << endl;
21 }
```

Listing 1: Impressão das listas.

1.b Computação paralela – openmp

A computação paralela surge como uma alternativa de programação mais eficiente e mais rápida que a forma clássica que temos utilizado até agora. De facto, com a computação paralela, estamos a maximizar as potencialidades do computador de tal modo que cada core, ou cada processador, fica encarregado da execução de uma dada tarefa específica.

Covenhamos que temos um programa que necessita de vários cálculos antes de chegar a um resultado final. Em vez de esses cálculos iniciais serem feitos sequencialmente, passam a ser feitos em simultâneo, por *cores* diferentes, poupando-se tempo de computação. O programa fica assim mais eficiente no sentido de que o código se torna mais rápido.

De modo a ilustrar a computação paralela, criámos um programa que apresenta um menu inicial que permite ao utilizador escolher que "subprograma" deseja explorar:

```
Escolha a opcao desejada
1: Numero de processadores e threads
2: Calculo dos numeros primos
```

Figura 4: *Menu inicial do código g25s6c1b.cpp.*

Cada um destes subprogramas ilustra um utilização diferente de openmp para computação paralela, como se explica seguidamente.

O primeiro subprograma implementa as funções

- `omp_get_num_procs()`
- `omp_get_num_threads()`
- `omp_get_max_threads()`

Que dão, respectivamente, o número de processadores, de threads em utilização e de threads máximo do computador.

O segundo subprograma trata-se de uma adaptação a um programa de contagem de primos desenvolvido o ano passado na cadeira de Programação. Assim,

pelo crivo de Atkin, encontramos os primos até um dado N que o utilizador estabelece, fazendo uma contagem dos mesmos.

De modo a ilustrar a maior eficiência associada à computação paralela, corremos este último programa com o comando `time` no terminal, tendo comparado o tempo que este programa demorava a correr normalmente, com OpenMP, e com as linhas de computação paralela comentadas. De notar que para ambas as simulações usámos $N = 1000000$.

Ilustramos de seguida os tempos com e sem computação paralela, respectivamente nas Figuras 5 e 6:

<code>real</code>	<code>0m2.653s</code>
<code>user</code>	<code>0m0.184s</code>
<code>sys</code>	<code>0m0.004s</code>

Figura 5: *Runtime do programa dos números primos com computação paralela.*

<code>real</code>	<code>0m3.484s</code>
<code>user</code>	<code>0m0.124s</code>
<code>sys</code>	<code>0m0.008s</code>

Figura 6: *Runtime do programa dos números primos sem computação paralela.*

Nota: O programa foi corrido num computador com *dual core*.

2 Overload de operadores e teste do código

2.a Classe para complexos

Para esta alínea construímos um código com overload de operadores para números complexos. Note-se que a nossa biblioteca de números complexos encontra-se definida em *compz.cpp* e *compz.h*, apresentados em Anexo.

O código pede ao utilizador que insira a parte real e a parte imaginária de um certo complexo c1, executando as seguintes operações referentes a esse complexo:

- Módulo ou norma
- Ângulo do argumento
- Conjugado
- Exponencial
- Logaritmo
- Raiz quadrada

Seguidamente, após repetir o processo para um segundo complexo, c2, executa as seguintes operações entre os dois complexos:

- Soma
- Diferença
- Produto
- Divisão

2.b Comparação dos resultados de ambos os códigos

Aqui comparamos os valores obtidos com os códigos desenvolvidos nas alíneas 1a e 2a, mostrando de seguida um print do terminal com os mesmos números complexos:

$$c_1 = 1 - 2i$$

$$c_2 = 3 + 5i$$

Apresentamos de seguida o print do terminal para a implementação do código g25s6c1a.cpp (Figura 7):

```
complexo c1
real(c1) = 1
im(c1) = -2
abs(1,-2) = 2.23607
exp(1,-2) = (-1.1312,-2.47173)
conj(1,-2) = (1,2)
log(1,-2) = (0.804719,-1.10715)
norma(1,-2) = 5
sqrt(1,-2) = (1.27202,-0.786151)

complexo c2
real(c2) = 3
im(c2) = 5
Abs(3,5) = 5.83095
Exp(3,5) = (5.69751,-19.2605)
Conj(3,5) = (3,-5)
log(3,5) = (1.76318,1.03038)
Norma(3,5) = 34
sqrt(3,5) = (2.1013,1.18974)

soma: (1,-2) + (3,5) = (4,3)
diferenca: (1,-2) - (3,5) = (-2,-7)
            (3,5) - (1,-2) = (2,7)
produto: (1,-2) * (3,5) = (13,-1)
divisao: (1,-2) / (3,5) = (-0.205882,-0.323529)
            (3,5) / (1,-2) = (-1.4,2.2)
```

Figura 7: Resultado da implementação do código g25s6c1a.cpp.

Com os mesmos números complexos exemplificamos agora a implementação do código g25s6c2a.cpp (Figura 8):

```

complexo c1
real(c1) = 1
im(c1) = -2
mod(1-2i) = 2.23607
arg(1-2i) = -1.10715
conjugado(1-2i) = 1 +2i
exp(1-2i) = 3.99232 +6.21768i
log(1-2i) = 0.804719-1.10715i
sqrt(1-2i) = 0.66874-1.33748i

complexo c2
real(c2) = 3
im(c2) = 5
mod(3 +5i) = 5.83095
arg(3 +5i) = 1.03038
conjugado(3 +5i) = 3-5i
exp(3 +5i) = -0.00667052 +0.000950859i
log(3 +5i) = 1.76318 +1.03038i
sqrt(3 +5i) = 1.24237 +2.07062i

soma: 1-2i + 3 +5i = 4 +3i
diferença: 1-2i - (3 +5i) = -2-7i
            3 +5i - (1-2i) = 2 +7i
produto: (1-2i) * (3 +5i) = 13-1i
divisao: (1-2i) / (3 +5i) = -0.205882-0.323529i
          (3 +5i) / (1-2i) = -1.4 +2.2i

```

Figura 8: Resultado da implementação do código g25s6c2a.cpp.

Note-se que os valores calculados são precisamente os mesmos para ambos os programas, o que nos mostra que definimos bem as nossas biblioteca e classe de complexos.

Por outro lado, podemos ainda observar que a forma de mostrar os complexos é diferente. Isto porque na biblioteca standard de C++ um complexo é mostrado da seguinte forma:

`Complexo = (x, y)`

E no nosso programa definimos o display de complexos da forma natural matemática:

`Complexo = x + yi`

Em que x e y representam respectivamente a parte real e a parte imaginária de um número complexo.

3 Equação de Gross-Pitaevskii

3.a Equação de Gross-Pitaevskii

Nesta alínea foi-nos pedido que desenvolvessemos um código capaz de resolver a equação diferencial de Gross-Pitaevskii a d = 2+1 dimensões – equação de Schrödinger não linear para condensados de Bose Einstein, a duas dimensões espaciais e uma temporal:

$$(i - \gamma) \frac{\partial \psi}{\partial t} = -\frac{1}{2} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \psi + \frac{x^2 + y^2}{2} \psi + G |\psi|^2 \psi - i\Omega \left(x \frac{\partial}{\partial y} - y \frac{\partial}{\partial x} \right) \psi \quad (1)$$

De notar que a função $\psi = \psi(x, y, t)$ é complexa e requer normalização. Essa normalização é da seguinte forma:

$$\psi(x, y, t) \rightarrow \frac{\psi(x, y, t)}{\sqrt{N}} \quad (2)$$

Com norma

$$N = \int \int \bar{\psi}(x, y, t) \psi(x, y, t) dx dy \quad (3)$$

Passamos a explicar como resolvemos o problema proposto, fazendo acompanhar os excertos de código relevantes.¹

Para a resolução da equação, são-nos dados parâmetros iniciais, que nos permitem trabalhar adimensionalmente:

$$\gamma = 0.01$$

$$\Omega = 0.85$$

$$G = 1000$$

Criámos ainda uma rede de 128×128 pontos nas dimensões espaciais, correspondente a uma caixa de dimensões $[-10, 10] \times [-10, 10]$, e com as seguintes

¹Lembrar que todos os códigos se encontram em Anexo.

condições fronteira:

$$\begin{aligned}\psi_{0,j}^n &= 0.0 \\ \psi_{n_{x-1},j}^n &= 0.0 \\ \psi_{i,0}^n &= 0.0 \\ \psi_{i,y-1}^n &= 0.0\end{aligned}$$

O que corresponde, fisicamente, a ter um potencial nulo nos limites da caixa.

Consideramos ainda a condição incial em $t = 0$:

$$\psi_{i,j}^0 = 1.0 \quad (4)$$

Quando $0 < i < n_x - 1$ e $0 < j < n_y - 1$, ou seja, para toda a extensão da caixa excepto nos seus limites, em que o potencial é nulo.

Passamos agora ao cálculo das derivadas de primeira e de segunda ordem nas duas dimensões espaciais:

$$\frac{\partial \psi}{\partial x} = \frac{\psi_{i+1,j}^n - \psi_{i-1,j}^n}{2\Delta x} \quad (5)$$

$$\frac{\partial \psi}{\partial y} = \frac{\psi_{i,j+1}^n - \psi_{i,j-1}^n}{2\Delta y} \quad (6)$$

$$\frac{\partial^2 \psi}{\partial x^2} = \frac{\psi_{i+1,j}^n - 2\psi_{ij}^n + \psi_{i-1,j}^n}{(\Delta x)^2} \quad (7)$$

$$\frac{\partial^2 \psi}{\partial y^2} = \frac{\psi_{i,j+1}^n - 2\psi_{ij}^n + \psi_{i,j-1}^n}{(\Delta y)^2} \quad (8)$$

No tempo, temos uma equação diferencial da forma $\frac{\partial \psi}{\partial t} = F(\psi)$. Tendo em conta a equação (1), fica então definida a função F:

```

18 Complex F(Complex ax1, Complex ax2, Complex ay1, Complex ay2,
    Complex a, int i, int j){ //funcao F
19     Complex k(-g,1.);
20     Complex iomg(0.,-omega);
21     Complex h=c(2.*L/((double)N-1.));
22     Complex dx2 = (ax1+ax2-a-a)/(h*h);
23     Complex dy2 = (ay1+ay2-a-a)/(h*h);
24     Complex lap = dx2+dy2;
25     Complex dx = (ax1-ax2)/(h*c(2.));
26     Complex dy = (ay1-ay2)/(h*c(2.));
27     Complex x=c(-L)+c(i)*h;

```

```

28     Complex y=c(-L)+c(j)*h;
29
30     return (c(-0.5)*lap + c(0.5)*(xxx+y*y)*a + c(G*mod(a)-
31 }  


```

Listing 2: Definição da função F.

A equação diferencial no tempo pode ser resolvida com diferenças finitas avançadas, sabendo que esta equação em 1ª ordem é da forma:

$$\psi_{i,j}^{n+1} = \psi_{i,j}^n + \Delta t F(\psi_{i,j}^n) \quad (9)$$

Visto $\Delta t < 0.5\Delta x \Delta y$, temos que, pelo critério de CFL (Courant-Friedrichs-Lowy), que a equação converge.

Usando o Método de Runge-Kutta de 3ª ordem temos:

$$\psi_{i,j}^{(1)} = \psi_{i,j}^n + \Delta t F(\psi_{i,j}^n) \quad (10)$$

$$\psi_{i,j}^{(2)} = \frac{3}{4}\psi_{i,j}^n + \frac{1}{4}\psi_{i,j}^{(1)} + \frac{1}{4}\Delta t F(\psi_{i,j}^{(1)}) \quad (11)$$

$$\psi_{i,j}^{(3)} = \frac{1}{3}\psi_{i,j}^n + \frac{2}{3}\psi_{i,j}^{(2)} + \frac{2}{3}\Delta t F(\psi_{i,j}^{(2)}) \quad (12)$$

Correspondente, no código, a:

```

116     for (i=1;i<N-1;++i){ // implementacao do metodo
117         for (j=1;j<N-1;++j){
118             psi1[i][j]=psi[i][j]+c(dt)*F(psi[i+1][j],psi[i-1][j],psi[i][j+1],
119             +psi[i][j-1],psi[i][j],i,j);
120         }
121         for (i=1;i<N-1;++i){
122             for (j=1;j<N-1;++j){
123                 psi2[i][j]=c(0.75)*psi[i][j]+c(0.25)*psi1[i][j]+c(0.25*dt)*F(
124                 psi1[i+1][j],psi1[i-1][j],psi1[i][j+1],psi1[i][j-1],psi1[i][j],
125                 +j,i,j);
126             }
127             for (i=1;i<N-1;++i){
128                 for (j=1;j<N-1;++j){
129                     psi[i][j]=c(1.0/3.)*psi[i][j]+c(2.0/3.)*psi2[i][j]+c((2.0/3.)*dt)*F(psi2[i+1][j],psi2[i-1][j],psi2[i][j+1],psi2[i][j-1],psi2[i][j],i,j);
130                 }
131             }
132         }
133     }

```

Listing 3: Implementação do método de Runge-Kutta de 3^a ordem.

Finalmente, procedemos à normalização apresentada em (2) :

```

131     sum=0.;
132     for (i=1;i<N-1;++i){ //calculo de N para a normalizacao
133         for (j=1;j<N-1;++j){
134             sum+=h*h*mod( psi[ i ][ j ]) *mod( psi[ i ][ j ]) ;
135         }
136     }
137
138     for (i=1;i<N-1;++i){ //normalizacao
139         for (j=1;j<N-1;++j){
140             psi[ i ][ j ] = psi[ i ][ j ]/ c( sqrt( sum )) ;
141         }
142     }

```

Listing 4: Cálculo de N (*sum*) seguido de normalização.

De modo a conseguir concretizar o programa pedido, verificámos uma lista interessante de erros, que foram detectados e corrigidos. De notar que esta detecção passou em grande parte por observação do vídeo resultante e comparação com o resultado esperado. A criação do vídeo nesta etapa, apesar de não pedido, foi um auxiliar importante para uma melhor análise da correção do código.

Numa fase inicial de desenvolvimento do código, começámos apenas plotar uma imagem com recurso ao gnuplot, ao fim de um certo tempo por nós escolhido. Deixando o programa correr durante o referido período de tempo, íamos observando a evolução do plot, sendo assim mais fácil a detecção de possíveis erros.

Passamos a listar os erros que nos foram aparecendo, fazendo-os acompanhar pelas imagens ilustrativas respectivas, bem como correções efectuadas.

Na figura que se segue (Figura 9), verificámos que estavamos a aplicar o método de Runge-kutta pela ordem errada. No código, a aplicação do método, a normalização e a impressão eram feitas no mesmo bloco de ciclo.

Associado a isso, tínhamos passos a serem feitos primeiro numas células que noutras, o que causou assimetria, incluindo a cor mais escura do lado direito.

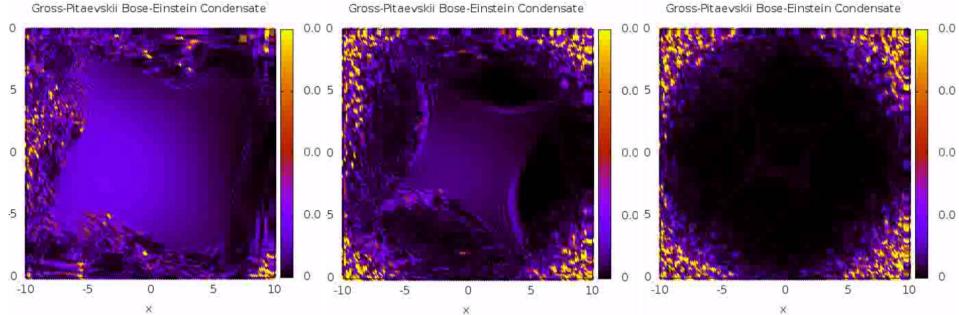


Figura 9: Assimetria; não sincronismo temporal no código

Este conjunto de imagens seguinte (Figura 10) ilustra uma combinação das condições anteriores com o facto de as células não estarem bem inicializadas, i.e., as condições fronteira não estavam bem definidas:

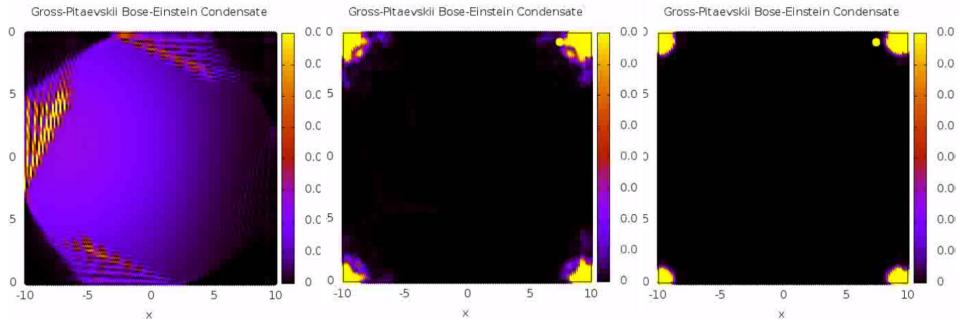


Figura 10: Condições fronteira mal definidas; ausência de autoscale

Aqui (Figura 11) verifica-se exactamente o mesmo que acima (Figura 10), só que agora foi usado autoscale. A partir de um determinado ponto no tempo no teste de cima, o vídeo ficava completamente negro, isto porque a escala não se adaptava ao potencial que efectivamente se verificava. Com o comando autoscale no gnuplot este problema foi corrigido.

De notar que a assimetria que se continua a verificar aqui, e resulta da tal ordem incorrecta das contas.

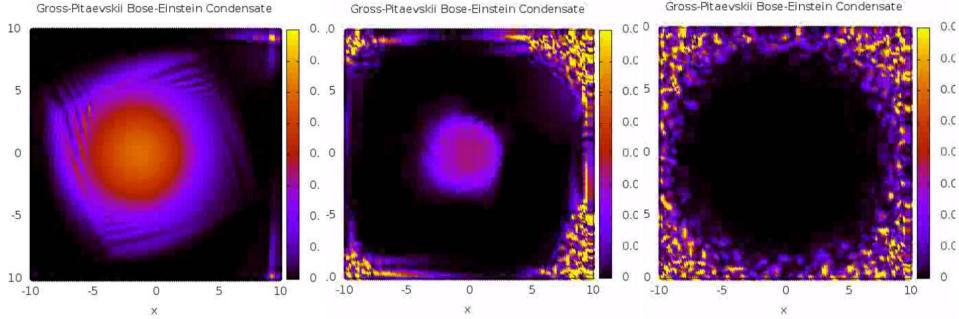


Figura 11: *Método de Runge-Kutta aplicado pela ordem incorrecta*

Neste print que se segue (Figura 12) vê-se um ponto preto, no canto superior direito. Este ponto tem que ver com a legenda do gnuplot. Após termos dado o comando `notitle` o ponto desapareceu. Se observar com atenção, poderá verificar que este ponto aparece em todos os outros plots mostrados anteriormente – especial destaque para a última imagem (da direita) da Figura 10, em que se observa este mesmo ponto a amarelo.

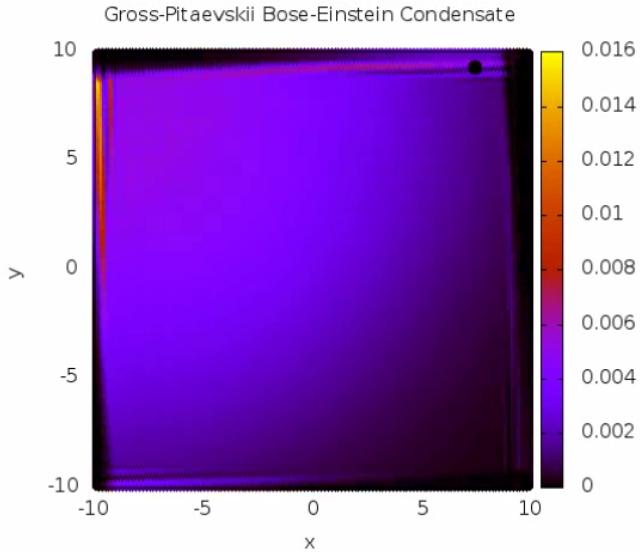


Figura 12: *Legenda gnuplot*

Passamos agora ao problema que nos demorou mais tempo a corrigir. Tendo já verificado o código de modo a garantir a correcta implementação do método de Runge-Kutta de 3^a ordem – pela ordem correcta e com as operações relevantes em ciclos separados –, continuámos a verificar um problema de simetria. Este

problema, no entanto, não era em todo o vídeo mas sim a partir de um certo tempo. Atente na Figura 13, que ilustra o referido problema de simetria:

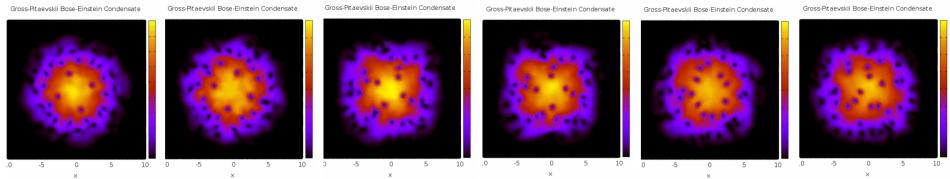


Figura 13: *Momento da perda de simetria do plot num teste ao programa g5s6c3a.cpp.*

Como é possível observar pelos frames sucessivos mostrados acima, há um instante em que há perda da simetria. É natural que, por erros de truncatura, o programa vá acumulando erros e aconteça algo semelhante a esta situação de perda de simetria. No entanto, há formas de colmatá-la, sendo que a que nós encontrámos foi realmente colocar todos os processos do método em ciclos diferentes, garantindo a correcta aplicação do método de Runge-Kutta de 3^a ordem, e pela ordem de operações correcta – inicialização, Runge-Kutta e normalização. Notámos, ainda, que para condições de tempo diferentes, o programa aguentava a simetria durante mais ou menos tempo.

De notar que no fim há a tendência a aparecer simetria triangular (Figura 14) em detrimento da quadrada verificada antes – confrontar com secção 4.b.1.

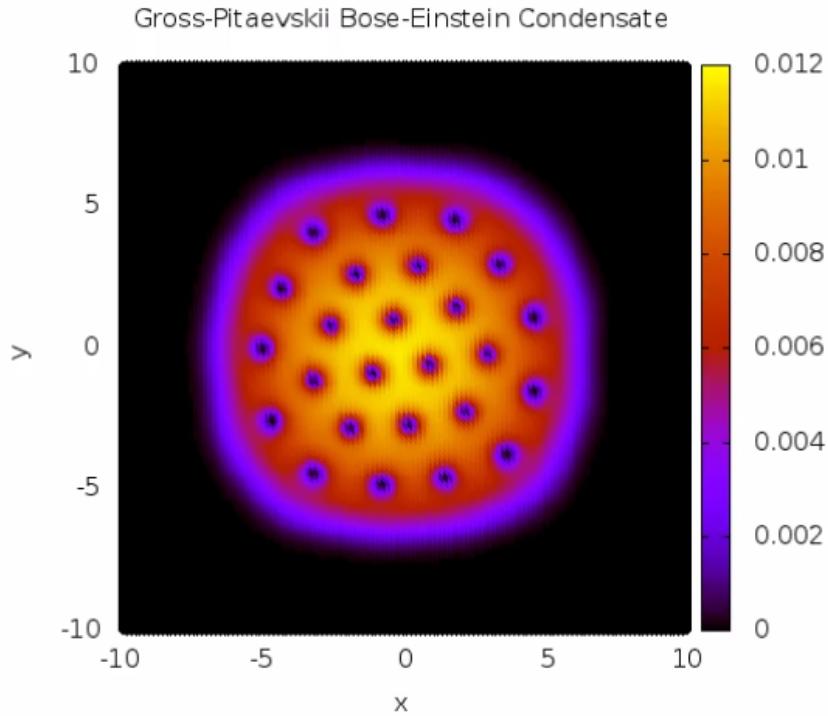


Figura 14: Simetria triangular na fase de estabilidade.

Mostramos, de seguida, algumas features do programa para as quais gostaríamos de chamar a atenção.

Barra de progresso

Para acompanhar o processo de execução do código, criámos uma progress bar:

```

152   if (!(n%(pop*cpf))){                                // progress bar
153     v=((double)pop*cpf)/(((double)clock()-(double)t)/
154       CLOCKS_PER_SEC);
155     if (v==0) v=1;
156     cout << "\r" << n*100./nmax << "%ETA=" << "f=" << v << " "
157     cyc/sec ETA=" << (int)((((double)nmax-(double)n)/v)
158       /60.) << "min=" << (int)((double)nmax-(double)n)/v)
159     -60*(int)((((double)nmax-(double)n)/v)/60.) << "sec=";
160     cout << "\n[";
161     for (i=0;i<100*n/nmax;++i) cout << " ";
162     for (i=0;i<100-100*n/nmax;++i) cout << " ";
163     cout << "]";

```

```

160     cout << flush ;
161     cout << "\e[A\r";
162 }
163 }
164 if (!(n%(pop*cpf))){ //actualizacao de v e pop
165     t=clock();
166     if (v<100.)
167         pop=((int)v)/cpf;
168     if (pop==0) pop=1;
169 }
```

Listing 5: Progress bar no código g25s6c3a.cpp.

A dificuldade de criação de uma barra de progresso prende-se com o reescrever de algo na mesma linha do terminal. Assim, naturalmente, a cada instrução nossa de cout para o terminal, essa impressão seria feita numa linha nova.

No entanto, isso não nos interessa numa progress bar, que deve ir aumentando à medida que o programa se aproxima do seu término; nem tão pouco numa linha que descreva o estado de execução do programa, que deve também ser reescrita a cada actualização de output.

De modo a contornar esta situação, recorremos ao seguinte:

- \r – força a escrita no início da linha actual do terminal
- \e[A – escape sequence para a maior parte dos terminais linux que faz com que se vá para a linha de cima no terminal
- flush – actualização da stream, ou seja, do output do terminal

Mostramos, na Figura 15, um exemplo ilustrativo do funcionamento da barra:

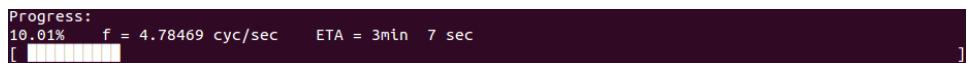


Figura 15: Funcionamento da progress bar no código g25s6c3a.cpp.

Criação do script

De modo a compilar o vídeo, incluímos a criação de um script dentro do próprio programa:

```

102     ofstream script("../script3a"); // criacao do script
103     script << "setterm -blank" << endl;
```

```

104     script << "set output \\"..../frame" << n/cpf << ".png\""
105     << endl;
106     script << "set title \"Gross-Pitaevskii-Bose-Einstein"
107     Condensate\"" << endl;
108     script << "set xrange[-10:10]" << endl;
109     script << "set yrange[-10:10]" << endl;
110     script << "set cbrange[0:*)" << endl;
111     script << "set size square" << endl;
112     script << "set xlabel \"x\"" << endl;
113     script << "set ylabel \"y\"" << endl;
114     script << "set xlabel \"\psi*(\psi*)\""
115     script << "plot \\"..../data3a\" using 1:2:3 notitle wpu
116     pt7ups2lt palette" << endl;
117     script.close();

```

Listing 6: Criação do script3a no código g25s6c3a.cpp.

Este script é reescrito a um intervalo de tempo definido pelo utilizador – cpf (ciclos por frame) – sendo que, de cada vez que é reescrito, guarda um plot do gnuplot para os dados referentes a esse instante de tempo. Este plot será um frame do vídeo. De notar que o ficheiro de dados é aberto *a priori* e reescrito da mesma forma que o script; após aquisição do frame não nos interessa manter o ficheiro de dados e por isso é que este pode ser reescrito sem qualquer problema (de perda de dados).

Note-se que é necessário escrever um novo script para cada frame de modo a podermos guardar todos os frames com nomes diferentes.

Quando o processo chega ao fim, após decorrido o intervalo de tempo total que o utilizador definiu, o vídeo é compilado através do compilador avconv.

Após a compilação do vídeo estar completa, todos os frames que serviram para a referida compilação são removidos, através do comando rm:

```

190 if (!system("rm \\"..../frame*.png")){ // eliminacao frames
    apois render do video

```

Listing 7: Linha do código g25s6c3a.cpp para a eliminação dos frames.

É ainda de notar que o utilizador é que escolhe o nome a atribuir ao vídeo (que fica com a extensão .mp4).

Finalmente, salientamos que a criação do vídeo foi feita com o compilador *avconv*, cuja instalação se processa da seguinte forma – no terminal, escrever as seguintes linhas:

```
1 $ apt-get install libav-tools  
2 $ sudo apt-get install avconv
```

A compilação genérica de um vídeo directamente no terminal é feita de acordo com o seguinte comando:

```
1 $ avconv -r fps -i input%d.png -r fps output.mp4
```

Em que *fps* é o número de frames por segundo desejado; os frames ficam guardados na forma de *input<#frame>.png*, sendo o resultado da compilação apresentado em *output.mp4*.

Nota: Apresentamos um código alternativo *g25s6c3a2.cpp* (em Anexo) em todo semelhante a *g25s6c3a.cpp* à excepção do render do vídeo, que não existe: ausência de script, o programa escreve os dados num ficheiro de texto.

3.b Adaptação do código para OpenMP

Por forma a adaptar o código para OpenMP, é primeiro necessário planeá-lo na medida de ver o que é que pode realmente ser passado para computação paralela (o que pode ser feito em simultâneo) e o que é que nos convém que seja mantido em série, de forma sequencial.

Pela alínea anterior, são executados 5 passos sequenciais (3 de Runge-Kutta, 1 para calcular a norma, 1 para normalizar). Porém, em cada passo, o cálculo de cada célula da matriz pode ser feito de forma não sequencial, isto porque não há uma dependência em valores que mudam noutras células nesse ciclo (método aplicado em modo "Jacobi"). Logo, cada passo de $(N - 2)^2$ ciclos pode ser feito de forma independente por threads diferentes. Assim, podemos distribuir os ciclos de cada passo pelo número de threads existentes no computador, aumentando assim o tempo de execução.

O ciclo de geral *for* que controla o tempo corrido, é obvia e necessariamente sequencial, não podendo ser dividido em várias threads, visto cada itereada depender directamente dos valores anteriores.

Assim, basta adicionar a linha `#pragma omp parallel for collapse(2)`, por cima de cada *for* exterior de cada passo. A clause *collapse(2)* indica que há um *for* dentro do *for* exterior.

Porém, para o ciclo que cálcula a norma, há que ter em conta o facto de haver uma variável que guarda o valor do somatório, e que é comum aos 2 ciclos. Por essa razão temos de adicionar a clause *reduction(+:sum)* para indicar que *sum* vai sofrer a operação de adição por todos os threads, para que não haja uma sobreposição do resultado obtido pelo thread mais lento.

Na alocação de espaço para ψ , na sua inicialização e na sua libertação, também foi implementado OpenMP, apesar de ser bastante irrelevante, visto que o tempo tomados nestas operações é bastante desprezável face ao resto do programa.

De notar que as operações de escrita não devem ser implementadas em OpenMP, visto que foi testado que o acesso às *output streams* por mais de um thread pode ser mais lento do que com apenas 1 thread.

Também devemos ter em conta, que caso mantenhamos a criação de frames com *gnuplot* no programa, o OpenMP não será muito mais rápido, pois a passagem frequente de um thread para mais threads e vice-versa, cada vez que é criado um frame (processo demoroso), atrasa o ciclo de tempo. Logo se queremos mesmo comparar os tempos de execução do método com e sem OpenMP, temos de descartar o script e a criação de frames.

Para esse efeito, foi criado o programa *g25s6c3a2.cpp*, que corresponde à versão sem OpenMP e sem criação de frames. O programa *g25s6c3b.cpp* corresponde à versão com OpenMP sem criação de frames. Na comparação dos tempos de execução, obteve-se o resultado presente na Figura (16).

```
Progress:
97.6562%    f = 27.3973 cyc/sec    ETA = 0min 0 sec
[ |██████████| ] 100% completed

real    0m9.408s
user    0m19.093s
sys     0m1.808s
rui75969@localhost:~/Dropbox/FC$ time p3a2
gamma = 0.01
Omega = 0.85
G = 1000
N = 128
L = 10
h = 0.15748
dt = 0.00195312
tf = 1
# cycles = 512
cycles per frame = 20
# frames = 25
Progress:
97.6562%    f = 28.5714 cyc/sec    ETA = 0min 0 sec
[ |██████████| ] 100% completed

real    0m18.883s
user    0m16.085s
sys     0m1.924s
```

Figura 16: *Comparação dos tempos de execução (sob as mesmas condições) de g25s6c3b.cpp e g25s6c3a2.cpp respectivamente no Intel QUAD core 2.3*

O tempo de execução relevante é o REAL, que como se pode observar é 2 vezes maior no programa sem OpenMP. Repare-se que não se pode esperar um tempo de execução 4 vezes menor com OpenMP só por haver 4 cores no CPU, visto que a própria biblioteca de OpenMP apresenta certas ineficiências não dependentes da implementação no código. Outro factor que contribuiu para a razão dos tempos de execução não ser maior é o factor de nem todos os ciclos serem não sequências como já foi indicado, e o ciclo de escrita numa única thread. Considerando estes factores, a razão obtida é bastante satisfatória, e em como este programa pode ser de longa execução compensa de facto usar OpenMP.

Note-se que acima da barra de progresso, no número de ciclos por segundo (indicador de velocidade de execução), apesar de com OpenMP aparecer um valor ligeiramente menor, tal deve-se ser um facto de no ciclo em que foi imprimido o valor no *cout*, a computação passar a usar só um thread, ou seja, nesse, ciclo a frequência é a mesma que sem OpenMP (e até ligeiramente maior, devido à transição). Por essa razão o tal mostrador de frequência de ciclos não é indicado para programas com OpenMP.

4 CUDA

4.a Adaptação para Global Memory

Nesta alínea pretende-se adaptar o código desenvolvido em *g25s6c3a.cpp* para CUDA. Antes de mais, convém perceber qual a diferença entre CUDA e OpenMP.

OpenMP é um tipo de programação paralela que recorre ao processador – CPU –, dividindo o processamento pelos vários threads. Por outro lado, CUDA utiliza placa gráfica – GPU –, sendo que no caso da resolução da equação de Gross-Pitaevskii, cada há um thread do GPU para cada ponto da matriz, havendo por isso $(N - 1)^2 = 127^2 = 16129$ threads a correr em paralelo, acelerando substancialmente o programa.

Tendo em conta as especificações dos computadores, será esperado que CUDA demore muito menos tempo que OpenMP.

Para tal, irá se recorrer a um tipo de memória do GPU, denominada Global Memory, na qual todos os dados nesta memória estão acessíveis a todos os threads independentemente da sua localização (grid e block).

O número de threads necessário é $(N - 1)$ em por eixo. Para garantir a eficiência da computação, pretendemos ocupar o máximo de cada bloco. O máximo de threads por bloco na arquitectura disponível (sem exceder o número de threads máximo por SM -1024) é de $16^2 = 512$ threads, o que implica que em cada eixo teremos $\frac{N-1}{16} = \frac{127}{16}$ o que não dá um número de blocos inteiro. Se deixarmos, os cálculos por aqui, ficariamos com 7 blocos por eixo, ou seja, 112 threads por eixo, restante por isso 15 threads.

Por essa razão, o teremos de adicionar 16 ao número de threads por eixo pretendidos, ficando $\frac{N-1+16}{16} = \frac{143}{16} = 8$, pois estamos a dividir números inteiros havendo truncatura à casa das unidades.

Resumindo, o número de threads por eixo será de 16×16 , e o número de blocos por grid será 8×8 , havendo por isso $(168)^2 = 16384 > 16129$. Temos por isso, um número de threads em excesso, apesar de estar minimizado ao necessário.

O programa *g25s63a.cpp* teve de sofrer algumas alterações para estar bem implementado em CUDA. Nomeadamente, foram criadas cópias em CUDA de todas os arrays ψ , usando as funções *cudaMalloc* (para alocar espaço no GPU) e *cudaMemcpy* (para cópia host to device). Foi também criado o array Normc, com o mesmo tamanho dos anteriores, destinado a guardar a contribuição para o integral de normalização de cada célula. São estas variáveis (presentes na Global Memory) que vão ser usadas no processamento paralelo. Juntamente com Normc, foi criado

Normc2, um ponteiro CUDA (função *cast* da biblioteca *Thrust*) para Normc.

No programa anterior, é fácil ver que como é usado Runge-Kutta de ordem 3, existem 5 passos principais num ciclo de tempo (3 de RK, um para calcular a norma, e outro para normalizar cada entrada). Cada um desses passos foi transformado numa função kernel (k1,k2,k3,k4,k5). Em cada uma dessas funções, define-se os índices usados como IDs de threads, e depois protege-se os indices não existentes nos arrays usando um *if*, visto que há mais threads que o número de células dos arrays. As funções são depois instanciadas no ciclo *for* de tempo, com a sintaxe de CUDA *kn << bpg, tpb >> (arg1,...,);*.

No final de cada chamada da kernel, é necessário sincronizar os threads, usando a função *cudaThreadSynchronize()*.

Para a normalização, k4 preenche em cada célula de Normc a sua contribuição para o integral de normalização. Foi usada a função *reduce* da biblioteca Thrust para calcular a soma de todos os elementos de Normc, sendo necessário usar o *cast*, Normc2, devido aos requerimentos em termos de tipo de variável dos argumentos de *reduce*. As células são normalizadas com k5.

Para passar os dados de novo para a memória do CPU (device to host) é necessário usar de novo a função *cudaMemcpy*. Esta transferência de dados tem de ser minimizada em frequência de modo ao código se tornar mais eficiente. Para isso é necessário aumentar o valor da variável *cpf* (ciclos por frame) para um valor igual ou superior a 1000. Isto porque, não só minimiza o número de transferências, como é necessário à produção de frames de qualidade (ver secção 4 (b)).

Finalmente, após adaptação do código para CUDA, fizemos o memory check, não tendo obtido quaisquer erros na compilação do programa (Figura 17). Note-se que foi escolhido o servidor disponibilizado para compilar e executar o código produzido.

```
[grupo25@cidhcp126 ~]$ cuda-memcheck g25s6c4a
=====
 CUDA-MEMCHECK
gamma = 0.01
Omega = 0.85
C = 1000
N = 128
L = 10
h = 0.15748
dt = 0.0001
tf = 5
# cycles = 50000
cycles per frame = 1000
# frames = 50
About 2.19727 Mb will be used
Progress:
98%   f = 438.596 cyc/sec    ETA = 0min  2 sec
[                                                 ]
=====
 ERROR SUMMARY: 0 errors
```

Figura 17: Memory check do CUDA, para o programa *g25s6c4a.cu*

Executando o programa normalmente, obteve-se o resultado presente na Figura (18).

```
[grupo25@cidhcp126 ~]$ g25s6c4a
gamma = 0.01
Omega = 0.85
G = 1000
N = 128
L = 10
h = 0.15748
dt = 0.00195312
tf = 100
# cycles = 51200
cycles per frame = 1000
# frames = 51
About 2.24121 Mb will be used
Progress:
99.6094%    f = 2380.95 cyc/sec    ETA = 0min 0 sec
```

Figura 18: *Código g25s6c4a.cua executado no server disponibilizado*

Constata-se de forma clara que o código é bastante mais rápido, cerca de 100 vezes (comparando o número de ciclos por segundo deste programa com o da Figura (16)).

4.b Vídeo density plot

Como foi indicado na secção 3, o video já foi feito nos programas anteriores (C++ simples e OpenMP), usando o método já descrito. O mesmo método foi usado para o programa em CUDA. Não foi usado o script disponibilizado pois era necessário a instalação de bibliotecas que já não se encontram em repositórios activos de Ubuntu. O método adoptado, tem a vantagem de produzir os frames ao mesmo tempo que são feitos os cálculos, não sendo por isso perdido tempo adicional a executar um script secundário.

Como em CUDA o programa é extremamente rápido, para um valor baixo de *cpf*, o gnuplot vai desacelerar significativamente a performance do CUDA, pois com uma grande frequência de criação de frames, o programa não avança na sua execução até o gnuplot terminal o rendering do frame. Por essa razão, é aconselhado um valor de *cpf* igual ou superior a 100.

Criar directamente os frames em vez de ficheiros de texto tem uma grande vantagem que verificamos experimentalmente: os frames ocupam cerca de 40 kB cada, enquanto cada ficheiro de texto ocupa cerca de 400 kB (10 vezes mais). Se tivermos a produzir ficheiros de texto, iremos gastar imensa memória em disco, assim como se gastará bastante tempo depois a transferir todos os ficheiros de texto do servidor para o computador local, e de seguida a converter cada ficheiro numa imagem, e finalmente num video.

Como no servidor, não existe o pacote *libav-tools*, foi necessário descarregar os frames para os nossos computadores, e o video teve de ser codificado localmente. Para extrair os frames usámos o comando de sftp *get frame*.png*, sendo os frames convertidos em video com o *avconv* tal como indicado na secção 3 (a).

Tal como pedido no enunciado, o vídeo resultante da adaptação para CUDA encontra-se no Youtube, no link <http://www.youtube.com/watch?v=c7IYpMyz8qQ>, com o nome FC0M13S6T2G25_RuiSofia.

4.b.1 Interpretação física do vídeo

Por curiosidade, apresentamos aqui uma rápida introdução ao que é um condensado de Bose-Einstein e, particularmente, da equação de Gross-Pitaevskii, atribuindo um significado físico ao observado no vídeo.

De um ponto de vista da termodinâmica, no mesmo gás, as partículas comportam-se da mesma forma e em princípio podem ocupar certos estados (quânticos). Se

essas partículas forem fermões, não podem ocupar o mesmo estado quântico, pelo princípio de exclusão de Pauli. No entanto, se essas partículas forem bosões, não existe limitação quanto ao número de partículas a ocupar cada estado. Para uma dada configuração, estas partículas irão distribuir-se pelos níveis de energia dessa dada configuração, com uma ocupação preferencial dos estados de energia mínima para baixas temperaturas.

Em particular, para um conjunto de bosões, no limite em que a temperatura vai para zero, todas as partículas vão ocupar o estado de energia mínimo do sistema. Assim, para uma temperatura suficientemente baixa, a maioria das partículas encontram-se no mesmo estado quântico, e com a mesma velocidade. Desta forma, o conjunto de bosões comporta-se como um fluido macroscópico com novas propriedades – superfluidez.

A superfluidez manifesta-se, entre outros efeitos, pela ausência de viscosidade e pela criação de vórtices no líquido. Os vórtices têm propriedades quantizadas, i.e. a velocidade do líquido não pode tomar valores arbitrários.

Observe-se a criação de vórtices num output do nosso programa:

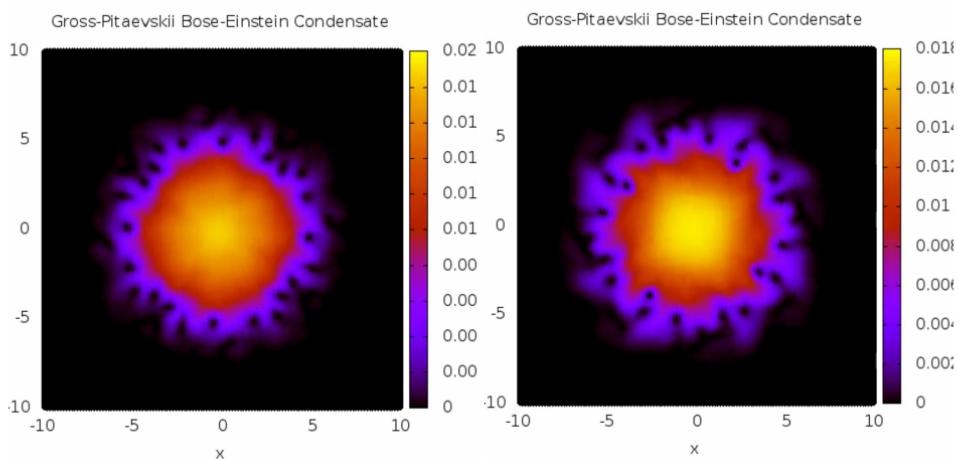


Figura 19: *Formação de vórtices na simulação do condensado de Bose-Einstein.*

A simetria mostrada pela simulação é inicialmente quadrada, sendo que depois passa a triangular; isto porque a simetria triangular corresponde, na natureza, a uma configuração de energia mais baixa e por isso preferencial à quadrada.

De notar que estes fenómenos são a manifestação macroscópica de efeitos quânticos, visíveis precisamente pelas baixas temperaturas a que o condensado se forma.

Referências

- [1] <http://www.cplusplus.com/reference/stl/>
- [2] <http://www.clarku.edu/~djoyce/complex/mult.html>
- [3] <http://www.cplusplus.com/reference/complex/>
- [4] <http://bisqwit.iki.fi/story/howto/openmp/>
- [5] http://software.intel.com/sites/products/documentation/doclib/iss/2013/sa_ptr/sa_ptr_win_lin/GUID-9B08C401-0845-431A-8CEC-E36CCDE66F4C.htm
- [6] <http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/>
- [7] <http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/>
- [8] <http://stackoverflow.com/questions/14093692/whats-the-difference-between-cuda-shared-and-global-memory>
- [9] <http://arxiv.org/pdf/1301.2073v1.pdf>

Anexo — Códigos C++

Makefile

```
1 DB_FLAGS = -Wall -g -ggdb -O3 -lexif -lglut -fopenmp
2
3 LINK_LIBS    = -lm -fopenmp
4
5 CFLAGS       = $(INC_FLAGS) $(DB_FLAGS)
6 LFLAGS       = $(LINK_FLAGS)
7
8 CC          = g++
9 CP          = cp
10 RM         = rm
11 LS          = ls
12
13 PROGS1      = g25s6c1a
14 OUT1        = g25s6c1a.o
15
16 PROGS2      = g25s6c2a compz
17 OUT2        = g25s6c2a.o compz.o
18
19 PROGS3      = g25s6c3a compz
20 OUT3        = g25s6c3a.o compz.o
21
22 PROGS4      = g25s6c3a2 compz
23 OUT4        = g25s6c3a2.o compz.o
24
25 PROGS5      = g25s6c3b compz
26 OUT5        = g25s6c3b.o compz.o
27
28 PROGS6      = g25s6c1b compz
29 OUT6        = g25s6c1b.o compz.o
30
31 all: comp link clean
32
33 comp: comp1 comp2 comp3 comp4 comp5 comp6
34
35 comp1:
36     for fname in $(PROGS1); do $(CC) $(CFLAGS) -c $$fname.cpp ;
37             done
38
39 comp2:
40     for fname in $(PROGS2); do $(CC) $(CFLAGS) -c $$fname.cpp ;
41             done
42
43 comp3:
44     for fname in $(PROGS3); do $(CC) $(CFLAGS) -c $$fname.cpp ;
45             done
46     for fname in $(PROGS4); do $(CC) $(CFLAGS) -c $$fname.cpp ;
```

```

47 comp5:
48     for fname in $(PROGS5); do $(CC) $(CFLAGS) -c $$fname.cpp ;
49         done
50 comp6:
51     for fname in $(PROGS6); do $(CC) $(CFLAGS) -c $$fname.cpp ;
52         done
53 link: link1 link2 link3 link4 link5 link6
54
55 link1:
56     $(CC) $(OUT1) $(LFLAGS) $(LINK_LIBS) -o p1a
57
58 link2:
59     $(CC) $(OUT2) $(LFLAGS) $(LINK_LIBS) -o p2a
60
61 link3:
62     $(CC) $(OUT3) $(LFLAGS) $(LINK_LIBS) -o p3a
63
64 link4:
65     $(CC) $(OUT4) $(LFLAGS) $(LINK_LIBS) -o p3a2
66
67 link5:
68     $(CC) $(OUT5) $(LFLAGS) $(LINK_LIBS) -o p3b
69
70 link6:
71     $(CC) $(OUT6) $(LFLAGS) $(LINK_LIBS) -o p1b
72
73 clean:
74     $(RM) -f *~ *.o $(PROGS1)
75     $(RM) -f *~ *.o $(PROGS2)
76     $(RM) -f *~ *.o $(PROGS3)
77     $(RM) -f *~ *.o $(PROGS4)
78     $(RM) -f *~ *.o $(PROGS5)
79     $(RM) -f *~ *.o $(PROGS6)

```

Listing 8: Código do Makefile, que permite compilar todos os códigos usados em simultâneo

Header para complexos

```
1 #include <cstdlib>
2 #include <cmath>
3 #include <iostream>
4 #include <fstream>
5 #include <ostream>
6 #include <istream>
7 #include <omp.h>
8
9 using namespace std;
10
11 class Complex {
12     public: //variaveis publicas
13         float re; //parte real
14         float im; //parte imaginaria
15     explicit Complex();
16     explicit Complex(float _re, float _im); // construcao explicita
17     Complex(const Complex& comp); //construtor por copia
18     Complex& operator=(const Complex& c);
19     friend ostream& operator<<( ostream& out, const Complex& c );
20     friend istream& operator>>( istream& in, Complex& c );
21 };
22 float mod(const Complex& c); //modulo
23 Complex operator+(const Complex& c1, const Complex& c2); //soma
24 Complex operator-(const Complex& c1, const Complex& c2); //subtracao
25 Complex operator*(const Complex& c1, const Complex& c2); //produto
26 Complex operator/(const Complex& c1, const Complex& c2); //divisao
27 Complex c(float k);
28 Complex conj(const Complex& c);
29 float arg(const Complex& z);
30 Complex polar(const float r, const float t);
31 Complex exp(const Complex& z);
32 Complex cos(const Complex& z);
33 Complex sin(const Complex& z);
34 Complex tan(const Complex& z);
35 Complex sinh(const Complex& z);
36 Complex cosh(const Complex& z);
37 Complex tanh(const Complex& z);
38 Complex log(const Complex& z);
39 Complex sqrt(const Complex& z);
40 Complex pow(const Complex& a, const Complex& b);
41 //Complex random(float re, float im);
42 //size_t operator==(Complex& c1, Complex& c2);
43 //float operator!=(Complex& c1, Complex& c2);
```

Listing 9: Código compz.h, header para complexos

Biblioteca de complexos

```
1 #include "compz.h"
2 #include <cstdlib>
3 #include <cmath>
4 #include <iostream>
5 #include <fstream>
6 #include <ostream>
7 #include <istream>
8 #include <omp.h>
9
10 using namespace std;
11
12 Complex::Complex(){}
13
14 Complex::Complex(float _re, float _im) : re(_re), im(_im) {}
15
16 Complex::Complex(const Complex& comp) : re(comp.re), im(comp.im)
17     {} //construtor por copia
18
19 //Complex::~Complex() { delete &re; delete &im;}
20
21 //Complex::~Complex() { delete &c}
22
23 Complex& Complex::operator=(const Complex& c){
24     Complex temp(c);
25     swap(re,temp.re);
26     swap(im,temp.im);
27     return *this;
28 }
29 ostream& operator<<( ostream& out, const Complex& c ) {
30     if (c.im < 0) out << c.re << c.im << "i";
31     else out << c.re << "+" << c.im << "i";
32     return out;
33 }
34 istream& operator>>( istream& in , Complex& c ) {
35     in >> c.re;
36     in >> c.im;
37     return in;
38 }
39
40 float mod(const Complex& c){ //modulo
41     return sqrt(c.re*c.re+c.im*c.im);
42 }
43
44 Complex operator+(const Complex& c1, const Complex& c2) { //soma
45     float real, imag;
46     real=c1.re+c2.re;
47     imag=c1.im+c2.im;
48     Complex plus(real, imag);
49     return plus;
50 }
```

```

51
52 Complex operator-(const Complex& c1, const Complex& c2) { //  

53     float real, imag;  

54     real=c1.re-c2.re;  

55     imag=c1.im-c2.im;  

56     Complex sub(real, imag);  

57     return sub;  

58 }
59
60 Complex operator*(const Complex& c1, const Complex& c2) { //  

61     float real, imag;  

62     real=c1.re*c2.re - c1.im*c2.im;  

63     imag=c1.re*c2.im + c1.im*c2.re;  

64     Complex prod(real, imag);  

65     return prod;  

66 }
67
68 Complex conj(const Complex& c) { //conjugado  

69     float real, imag;  

70     real=c.re;  

71     imag=c.im;  

72     Complex conju(real, -imag);  

73     return conju;  

74 }
75
76 Complex c(float k){ //real  

77     Complex res(k,0.);  

78     return res;  

79 }
80
81 Complex operator/(const Complex& c1, const Complex& c2) { //  

82     divisao  

83     return c(1. / (mod(c2)*mod(c2)) *c1*conj(c2));  

84 }
85 float arg(const Complex& z){ //argumento do complexo  

86     if(z.re)  

87         return atan(z.im/z.re);  

88     else {  

89         if(z.im>0){  

90             return M_PI/2.;  

91         }  

92         if(z.im<0){  

93             return -M_PI/2.;  

94         }  

95         if(z.im==0) return 0;  

96         else return 0;  

97     }  

98 }
99
100 Complex polar(float r, float t){ // complexo na forma polar  

101    Complex z(r*cos(t),r*sin(t));

```

```

102     return z;
103 }
104
105 Complex exp(const Complex& z){ // exponencial
106     return polar(exp(-z.im),z.re);
107 }
108
109 Complex cos(const Complex& z){ //coseno
110     Complex i(0,1);
111     return c(0.5)*(exp(i*z)+exp(c(-1.)*i*z));
112 }
113
114 Complex sin(const Complex& z){ //seno
115     Complex i(0,1);
116     return c(0.5*(-1.))*i*(exp(i*z)-exp(c(-1.)*i*z));
117 }
118
119 Complex tan(const Complex& z){ //tangente
120     return sin(z)/cos(z);
121 }
122
123 Complex sinh(const Complex& z){ //seno hiperbolico
124     Complex i(0,1);
125     return sin(i*z);
126 }
127
128 Complex cosh(const Complex& z){ //cosseno hiperbolico
129     Complex i(0,1);
130     return cos(i*z);
131 }
132
133 Complex tanh(const Complex& z){ //tangente hiperbolica
134     Complex i(0,1);
135     return tan(i*z);
136 }
137
138 Complex log(const Complex& z){ //logaritmo
139     Complex res(log(mod(z)),arg(z));
140     return res;
141 }
142
143 Complex sqrt(const Complex& z){ //raiz quadrada
144     return polar(sqrt(mod(z)),arg(z));
145 }
146
147 Complex pow(const Complex& a,const Complex& b){ //potencia
148     return exp(b*log(a));
149 }

```

Listing 10: Código compz.cpp, onde se definem as funções

g25s6c1a

```
1 #include <iostream>
2 #include <complex> //numeros complejos
3 #include <cstdlib>
4 #include <list>
5 #include <algorithm>
6 #include <iterator>
7 #include <vector>
8 #include <fstream>
9
10 using namespace std;
11
12 void printLists (const list<int>& l1, const list<int>& l2)
13 {
14     cout << "list1:" << endl;
15     copy (l1.begin(), l1.end(), ostream_iterator<int>(cout, " "))
16     ;
17     cout << endl << "list2:" << endl;
18     copy (l2.begin(), l2.end(), ostream_iterator<int>(cout, " "))
19     ;
20     cout << endl << endl;
21 }
22
23 int main(void)
24 {
25     double re, im, re2, im2;
26
27     cout << "complexo c1" << endl << "real(c1) = ";
28     cin >> re;
29     cout << "im(c1) = ";
30     cin >> im;
31
32     complex<double> c(re, im);
33
34     cout << "abs" << c << " = " << abs(c) << endl;
35     cout << "exp" << c << " = " << exp(c) << endl;
36     cout << "conj" << c << " = " << conj(c) << endl;
37     cout << "log" << c << " = " << log(c) << endl;
38     cout << "norma" << c << " = " << norm(c) << endl;
39     cout << "sqrt" << c << " = " << sqrt(c) << endl;
40
41     cout << endl << "complexo c2" << endl << "real(c2) = ";
42     cin >> re2;
43     cout << "im(c2) = ";
44     cin >> im2;
45
46     complex<double> c2(re2, im2);
47
48     cout << "Abs" << c2 << " = " << abs(c2) << endl;
49     cout << "Exp" << c2 << " = " << exp(c2) << endl;
50     cout << "Conj" << c2 << " = " << conj(c2) << endl;
51     cout << "log" << c2 << " = " << log(c2) << endl;
```

```

51     cout << "Norma" << c2 << "u=u" << norm(c2) << endl;
52     cout << "sqrt" << c2 << "u=u" << sqrt(c2) << endl;
53
54     cout << endl << "soma: u" << c << "u+u" << c2 << "u=u" << c+c2
55         << endl; //soma
56
56     cout << "diferenca: u" << c << "u-u" << c2 << "u=u" << c-c2 << endl
57         ; //subtracao
57     cout << "\t u" << c2 << "u-u" << c << "u=u" << c2-c << endl;
58
59     cout << "produto: u" << c << "u*u" << c2 << "u=u" << c*c2 << endl;
59         //produto
60
61     cout << "divisao: u" << c << "u/u" << c2 << "u=u" << c/c2 << endl;
61         //divisao
62     cout << "\t u" << c2 << "u/u" << c << "u=u" << c2/c << endl;
63
64     cout << endl << "Uso de containers: LIST" << endl;
65
66     list<int> l1, l2; // criação de duas listas
67
68     for (int i=0; i<6; ++i) { // preenchimento das listas
69         l1.push_back(i);
70         l2.push_front(i);
71     }
72     printLists(l1, l2);
73
74     // o primeiro elemento passa para o fim
75     l2.splice(l2.end(),           // destino
76                l2,                 // lista da fonte
77                l2.begin());        // posição da fonte
78     printLists(l1, l2);
79
80     // junção das duas listas na primeira
81     l1.merge(l2);
82     printLists(l1, l2);
83
84     return 0;
85 }
```

Listing 11: Código g25s6c1a

g25s6c1b

```
1 #include <cstdlib>
2 #include <iostream>
3 #include <iomanip>
4 #include <cmath>
5 #include <omp.h>
6 #include <complex>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <time.h>
11
12 using namespace std;
13
14 //processadores
15 void proc(){
16     int N, Nmax;
17     N = omp_get_num_threads();
18     Nmax = omp_get_max_threads();
19
20     cout<<"Processadores:"<<omp_get_num_procs()<<endl;
21
22     if(N==1 && Nmax==1){
23         cout<<"Está activo " << N << " thread, de um total de "
24             << Nmax << " threads permitido" <<endl;
25     }
26     else if(N==1 && Nmax>1){
27         cout<<"Está activo " << N << " thread, de um total de "
28             << Nmax << " threads permitidos" <<endl;
29     }
30     else{
31         cout<<"Estão activos " << N << " threads, de um total de "
32             << Nmax << " threads permitidos" <<endl;
33     }
34
35 void primos(){
36     long int limit ,n,i,j;
37     short int *prime;
38
39     long int sum=1;
40
41     cout << "Limite:" ;
42     cin >> limit;
43
44     //ATKIN metodo para encontrar os primos
45     prime=( short int *)malloc((limit+1)*sizeof( short int));
46     #pragma omp parallel for
47     for(i=0;i<=limit ; i++)
48         prime[ i]=0;
49
```

```

50     prime[2]=1;
51     prime[3]=1;
52
53     clock_t t1=clock(); //start clock
54     int sqlimit= (int)sqrt(limit);
55 #pragma omp parallel for collapse(2) private(n)
56     for(i=1;i<=sqlimit ; i++){      // encontrar candidatos
57         for(j=1;j<=sqlimit ; j++){
58             n=4*i*i+j*j;
59             if ((n<=limit)&&(n%12==1||n%12==5))
60                 prime[n]=(prime[n]? 0 : 1);
61             n=3*i*i+j*j;
62             if ((n<=limit)&&(n%12==7))
63                 prime[n]=(prime[n]? 0 : 1);
64             n=3*i*i-j*j;
65             if ((i>j)&&(n<=limit)&&(n%12==11))
66                 prime[n]=(prime[n]? 0 : 1);
67         }
68     }
69     int n0;
70 #pragma omp parallel for private(n,n0)
71     for(i=5;i<=sqlimit ; i+=2){    // eliminar multiplos dos
72         quadrados
73         if(prime[i]){
74             n0=i*i;
75             for(n=n0;n<=limit ; n+=n0 )
76                 prime[n]= 0;
77         }
78     }
79 #pragma omp parallel for reduction(:sum)
80     for(i=3;i<=limit ; i+=2){
81         if(prime[i]) sum++;
82     }
83     cout << "#primos:" << sum;
84     sum=1;
85
86     for(i=3;i<=limit ; i+=2){
87         if(prime[i]){
88             sum++;
89
90             if (!(sum%7)){
91             }
92         }
93     }
94     cout << endl << "RUNTIME:" << (double)((clock()-t1)/(double)
95           CLOCKS_PER_SEC) << endl; //stop clock
96     cout << endl;
97
98 int main(){
99
100    int a;
101

```

```
102     cout << "Escolha a opcao desejada" << endl;
103     cout << "1: Numero de processadores e threads" << endl;
104     cout << "2: Calculo dos numeros primos" << endl; //adaptacao
105     programacao 1 ano
106     cin >> a;
107     if (a==1){
108         proc();
109     }
110     if (a==2){
111         primos();
112     }
113
114     return 0;
115 }
116 }
```

Listing 12: Código g25s6c1b

g25s6c2a

```
1 #include "compz.h"
2 #include <cstdlib>
3 #include <cmath>
4 #include <iostream>
5
6 using namespace std;
7
8 int main(void)
9 {
10     double re1, im1, re2, im2;
11     //Complex c;
12
13     cout << "complexo c1" << endl << "real(c1) = ";
14     cin >> re1;
15     cout << "im(c1) = ";
16     cin >> im1;
17
18     Complex c1(re1, im1); //inicializacao do complexo1
19
20     cout << "mod(" << c1 << ")" = << mod(c1) << endl; //modulo ou
21         valor absoluto
22     cout << "arg(" << c1 << ")" = << arg(c1) << endl; //argumento
23     cout << "conjulado(" << c1 << ")" = << conj(c1) << endl; //
24         conjugado
25     cout << "exp(" << c1 << ")" = << exp(c1) << endl; //
26         exponencial
27     cout << "log(" << c1 << ")" = << log(c1) << endl; //logaritmo
28     cout << "sqrt(" << c1 << ")" = << sqrt(c1) << endl; //raiz
29         quadrada
30
31     cout << endl << "complexo c2" << endl << "real(c2) = ";
32     cin >> re2;
33     cout << "im(c2) = ";
34     cin >> im2;
35
36     Complex c2(re2, im2); //inicializacao do complexo2
37
38     cout << "mod(" << c2 << ")" = << mod(c2) << endl;
39     cout << "arg(" << c2 << ")" = << arg(c2) << endl;
40     cout << "conjulado(" << c2 << ")" = << conj(c2) << endl; //
41         conjugado
42     cout << "exp(" << c2 << ")" = << exp(c2) << endl; //
43         exponencial
44     cout << endl << "soma: " << c1 << "+ " << c2 << " = " << c1+c2 <<
45         endl; //soma
```

```
45     cout << "diferenca : " << c1 << " - " << (c2 << " ) " << c1-c2 <<
46         endl; //subtracao
47     cout << "\t" << c2 << " - " << (c1 << " ) " << c2-c1 << endl;
48
49     cout << "produto : " << (c1 << " ) * " << (c2 << " ) " << c1*c2 <<
50         endl; //produto
51     cout << "divisao : " << (c1 << " ) / " << (c2 << " ) " << c1/c2 <<
52         endl; //divisao
53     cout << "\t" << (c2 << " ) / " << (c1 << " ) " << c2/c1 << endl;
54
55     return 0;
56 }
```

Listing 13: Código g25s6c2a

g25s6c3a

```
1 #include <iostream>
2 #include <cmath>
3 #include <string>
4 #include <cstring>
5 #include <ctime>
6 #include "compz.h"
7
8
9 //definicao do valor das constantes
10 #define g 0.01
11 #define omega 0.85
12 #define G 1000.0
13 #define L 10.0
14 #define N 128
15 #define dt 0.002
16 using namespace std;
17
18 Complex F(Complex ax1, Complex ax2, Complex ay1, Complex ay2,
19             Complex a,int i,int j){ //funcao F
20     Complex k(-g,1.);
21     Complex iomg(0.,-omega);
22     Complex h=c(2.*L/((double)N-1.));
23     Complex dx2 = (ax1+ax2-a-a)/(h*h);
24     Complex dy2 = (ay1+ay2-a-a)/(h*h);
25     Complex lap = dx2+dy2;
26     Complex dx = (ax1-ax2)/(h*c(2.));
27     Complex dy = (ay1-ay2)/(h*c(2.));
28     Complex x=c(-L)+c(i)*h;
29     Complex y=c(-L)+c(j)*h;
30
31     return (c(-0.5)*lap + c(0.5)*(x*x+y*y)*a + c(G*mod(a)*
32         mod(a))*a + iomg*(x*dy-y*dx))/k;
33 }
34
35
36 int main(void){
37     string name, sfps;
38     int i,j,n, cpf;
39     double h, tf ,sum, fps;
40     h=2.*L/((double)N-1.); //variaveis necessarias e se output
41     cout << "gamma=" << g << endl;
42     cout << "Omega=" << omega << endl;
43     cout << "G=" << G << endl;
44     cout << "N=" << N << endl;
45     cout << "L=" << L << endl;
46     cout << "h=" << h << endl;
47     cout << "dt=" << dt << endl;
48     cout << "tf=";
49     cin >> tf;
50     int nmax=tf/dt;
51     cout << "#cycles=" << nmax << endl;
```

```

51     cout << "cycles per frame = ";
52     cin >> cpf;
53     int nframes = nmax/cpf;
54     cout << "# frames = " << nframes << endl;
55     cout << "About " << nframes*45./1024. << " Mb will be used" <<
56         endl;
57     cout << "fps = ";
58     cin >> sfps;
59     fps=atoi(sfps.c_str());
60     cout << "Length of video = " << nframes/fps << " secs" << endl
61         ;
62     cout << "Video name (no extension): ";
63     cin >> name;
64     cout << "Video output will be" << name << ".mp4, about " <<
65         nframes/1024. << " Mb" << endl;
66     cout << "Progress:" << endl;
67     Complex **psi, **psi1, **psi2; // alocação de memoria
68     psi = new Complex*[N];
69     psi1 = new Complex*[N];
70     psi2 = new Complex*[N];
71     for(i=0;i<N;++i){
72         psi[i] = new Complex[N];
73         psi1[i] = new Complex[N];
74         psi2[i] = new Complex[N];
75     }
76     Complex one(1.,0.);
77     Complex zero(0.,0.);
78     for(i=0;i<N;++i){ // valores iniciais e normalização
79         for(j=0;j<N;++j){
80             if(i==0||j==0||i==N-1||j==N-1) {
81                 psi[i][j]=zero;
82                 psi1[i][j]=zero;
83                 psi2[i][j]=zero;
84             }
85             else {
86                 psi[i][j]=c(1./((N-2.)*h))*one;
87                 psi1[i][j]=c(1./((N-2.)*h))*one;
88                 psi2[i][j]=c(1./((N-2.)*h))*one;
89             }
90         }
91         ofstream out("../data3a"); // escrita inicial no ficheiro
92         for(i=1;i<N-1;++i){
93             for(j=1;j<N-1;++j){
94                 out << -L+i*h << "\t" << -L+j*h << "\t" << mod(psi[i][j])*
95                     mod(psi[i][j]) << endl;
96             }
97         }
98         clock_t t=clock();
99         double v=cpf; // variável que define a frequência do ciclo de
100            tempo
101         int pop=v/cpf; // pop define o múltiplo de cpf com que é
102            feito o refresh da barra de progresso
103         sum=0;

```

```

99   for(n=0;n<nmax;n++){
100     if(!(n%cpf)){
101       out.open("../data3a"); // abertura do ficheiro de
102         escrita de dados
103         ofstream script("../script3a"); // criacao do script
104         script << "settermpng" << endl;
105         script << "setoutput\..\frame" << n/cpf << ".png\""
106         << endl;
107         script << "settitle\"Gross-Pitaevskii-Bose-Einstein"
108           Condensate\"" << endl;
109         script << "setxrange[-10:10]" << endl;
110         script << "setyrange[-10:10]" << endl;
111         script << "setcbrange[0:*)" << endl;
112         script << "setsize square" << endl;
113         script << "set xlabel\"x\"" << endl;
114         script << "set ylabel\"y\"" << endl;
115         script << "set xlabel\"psi*(psi*)\"" << endl;
116         script << "plot\..\data3a\"using 1:2:3 notitle wpu
117           pt7ups2ulpalette" << endl;
118         script.close();
119     }
120   }
121   for(i=1;i<N-1;++i){ // implementacao do metodo
122     for(j=1;j<N-1;++j){
123       psi1[i][j]=psi[i][j]+c(dt)*F(psi[i+1][j],psi[i-1][j],psi[i][j
124         +1],psi[i][j-1],psi[i][j],i,j);
125     }
126   }
127   for(i=1;i<N-1;++i){
128     for(j=1;j<N-1;++j){
129       psi2[i][j]=c(0.75)*psi[i][j]+c(0.25)*psi1[i][j]+c(0.25*dt)*F(
130         psi1[i+1][j],psi1[i-1][j],psi1[i][j+1],psi1[i][j-1],psi1[i
131         ][j],i,j);
132     }
133   }
134   for(i=1;i<N-1;++i){
135     for(j=1;j<N-1;++j){
136       psi[i][j]=psi[i][j]+c(2.0/3.)*psi2[i][j]+c((2.0/3.)
137         *dt)*F(psi2[i+1][j],psi2[i-1][j],psi2[i][j+1],psi2[i][j
138         -1],psi2[i][j],i,j);
139     }
140   }
141   sum=0.;
142   for(i=1;i<N-1;++i){ // calculo de N para a normalizacao
143     for(j=1;j<N-1;++j){
144       sum+=h*h*mod(psi[i][j])*mod(psi[i][j]);
145     }
146   }
147   for(i=1;i<N-1;++i){ //normalizacao
148     for(j=1;j<N-1;++j){
149       psi[i][j] = psi[i][j]/c(sqrt(sum));
150     }
151   }
152 }
```

```

144 if (!(n%cpf)){
145     for (i=1;i<N-1;++i){
146         for (j=1;j<N-1;++j){ //escrita no ficheiro de dados
147             out << -L+i*h << "\t" << -L+j*h << "\t" << mod(psi[i][j])*mod(psi[i][j]) << endl;
148     }
149 }
150 out.close(); //fecho do ficheiro de escrita de dados
151 if (!system("gnuplot.../script3a&")) //criacao do frame
152 if (!(n%(pop*cpf))){ // progress bar
153     v=((double)pop*cpf)/(((double)clock())-(double)t)/CLOCKS_PER_SEC;
154     if (v==0) v=1.;
155     cout << "\r" << n*100./nmax << "%\u00b7\u00b7\u00b7" << "f\u00b7=\u00b7" << v << "\u00b7cyc/sec\u00b7\u00b7\u00b7ETA\u00b7\u00b7" << (int)((((double)nmax-(double)n)/v)/60.) << "min\u00b7\u00b7" << (int)((((double)nmax-(double)n)/v)-60*(int)((((double)nmax-(double)n)/v)/60.)) << "\u00b7sec\u00b7\u00b7\u00b7\u00b7\u00b7\u00b7\u00b7";
156     cout << "\n[\u00b7";
157     for (i=0;i<100*n/nmax;++i) cout << " ";
158     for (i=0;i<100-100*n/nmax;++i) cout << "\u00b7";
159     cout << "]";
160     cout << flush;
161     cout << "\e[A\r";
162 }
163 }
164 if (!(n%(pop*cpf))){ //actualizacao de v e pop
165     t=clock();
166     if (v<100.)
167         pop=((int)v)/cpf;
168     if (pop==0) pop=1;
169 }
170 }
171 for (i=0;i<N;++i){ // libertacao memoria
172     delete [] psi[i];
173     delete [] psi1[i];
174     delete [] psi2[i];
175 }
176 delete [] psi;
177 delete [] psi1;
178 delete [] psi2;
179 cout << "\n" << endl; //manipulacao de strings usados em
180 //system
181 string compile = "avconv\u00b7-r\u00b7" + sfps + "\u00b7-i\u00b7" + ".../frame\u00b7%d.png\u00b7-r\u00b7" + sfps + "\u00b7" + name + ".mp4"; //compilacao video
182 char * cstring = new char[compile.length()+1];
183 strcpy(cstring,compile.c_str());
184
185 string video = "totem\u00b7" + name + ".mp4&";
186 char * cvideo = new char[video.length()+1];
187 strcpy(cvideo,video.c_str());

```

```
188
189 if (!system(cstring)){
190     if (!system("rm .../.../frame*.png")){ // eliminacao frames
191         apos render do video
192         if (!system(cvideo)){
193             return 0;
194         }
195         else return 1;
196     }
197     else return 1;
198 }
```

Listing 14: Código g25s6c3a

g25s6c3a2²

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 #include <cmath>
5 #include <string>
6 #include <cstring>
7 #include <ctime>
8 #include "compz.h"
9
10
11 //definicao do valor das constantes
12 #define g 0.01
13 #define omega 0.85
14 #define G 1000.0
15 #define L 10.0
16 #define N 128
17 #define dt 0.001953125
18 using namespace std;
19
20
21 Complex F(Complex ax1, Complex ax2, Complex ay1, Complex ay2,
22             Complex a,int i,int j){ //funcao F
22     Complex k(-g,1.);
23     Complex iomg(0.,-omega);
24     Complex h=c(2.*L/((double)N-1.));
25     Complex dx2 = (ax1+ax2-a-a)/(h*h);
26     Complex dy2 = (ay1+ay2-a-a)/(h*h);
27     Complex lap = dx2+dy2;
28     Complex dx = (ax1-ax2)/(h*c(2.));
29     Complex dy = (ay1-ay2)/(h*c(2.));
30     Complex x=c(-L)+c(i)*h;
31     Complex y=c(-L)+c(j)*h;
32
33     return (c(-0.5)*lap + c(0.5)*(x*x+y*y)*a + c(G*mod(a)*
34         mod(a))*a + iomg*(x*dy-y*dx))/k;
34 }
35
36
37 int main(void){
38
39     int i,j,n, cpf;
40     double h,tf,sum;
41
42     h=2.*L/((double)N-1.);
43     cout << "gamma=" << g << endl;
44     cout << "Omega=" << omega << endl;
45     cout << "G=" << G << endl;
46     cout << "N=" << N << endl;
47     cout << "L=" << L << endl;
48     cout << "h=" << h << endl;
```

²Código alternativo, sem criação do vídeo

```

49     cout << "dt=" << dt << endl;
50     cout << "tf=";
51     cin >> tf;
52     int nmax=tf/dt;
53     cout << "#cycles=" << nmax << endl;
54     cout << "cycles per frame=";
55     cin >> cpf;
56     int nframes = nmax/cpf;
57     cout << "#frames=" << nframes << endl;
58
59     cout << "Progress:" << endl;
60     Complex **psi, **psi1, **psi2;
61     psi = new Complex*[N];
62     psi1 = new Complex*[N];
63     psi2 = new Complex*[N];
64
65
66     for(i=0;i<N;++i){
67         psi[i] = new Complex[N];
68         psi1[i] = new Complex[N];
69         psi2[i] = new Complex[N];
70     }
71     Complex one(1.,0.);
72     Complex zero(0.,0.);
73
74
75     for(i=0;i<N;++i){
76         for(j=0;j<N;++j){
77             if(i==0||j==0||i==N-1||j==N-1) {
78                 psi[i][j]=zero;
79                 psi1[i][j]=zero;
80                 psi2[i][j]=zero;
81             }
82             else {
83                 psi[i][j]=c(1./((N-2.)*h))*one;
84                 psi1[i][j]=c(1./((N-2.)*h))*one;
85                 psi2[i][j]=c(1./((N-2.)*h))*one;
86             }
87         }
88     }
89     ofstream out("../data3c");
90     for(i=1;i<N-1;++i){
91         for(j=1;j<N-1;++j){
92             out << -L+i*h << "\t" << -L+j*h << "\t" << mod(psi[i][j])*
93                         mod(psi[i][j]) << endl;
94         }
95     }
96     clock_t t=clock();
97     double v=cpf;
98     int pop=v/cpf;
99     sum=0;
100    for(n=0;n<nmax;n++){
101

```

```

102     for( i=1;i<N-1;++i){ // implementacao do metodo
103         for( j=1;j<N-1;++j ){
104             psi1[ i ][ j ]=psi[ i ][ j ]+c( dt )*F( psi[ i+1 ][ j ], psi[ i-1 ][ j ], psi[ i ][ j
105                         +1], psi[ i ][ j-1], psi[ i ][ j ], i , j );
106         }
107     }
108
109     for( i=1;i<N-1;++i){
110         for( j=1;j<N-1;++j ){
111             psi2[ i ][ j ]=c( 0.75 )*psi[ i ][ j ]+c( 0.25 )*psi1[ i ][ j ]+c( 0.25*dt )*F(
112                 psi1[ i+1 ][ j ], psi1[ i-1 ][ j ], psi1[ i ][ j+1 ], psi1[ i ][ j-1 ], psi1[ i
113                         ][ j ], i , j );
114         }
115     }
116
117     for( i=1;i<N-1;++i){
118         for( j=1;j<N-1;++j ){
119             psi[ i ][ j ]=c( 1.0 / 3. )*psi[ i ][ j ]+c( 2.0 / 3. )*psi2[ i ][ j ]+c( ( 2.0 / 3.
120                         ) *dt )*F( psi2[ i+1 ][ j ], psi2[ i-1 ][ j ], psi2[ i ][ j+1 ], psi2[ i ][ j
121                         -1 ], psi2[ i ][ j ], i , j );
122         }
123     }
124     sum=0. ;
125
126     for( i=1;i<N-1;++i){ // calculo de N para a normalizacao
127         for( j=1;j<N-1;++j ){
128             sum+=h*h*mod( psi[ i ][ j ]) *mod( psi[ i ][ j ]) ;
129         }
130
131     for( i=1;i<N-1;++i){ //normalizacao
132         for( j=1;j<N-1;++j ){
133             psi[ i ][ j ] = psi[ i ][ j ]/ c( sqrt( sum ) );
134         }
135     }
136
137     if( !( n%cpf )) {
138         out.open( " .. / .. / data3c " );
139         for( i=1;i<N-1;++i ){
140             for( j=1;j<N-1;++j ){ // escrita no ficheiro de dados
141                 out << -L+i*h << "\t" << -L+j*h << "\t" << mod( psi[ i ][ j ]) *
142                     mod( psi[ i ][ j ]) << endl ;
143             }
144         out.close(); // fecho do ficheiro de escrita de dados
145
146         if( !( n%(pop*cpf ))){ // progress bar
147             v=((double)pop*cpf)/(((double)clock()-(double)t) /
148                         CLOCK_S_PER_SEC) ;
149             if( v==0 ) v=1. ;

```

```

149     cout << "\r" << n*100./nmax << "%\u00d7" << f_u=u" << v << "\u
cyc/sec\u00d7ETA\u00d7" << (int) (((double)nmax-(double)n)/v)
/60.) << "min\u00d7" << (int) (((double)nmax-(double)n)/v)
-60*(int) (((double)nmax-(double)n)/v)/60.) << "\u sec\u00d7
\u00d7\u00d7\u00d7\u00d7\u00d7\u00d7";
150     cout << "\n[\u00d7";
151     for(i=0;i<100*n/nmax;++i) cout << "    ";
152     for(i=0;i<100-100*n/nmax;++i) cout << "\u00d7";
153     cout << "]";
154     cout << flush;
155     cout << "\e[A\r";
156 }
157 }
158 if (!(n%(pop*cpf))){
159     t=clock();
160     if (v<100.)
161     pop=((int)v)/cpf;
162     if (pop==0) pop=1;
163 }
164
165 }
166
167
168
169 for(i=0;i<N;++i){ // libertacao memoria
170     delete [] psi[i];
171     delete [] psi1[i];
172     delete [] psi2[i];
173 }
174
175 delete [] psi;
176 delete [] psi1;
177 delete [] psi2;
178 cout << "\n" << endl;
179
180 return 0;
181
182 }

```

Listing 15: Código g25s6c3a2

g25s6c3b

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 #include <cmath>
5 #include <string>
6 #include <cstring>
7 #include <ctime>
8 #include <omp.h>
9 #include "compz.h"
10
11 //definicao do valor das constantes
12 #define g 0.01
13 #define omega 0.85
14 #define G 1000.0
15 #define L 10.0
16 #define N 128
17 #define dt 0.001953125
18 using namespace std;
19
20
21 Complex F(Complex ax1, Complex ax2, Complex ay1, Complex ay2,
22             Complex a,int i,int j){ //funcao F
22     Complex k(-g,1.);
23     Complex iomg(0.,-omega);
24     Complex h=c(2.*L/((double)N-1.));
25     Complex dx2 = (ax1+ax2-a-a)/(h*h);
26     Complex dy2 = (ay1+ay2-a-a)/(h*h);
27     Complex lap = dx2+dy2;
28     Complex dx = (ax1-ax2)/(h*c(2.));
29     Complex dy = (ay1-ay2)/(h*c(2.));
30     Complex x=c(-L)+c(i)*h;
31     Complex y=c(-L)+c(j)*h;
32
33     return (c(-0.5)*lap + c(0.5)*(x*x+y*y)*a + c(G*mod(a)*
34         mod(a))*a + iomg*(x*dy-y*dx))/k;
34 }
35
36
37 int main(void){
38     int i,j,n, cpf;
39     double h,tf,sum;
40
41     h=2.*L/((double)N-1.);
42     cout << "gamma_u=" << g << endl;
43     cout << "Omega_u=" << omega << endl;
44     cout << "G_u=" << G << endl;
45     cout << "N_u=" << N << endl;
46     cout << "L_u=" << L << endl;
47     cout << "h_u=" << h << endl;
48     cout << "dt_u=" << dt << endl;
49     cout << "tf_u=";
50     cin >> tf;
```

```

51   int nmax=tf/dt;
52   cout << "#cycles" << nmax << endl;
53   cout << "cycles per frame";
54   cin >> cpf;
55   int nframes = nmax/cpf;
56   cout << "#frames" << nframes << endl;
57   cout << "Progress:" << endl;
58   Complex **psi, **psi1, **psi2;
59   psi = new Complex*[N];
60   psi1 = new Complex*[N];
61   psi2 = new Complex*[N];
62   //implementacao de openmp
63   #pragma omp parallel for
64   for(i=0;i<N;++i){
65     psi[i] = new Complex[N];
66     psi1[i] = new Complex[N];
67     psi2[i] = new Complex[N];
68   }
69   Complex one(1.,0.);
70   Complex zero(0.,0.);
71   //nested for
72   #pragma omp parallel for collapse(2)
73   for(i=0;i<N;++i){
74     for(j=0;j<N;++j){
75       if(i==0||j==0||i==N-1||j==N-1) {
76         psi[i][j]=zero;
77         psi1[i][j]=zero;
78         psi2[i][j]=zero;
79       }
80       else {
81         psi[i][j]=c(1./((N-2.)*h))*one;
82         psi1[i][j]=c(1./((N-2.)*h))*one;
83         psi2[i][j]=c(1./((N-2.)*h))*one;
84       }
85     }
86   }
87   ofstream out("data3b");
88   for(i=1;i<N-1;++i){
89     for(j=1;j<N-1;++j){
90       out << -L+i*h << "\t" << -L+j*h << "\t" << mod(psi[i][j]) *
91       mod(psi[i][j]) << endl;
92     }
93   }
94   clock_t t=clock();
95   double v=cpf;
96   int pop=v/cpf;
97   sum=0;
98   for(n=0;n<nmax;n++){
99   #pragma omp parallel for collapse(2)
100    for(int i=1;i<N-1;++i){ // implementacao do metodo
101      for(int j=1;j<N-1;++j){
102        psi1[i][j]=psi[i][j]+c(dt)*F(psi[i+1][j],psi[i-1][j],psi[i][j]
103        +1),psi[i][j-1],psi[i][j],i,j);

```

```

103         }
104     }
105
106 #pragma omp parallel for collapse(2)
107 for(int i=1;i<N-1;++i){
108     for(int j=1;j<N-1;++j){
109         psi2[i][j]=c(0.75)*psi[i][j]+c(0.25)*psi1[i][j]+c(0.25*dt)*F(
110             psi1[i+1][j],psi1[i-1][j],psi1[i][j+1],psi1[i][j-1],psi1[i]
111             ][j],i,j);
112     }
113
114 #pragma omp parallel for collapse(2)
115 for(int i=1;i<N-1;++i){
116     for(int j=1;j<N-1;++j){
117         psi[i][j]=c(1.0/3.)*psi[i][j]+c(2.0/3.)*psi2[i][j]+c((2.0/3.)
118             *dt)*F(psi2[i+1][j],psi2[i-1][j],psi2[i][j+1],psi2[i][j
119             -1],psi2[i][j],i,j);
120     }
121     sum=0.;
122
123 #pragma omp parallel for collapse(2) reduction(:sum)
124 for(int i=1;i<N-1;++i){ //calculo de N para a normalizacao
125     for(int j=1;j<N-1;++j){
126         sum+=h*h*mod(psi[i][j])*mod(psi[i][j]);
127     }
128
129 #pragma omp parallel for collapse(2)
130 for(int i=1;i<N-1;++i){ //normalizacao
131     for(int j=1;j<N-1;++j){
132         psi[i][j] = psi[i][j]/c(sqrt(sum));
133     }
134
135 if(!(n%cpf)){
136     out.open("data3b");
137     for(i=1;i<N-1;++i){
138         for(j=1;j<N-1;++j){ //escrita no ficheiro de dados
139             out << -L+i*h << "\t" << -L+j*h << "\t" << mod(psi[i][j])*
140                 mod(psi[i][j]) << endl;
141         }
142         out.close(); //fecha o ficheiro de escrita de dados
143     if(!(n%(pop*cpf))){ // progress bar
144         v=((double)pop*cpf)/(((double)clock()-(double)t)/
145             CLOCKS_PER_SEC);
146         if(v==0) v=1;
147         cout << "\r" << n*100./nmax << "% " << "f = "
148             << (int)((((double)nmax-(double)n)/v)
149             /60.) << "min " << (int)((((double)nmax-(double)n)/v)
150             -60*(int)((((double)nmax-(double)n)/v)/60.)) << "sec ";

```

```

147     cout << "\n[" ;
148     for(i=0;i<100*n/nmax;++i) cout << "    ";
149     for(i=0;i<100-100*n/nmax;++i) cout << " ]";
150     cout << "]";
151     cout << flush;
152     cout << "\e[A\rf";
153 }
154 }
155 if (!(n%(pop*cpf))){ 
156     t=clock();
157     if(v<100.)
158     pop=((int)v)/cpf;
159     if(pop==0) pop=1;
160 }
161
162 }
163
164
165 #pragma omp parallel for private(i)
166 for(i=0;i<N;++i){ // libertacao memoria
167     delete [] psi[i];
168     delete [] psi1[i];
169     delete [] psi2[i];
170 }
171
172 delete [] psi;
173 delete [] psi1;
174 delete [] psi2;
175 cout << "\n" << endl;
176
177 return 0;
178 }

```

Listing 16: Código g25s6c3b

g25s6c4a

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 #include <cmath>
5 #include <string>
6 #include <cstring>
7 #include <ctime>
8 #include <cuda.h>
9 #include "cuda_common.h"
10 #include <thrust/version.h>
11 #include <thrust/transform.h>
12 #include <thrust/device_vector.h>
13 #include <thrust/host_vector.h>
14 #include <thrust/sort.h>
15
16 #ifdef __CUDACC__
17 #define CUDA_CALLABLE_MEMBER __host__ __device__
18 #else
19 #define CUDA_CALLABLE_MEMBER
20 #endif
21
22
23 //definicao do valor das constantes
24 #define g 0.01
25 #define omega 0.85
26 #define G 1000.0
27 #define L 10.0
28 #define N 128
29 #define S 16384
30 #define dt 0.00390625
31
32 using namespace std;
33
34 class Complex {
35
36
37
38 public: //variaveis publicas
39     double re; //parte real
40     double im; //parte imaginaria
41     CUDA_CALLABLE_MEMBER explicit Complex() {};
42     CUDA_CALLABLE_MEMBER explicit Complex(double _re, double _im) :
43         re(_re), im(_im) {} // construcao explicita
44     CUDA_CALLABLE_MEMBER Complex(const Complex& comp) : re(comp.re)
45         , im(comp.im) {} //construtor por copia
46     CUDA_CALLABLE_MEMBER Complex& operator=(const Complex& c){
47
48         re=c.re;
49         im=c.im;
50     }
```

```

51     return *this;
52 }
53 CUDA_CALLABLE_MEMBER ~Complex() {}
54
55 CUDA_CALLABLE_MEMBER friend double mod(const Complex& c); //  

56     modulo
56 CUDA_CALLABLE_MEMBER friend Complex operator+(const Complex&  

57         c1, const Complex& c2); //soma
57 CUDA_CALLABLE_MEMBER friend Complex operator-(const Complex&  

58         c1, const Complex& c2); //subtracao
58 CUDA_CALLABLE_MEMBER friend Complex operator*(const Complex&  

59         c1, const Complex& c2); //produto
59 CUDA_CALLABLE_MEMBER friend Complex operator/(const Complex&  

60         c1, const Complex& c2); //divisao
60 CUDA_CALLABLE_MEMBER friend Complex c(double k);
61 CUDA_CALLABLE_MEMBER friend Complex conj(const Complex& c);
62 }
63
64 double mod(const Complex& c){ //modulo
65     return sqrt(c.re*c.re+c.im*c.im);
66 }
67
68 Complex operator+(const Complex& c1, const Complex& c2) { //  

69     soma
70     double real, imag;
71     real=c1.re+c2.re;
72     imag=c1.im+c2.im;
73     Complex plus(real, imag);
74     return plus;
75 }
75
76 Complex operator- (const Complex& c1, const Complex& c2) { //  

77     subtracao
77     double real, imag;
78     real=c1.re-c2.re;
79     imag=c1.im-c2.im;
80     Complex sub(real, imag);
81     return sub;
82 }
83
84 Complex operator* (const Complex& c1, const Complex& c2) { //  

85     produto
85     double real, imag;
86     real=c1.re*c2.re - c1.im*c2.im;
87     imag=c1.re*c2.im + c1.im*c2.re;
88     Complex prod(real, imag);
89     return prod;
90 }
91
92 Complex conj (const Complex& c) { //conjugado
93     double real, imag;
94     real=c.re;
95     imag=c.im;
96     Complex conju(real, -imag);

```

```

97     return conju;
98 }
99
100 Complex c(double k){ //real
101     Complex res(k,0.);
102     return res;
103 }
104
105 Complex operator/(const Complex& c1, const Complex& c2) { //divisao
106     return c(1./((mod(c2)*mod(c2)))*c1*conj(c2));
107 }
108
109
110
111 _host__device_ Complex F(Complex ay1, Complex ay2, Complex
112     ax1, Complex ax2, Complex a,int i,int j){ //funcao F
113     Complex k(-g,1.);
114     Complex iomg(0.,-omega);
115     Complex h=c(2.*L/((double)N-1.));
116     Complex dx2 = (ax1+ax2-a-a)/(h*h);
117     Complex dy2 = (ay1+ay2-a-a)/(h*h);
118     Complex lap = dx2+dy2;
119     Complex dx = (ax1-ax2)/(h*c(2.));
120     Complex dy = (ay1-ay2)/(h*c(2.));
121     Complex x=c(-L)+c(i)*h;
122     Complex y=c(-L)+c(j)*h;
123
124     return (c(-0.5)*lap + c(0.5)*(x*x+y*y)*a + c(G*mod(a)*
125         mod(a))*a + iomg*(x*dy-y*dx))/k;
126 }
127 //kernels
128 _global_ void k1( Complex* psi ,Complex* psi1 , Complex* psi2)
129 {
130     int i=threadIdx.y+blockIdx.y*blockDim.y;
131     int j=threadIdx.x+blockIdx.x*blockDim.x;
132     int index=j*N+i;
133     if( ( i > 0 && i < N-1) && ( j > 0 && j < N-1) )
134     {
135         psi1[j*N+i] = psi[j*N+i] + c(dt)*F(psi[(j+1)*N+i],psi[(j
136             -1)*N+i],psi[j*N+i+1],psi[j*N+i-1],psi[index],i,j);
137     }
138 }
139 _global_ void k2( Complex* psi ,Complex* psi1 , Complex* psi2)
140 {
141
142     int i=threadIdx.y+blockIdx.y*blockDim.y;
143     int j=threadIdx.x+blockIdx.x*blockDim.x;
144     if( ( i > 0 && i < N-1) && ( j > 0 && j < N-1) )

```

```

147    {
148        int index=j*N+i;
149        psi2[index] = c(3.0/4)*psi[index] + c(1.0/4)*psi1[index] +
150            c(dt*1.0/4)*F(psi1[(j+1)*N+i],psi1[(j-1)*N+i],psi1[j*N+
151                i+1],psi1[j*N+i-1],psi1[index],i,j);
152    }
153    __global__ void k3(Complex* psi, Complex* psi1, Complex* psi2)
154    {
155        int i=threadIdx.y+blockIdx.y*blockDim.y;
156        int j=threadIdx.x+blockIdx.x*blockDim.x;
157
158        if( (i > 0 && i < N-1) && (j > 0 && j < N-1) )
159        {
160            int index=j*N+i;
161            psi[index] = c(1.0/3.)*psi[index] + c(2.0/3.)*psi2[index]
162                + c(dt*2.0/3.)*F(psi2[(j+1)*N+i],psi2[(j-1)*N+i],psi2[j*
163                    N+i+1],psi2[j*N+i-1],psi2[index],i,j);
164        }
165    __global__ void k4( Complex* psi, Complex* psi1, double *Normc)
166    {
167        double h2= (2.*L/((double)N-1.))*(2.*L/((double)N-1.));
168
169        int i=threadIdx.y+blockIdx.y*blockDim.y;
170        int j=threadIdx.x+blockIdx.x*blockDim.x;
171
172        if( (i > 0 && i < N-1) && (j > 0 && j < N-1) )
173        {
174            int index=j*N+i;
175
176            Normc[index]=mod(psi[index])*mod(psi[index])*h2;
177        }
178    }
179
180    __global__ void k5( Complex* psi , double Norm)
181    {
182        int i=threadIdx.y+blockIdx.y*blockDim.y;
183        int j=threadIdx.x+blockIdx.x*blockDim.x;
184
185        if( (i > 0 && i < N-1) && (j > 0 && j < N-1) )
186        {
187            int index=j*N+i;
188            psi[index]=psi[index]/c(sqrt(Norm));
189        }
190    }
191
192
193
194
195
196 int main(void){

```

```

197
198     int i,j,n, cpf;
199     double h,tf;
200
201     h=2.*L/((double)N-1.);
202     cout << "gamma_u" << g << endl;
203     cout << "Omega_u" << omega << endl;
204     cout << "G_u" << G << endl;
205     cout << "N_u" << N << endl;
206     cout << "L_u" << L << endl;
207     cout << "h_u" << h << endl;
208     cout << "dt_u" << dt << endl;
209     cout << "tf_u";
210     cin >> tf;
211     int nmax=tf/dt;
212     cout << "#cycles_u" << nmax << endl;
213     cout << "cycles_per_frame_u";
214     cin >> cpf;
215     int nframes = nmax/cpf;
216     cout << "#frames_u" << nframes << endl;
217     cout << "About_u" << nframes*45./1024. << "Mb will be used" <<
218     endl;
219
220     cout << "Progress:" << endl;
221     //allocacao
222     Complex * psi=(Complex *) malloc(S*sizeof(Complex));
223     Complex * psi1=(Complex *) malloc(S*sizeof(Complex));
224     Complex * psi2=(Complex *) malloc(S*sizeof(Complex));
225     double * Norm = (double *) malloc(S*sizeof(double));
226     double Nm=0;
227     Complex * psic;
228     Complex * psi1c;
229     Complex * psi2c;
230     double *Normc;
231
232     cudaMalloc((void**)&psic, S*sizeof(Complex));
233     cudaMalloc((void**)&psi1c, S*sizeof(Complex));
234     cudaMalloc((void**)&psi2c, S*sizeof(Complex));
235     cudaMalloc((void**)&Normc, S*sizeof(double));
236
237     thrust::device_ptr<double> Normc2=thrust::device_pointer_cast(
238         Normc);
239
240     Complex one(1.,0.);
241     Complex zero(0.,0.);
242     for(i=0;i<S;++i){
243
244         if(i<N || i>=S-N || (i%N)==0 || ((i-(N-1))%N)==0) {
245             psi[i]=zero;
246             psi1[i]=zero;
247             psi2[i]=zero;
248         }
249         else {
250             psi[i]=c(1./((N-2.)*h))*one;

```

```

249     psi1[ i]=c(1./((N-2.)*h))*one;
250     psi2[ i]=c(1./((N-2.)*h))*one;
251     }
252
253 }
254 ofstream out("data4a");
255 for(i=1;i<N-1;++i){
256     for(j=1;j<N-1;++j){
257         out << -L+i*h << "\t" << -L+j*h << "\t" << mod( psi[ j*N+i])
258             *mod( psi[ j*N+i]) << endl;
259     }
260     clock_t t=clock();
261     double v=cpf;
262     int pop=v/cpf;
263
264
265
266     cudaMemcpy(psic , psi , S*sizeof(Complex) ,
267                 cudaMemcpyHostToDevice) ;
268     cudaMemcpy(psi1c , psi1 , S*sizeof(Complex) ,
269                 cudaMemcpyHostToDevice) ;
270     cudaMemcpy(psi2c , psi2 , S*sizeof(Complex) ,
271                 cudaMemcpyHostToDevice) ;
272     cudaMemcpy(Normc , Norm , S*sizeof(double) ,
273                 cudaMemcpyHostToDevice) ;
274
275     dim3 tpb(16, 16, 1);
276     dim3 bpg((N + tpb.x - 1)/ tpb.x, (N + tpb.y - 1) / tpb.y, 1);
277     k4<<<bpg , tpb>>>(psic , psi1c , Normc); //sincronizao de threads
278     cudaThreadSynchronize();
279
280     Nm=thrust :: reduce(Normc2 , Normc2+S);
281
282
283
284     for(n=0;n<nmax;n++){
285         if (!(n%cpf)){
286             out.open("data4a"); // abertura do ficheiro de escrita de
287             dados
288             ofstream script("script4a"); // criacao do script
289             script << "settermupng" << endl;
290             script << "setoutputu\"frame" << n/cpf << ".png\"" <<
291             endl;
292             script << "settitleu\"\" << n/cpf << "\"\" << endl;
293             script << "setuxrangeu[-10:10]" << endl;
294             script << "setuyrangeu[-10:10]" << endl;
295             script << "setucbrangeu[0:*]" << endl;
296             script << "setu sizeusquare" << endl;
297             script << "set xlabelu\"x\" << endl;

```

```

296     script << "set xlabel \"y\" << endl;
297     script << "set xlabel \"|\psi|^2\" << endl;
298     script << "plot \"data4a\" using 1:2:3 no title w p pt 7 ps
299         2 palette" << endl;
300     script.close();
301 }
302 //runge kutta
303 k1<<<bpg , tpb>>>(psic , psi1c , psi2c);
304 cudaThreadSynchronize();
305
306 k2<<<bpg , tpb>>>(psic , psi1c , psi2c);
307 cudaThreadSynchronize();
308
309 k3<<<bpg , tpb>>>(psic , psi1c , psi2c);
310 cudaThreadSynchronize();
311
312 k4<<<bpg , tpb>>>(psic , psi1c , Normc);
313 cudaThreadSynchronize();
314
315 //normalizacao
316 Nm=thrust::reduce(Normc2 , Normc2+S);
317
318 k5<<<bpg , tpb>>>(psic , Nm);
319 cudaThreadSynchronize();
320 Nm=0;
321
322 if (!(n%cpf)){
323     cudaMemcpy( psi , psic , S*sizeof(Complex) ,
324                 cudaMemcpyDeviceToHost );
325     for(i=1;i<N-1;++i){
326         for(j=1;j<N-1;++j){ //escrita no ficheiro de dados
327             out << -L+i*h << "\t" << -L+j*h << "\t" << mod(psi[j*N+i])*
328                 mod(psi[j*N+i]) << endl;
329         }
330         out.close(); //fecho do ficheiro de escrita de dados
331         if (!system("gnuplot script4a")) //criacao do frame
332
333         if (!(n%(pop*cpf))){ // progress bar
334             v=((double)pop*cpf)/(((double)clock()-(double)t)/
335             CLOCKS_PER_SEC);
336             if(v==0) v=1.;
337             cout << "\r" << n*100./nmax << "% " << f=u" << v << "
338             cyc/sec ETA=u" << (int)((((double)nmax-(double)n)/v)
339             /60.) << "min" << (int)((((double)nmax-(double)n)/v)
340             -60*(int)((((double)nmax-(double)n)/v)/60.)) << "sec";
341             cout << "\n[u";
342             for(i=0;i<100*n/nmax;++i) cout << " ";
343             for(i=0;i<100-100*n/nmax;++i) cout << "u";
344             cout << "]";
345             cout << flush;
346             cout << "\e[A\r";
347     }

```

```

342      }
343      if (!(n%(pop*cpf))){
344          t=clock();
345          if (v<100.)
346              pop=((int)v)/cpf;
347              if (pop==0) pop=1;
348      }
349
350 } //libertacao de memoria
351 free(psi);
352 free(psi1);
353 free(psi2);
354 free(Norm);
355
356 cudaFree(psic);
357 cudaFree(psi1c);
358 cudaFree(psi2c);
359 cudaFree(Normc);
360
361 cudaThreadExit();
362
363 cout << endl;
364 cout << endl;
365
366 return 0;
367
368 }

```

Listing 17: Código g25s6c4a