Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Projeto 2 - 'Performance evaluation of parallel computation'

Trabalho realizado por
Grupo 2:
Pedro Alves (up201707234)
Jorge Pacheco (up201705754)
Raul Viana (up201208089)

23 de maio de 2021

Table of Contents

Introdução

O presente trabalho tem como objetivo desenvolver a paralelização de dois algoritmos muito importantes: a multiplicação e a decomposição *LU* de matrizes.

Estes dois algoritmos beneficiam grandemente com a paralelização, pois a parte de código paralelizável é extremamente grande e cada operação pode ser realizada independentemente das outras, até um certo ponto, fazendo com que o ganho em termos de desempenho, seja também muito grande, permitindo assim uma óptima escalabilidade .

Esta paralelização foi obtida através da modificação de algoritmos sequenciais oprimizados para aproveitar da melhor forma as caracteríStcas de localidade e temporalidade da memória cache.

O ambiente utilizado para compilar e correr os algoritmos foi a *API* da Intel “oneAPI” na plataforma *devcloud*. Esta interface pretende ser um meio para que os desenvolvedores possam escrever código único para diferentes arquiteturas. Assim, através desta interface, é possível utilizar um algoritmo único e testá-lo num processador, numa placa gráfica ou mesmo num dispositivo acelerador dedicado, como um FPGA, ou outro.

Foram testadas duas formas de paralelização, utilizando os compiladores DPC++ e openMP. Tanto o primeiro como o segundo permitem paralelizar programas escritos em C e C++. No anexo 1 podem ser encontradas instruções sobre como aceder à platafrma “oneAPI” e como compilar os programas utilizados com estes dois *standards*.

Decomposição *LU*

Para a decomposição *LU*, começámos por desenvolver uma solução que, dado um certo tamanho n, gera uma matriz de n x n elementos, aleatoriamente. Depois é aplicado o algoritmo de decomposição adequado e após esta ter sido feita, a matriz resultado é dividida em 2: a matriz *U*, ou matriz superior, e a matriz *L*, ou matriz inferior.

O algoritmo sequencial utilizado é simples, na medida em que, percorrendo a diagonal principal da matriz, obtemos um *pivot* com o qual os elementos da coluna *pivot* em cada linha se tornam 0. Com este pivot, vamos atualizar todos os elementos das linhas abaixo da atual subtraindo pelo *pivot* calculado anteriormente.

```
bool lu_seq(double matrix[], int size){
    for (int line = 0; line < size; line++)
    {
        for (int col = line + 1; col < size; col++)
        {
            double mul = matrix[size*col + line] / matrix[size*line + line];
            for (int i = 0; i < size; i++)
            {
                matrix[col*size + i] -= mul*matrix[size*line + col];
            }

            matrix[size*col + line] = mul;

        }

    }

}
```

Para separar as matrizes em *L* e *U*, é apenas necessário ter em conta os indíces:

- Caso o índice da linha for maior que o índice da coluna, então estamos na parte inferior da matriz. O valor é colocado na matriz *L* e esse elemento terá o valor 0 na matriz *U*.
- Caso o índice da linha for menor que o índice da coluna, então estamos na parte superior da matriz. O valor é colocado na matriz *U* e esse elemento terá o valor 0 na matriz *L*.

No caso da diagonal principal, ambas as matrizes são preenchidas com 1.

```
    for (int line = 0; line < size; line++)
    {
        for (int col = 0; col < size; col++)
        {
            if (line > col)
            {
                l[line*size + col] = 0;
                u[line*size + col] = matrix[size*line + col];
            }
            else if (line < col)
            {
                l[line*size + col] = matrix[size*line + col];
                u[line*size + col] = 0;
            }
            else
            {
                l[line*size + col] = 1;
                u[line*size + col] = matrix[size*line + col];
            }
        }
    }
    return true;
}
```

O algoritmo de decomposição *LU* por blocos implementado, é parecido com o anterior, no entanto, em vez de calcular a decomposição matricial de uma só na matriz toda, tem em conta o *atual* e "apenas" decompõe esse bloco em cada iteração.

Quanto aos algoritmos paralelos, nós implementámos uma versão com *OpenMP parallel*, e uma outra com *OpenMP shared with tasks*.

Sendo ambas obtidas a partir da modificação da solução sequencial, a primeira foi obtida adicionando `#pragma omp parallel for` no ciclo "for" da decomposição.

Já a versão *shared OpenMP* foi desenhada de uma forma algo mais complexa, adicionando `#pragma omp parallel shared(matrix, size, chunk) private(line, col, mul)` e `#pragma omp for schedule(static, chunk)` bem como `#pragma omp task` nos locais adequados da função de decomposição, nomeadamente nos ciclos "*for*" em que se percorre a matriz.

Como se verá mais à frente, estes algoritmos permitiram, como esperado, um ganho de desempenho superior a 400% (em média, sequencial vs *shared OpenMP*).

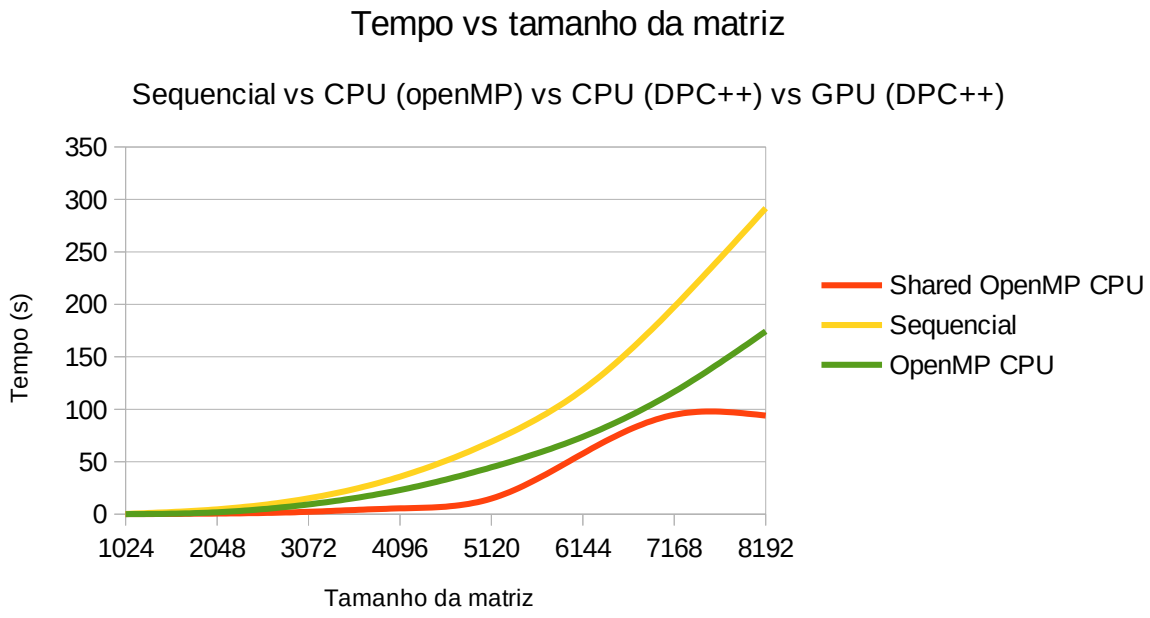


Gráfico 1: tempo vs tamanho da matriz p/ diferentes algoritmos

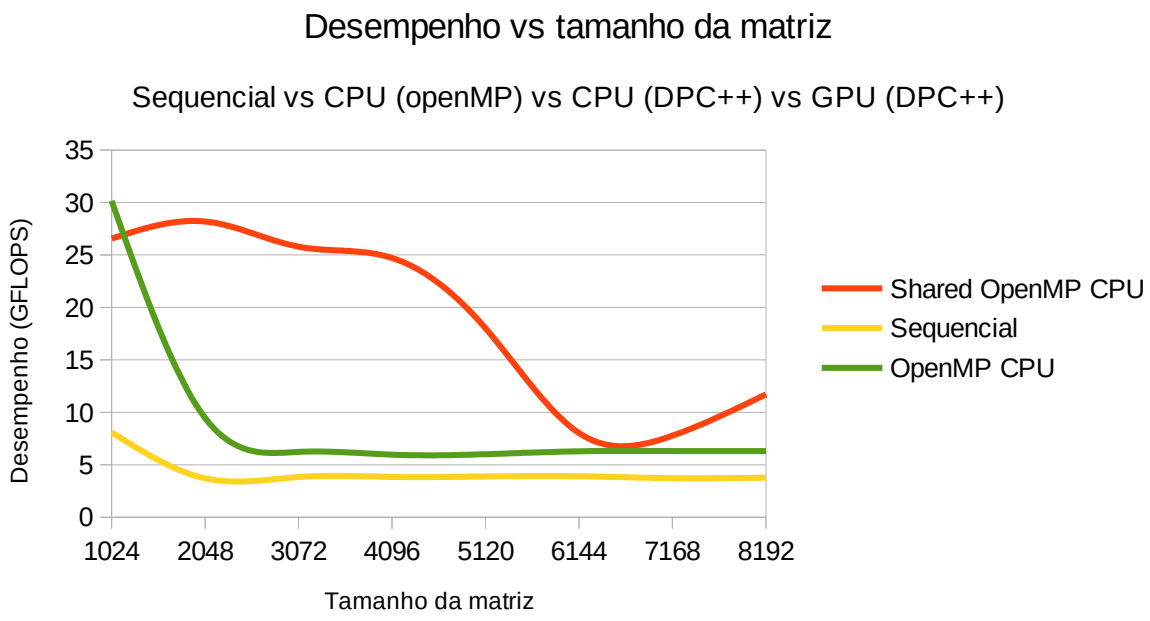


Gráfico 2: desempenho (GFLOPS) vs tamanho da matriz p/ diferentes algoritmos

Relativamente ao desempenho, pela análise dos gráficos 1 e 2, pode-se concluir que o tempo de execução é "cúbico", uma vez que a complexidade temporal é de $O(2/3.n^3)$. Esta complexidade verifica-se para o algoritmo sequencial, mas para a versão *OpenMP* o declive da curva é menor e, como podemos analisar no gráfico 2, o desempenho em GFLOP/s é, em média, cerca do dobro. Já a versão *Shared OpenMP with tasks* supera as restantes, tanto em termos de tempo de execução como em termos de desempenho em GFLOP/s. A paralelização da pivotagem e decomposição da matriz

permite um ganho de 2 vezes relativamente ao algoritmo simples *OpenMP* e de cerca de 4 vezes relativamente ao sequencial (com a exceção da matriz de tamanho 6144 e a de 7168).

Em termos de escalabilidade podemos, portanto, concluir que o algoritmo *OpenMP* deve ser o escolhido para soluções potencialmente grandes, uma vez que permite um ganho potencial de desempenho enorme, bem como redução relativa do tempo de execução. A paralelização do problema, no caso da decomposição *LU*, demonstrou-se positiva. Os resultados da exeução destes algortimos foram obtidos com um processador de 8 cores e 16 threads.

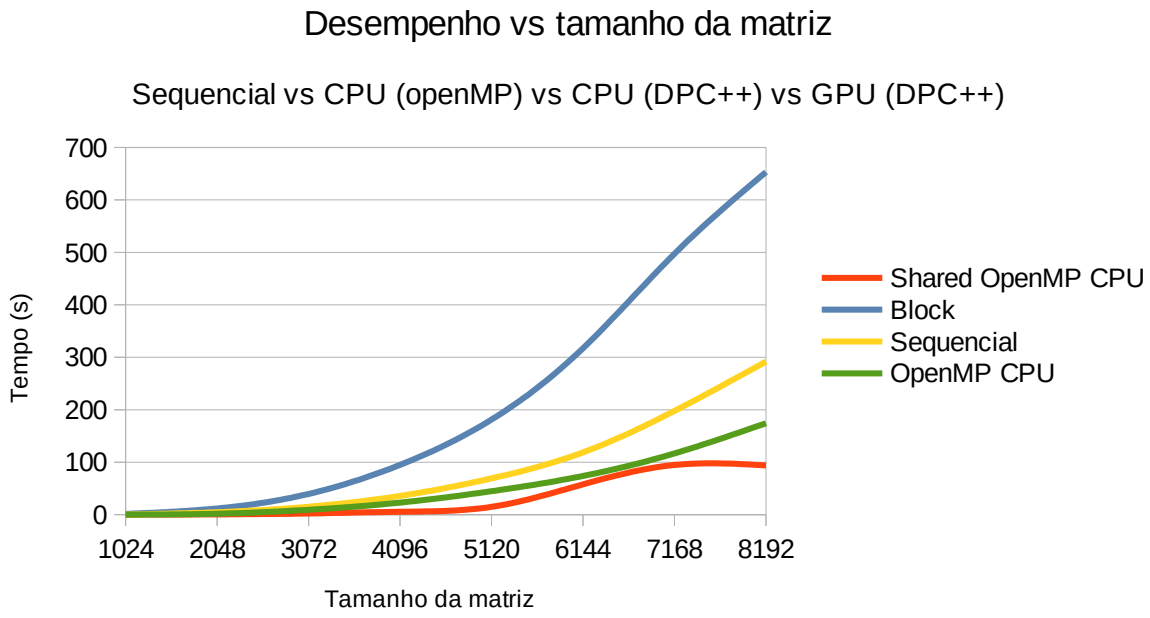


Gráfico 3: tempo vs tamanho da matriz p/ diferentes algoritmos (+block)

Os resultados da execução do algoritmo por blocos, acima representado, acabaram por se revelar os piores em termos de desempenho e escalabilidade. No início pensávamos que este seria o que obteria um melhor resultado do que o sequencial. Isto porque, uma vez que permite "sub-dividir" a matriz inicial em blocos mais pequenos, seria mais eficiente em termos de gestão da memória *cache*. No entanto, não foi isso que se veio a revelar. Este resultado poderá ser explicado uma vez que, ao subdividir a matriz em sub matrizes, o número de cálculos necessários aumenta e a mesma é percorrida mais vezes.

Multiplicação de matrizes

O algoritmo sequencial para a multiplicação de matrizes utilizado neste trabalho foi o mesmo que resultou do trabalho anterior. Este algoritmo já tinha sido optimizado nesse primeiro trabalho, tendo em conta as características de localidade e temporalidade da memória cache.

Para a versão *DPC++*, para além de se ter introduzido uma interface do utilizador, onde é possível escolher o dispositivo a utilizar, foram realizadas mais algumas alterações. Foram criados uma "queueu", "buffers" e "accessors" de forma a poder transferir e manipular os dados no dispositivo escolhido. Foi também implementada uma forma de seleccionar o númeo de *threads*

óptimo para o dispositivo que foi escolhido pelo utilizador e que será utilizado nos calculos seguintes.

Para a implementação da versão *openMP* do algoritmo apenas foi acrescentado um par de linhas “pragma” que permitiram realizar a paralelização do mesmo, e a *interface* do utilizador.

O algortimo de multiplicação de matrizes tem uma percentagem de código paralilizável extremamente grande, na ordem dos 99%. Os calculos efetuados com a versão sequencial resultaram nos resultados apresentados na tabela seguinte.

Grandeza	**Valor**
1 – s	0,998079
s	0,001920
Speedup máximo	520,74

Tabela 1: Valores obtidos para o calculo do Speedup máximo teórico

Como o valor do S*peedUp* é dado pela seguinte formula e assumindo que o número de processadores tende para infinito:

$$SpeedUp = \frac{1}{\frac{1-s}{p} + s} = \frac{1}{s} = 520{,}74$$

Assim é possível afirmar que o tempo de processamento utilizando um número infinito de processadores não vai ser menor do que 1/520 o tempo de processamento sequencial.

Este é um valor bastante alto, o que demonstra que o algoritmo é altamente paralelizável e que o aumento de processadores utilizados diminui de forma consistente o tempo de processamento.

Desta forma é possível pressupor que a multiplização de matrizes tem o potencial de ter um ganho muito significativo com a aplicação da paralelização do algoritmo e dessa forma ser possível escalar o processamento para valores extemamente altos.

No gráfico seguinte estão representados os resultados obtidos com os diferentes algoritmos e nos diferentes dispositivos.

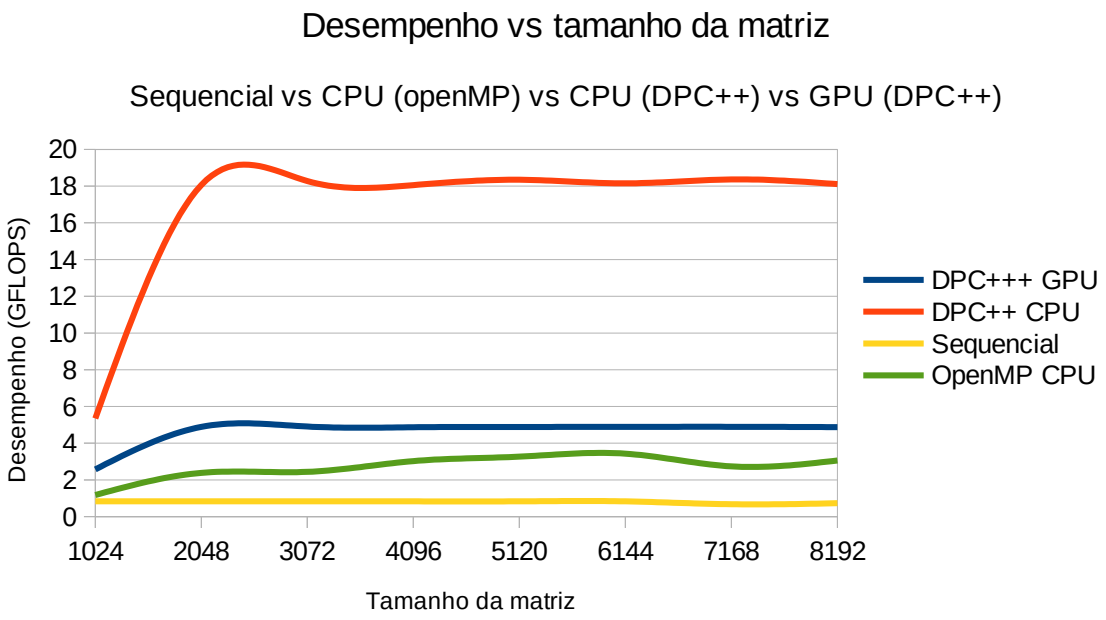


Gráfico 4: desempenho das diferentes versões do algoritmo de multipicação

Os resultados dos tempos de processamento e inerente desempenho das várias versões do algoritmo encontram-se dentro do esperado, com excepção da versão que correu na placa gráfica. Como o processmento deste algoritmo pode ser completamente separado em blocos isolados, que não necessitam de comunicação entre si, apenas que haja uma barreira para as *threads* no fim do processamento de cada bloco, seria de esperar que quanto mais unidades de processamento um dispositivo possua, mais rápido conseguirá processar o algoritmo. Assim seria de esperar uma performance menor na versão sequencial, e performances cada vez melhores na versão openMP, DPC++ num processador típico e na versão DPC++ numa placa gráfica.

A diferença entre a versão openMP e DPC++ deve-se, muito provavelmente, à forma mais fina e detalhada como a criação e distribuição das *threads* e ainda como a gestão da memória no dispositivo foi definida antecipadamente, na versão DPC++. Na versão openMP toda esta gestão foi deixada para o compilador o que leva, necessáriamente, a resultados menos bons.

Dentro das versões DPC++, seria de esperar uma maior performance da versão GPU em relação à versão CPU. Isto devido à possibilidade de grande paralelismo e este se adequar à forma como as placas gráficas estão desenhadas. Não foi, no entanto, este o resultado encontrado. O desempenho do processador foi bastante superior ao desempenho da GPU. Isto pode dever-se ao desenho da plataforma *oneAPI* da Intel, em que a virtualização apresenta um peso muito grande, sendo que os recursos disponibilizados a cada utilizador variam em conformidade com a carga de utilização geral. Pode ainda estar relacionado com um erro no algoritmo implementado. O algoritmo implementado permite ao utilizador escolher o dispositivo em “run-time”. Apesar de se tentar adaptar às diferentes características do dispositivo selecionado, provavelmente existe um erro que não permite aproveitar todos os recursos da placa gráfica, havendo unidades de processamento que não estarão a computar. Deveria ser revisto este ponto.

Em relação à escalabilidade dos algoritmos de multiplicação apresentados é possível afirmar que são também escaláveis. Isto significa que é possível aumentar a dimensão do problema, aumentando o número de processadores e mantendo a eficiência dentro do intervalo entre 0 e 1. No gráfico 5 é possível observar que a curva de iso-eficiência mantém um declive constante de aproximadamente 1, o que significa que aumetando o número de processadores é possível aumentar a dimensão do problema, mantendo a eficiência, o que o torna num algoritmo escalável.

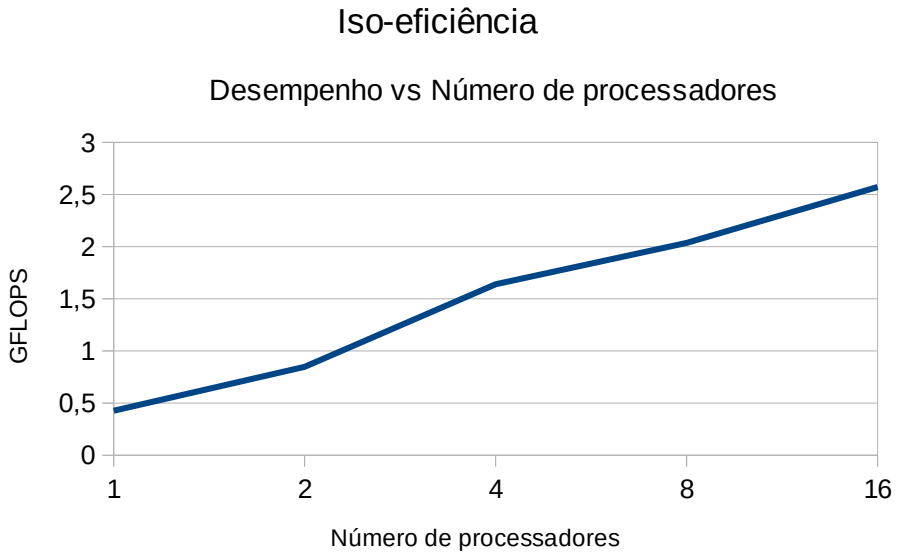


Gráfico 5: Curva de Iso-Eficiência

Para a cosntrução do gráfico de iso-eficiência foi utilizada a versão *openMP* do algoritmo, por ser a versão que apresenta uma maior facilidade no controlo do número de *threads* a criar. O algoritmo foi corrido sucessivamente com um aumento de fator 2 no número de processadores como também no tamanho das matrizes. Desta forma foi possível obter os dados relativos à iso-eficiência e comprovar a escalabilidade do algortimo.

Conclusão

Os objetivos do trabalho foram atingidos, tendo sido desenvolvidos os algortimos de computação paralela propostos partindo das suas versões sequenciais optimizadas.

Existem ferramentas de paralelização da computação que permitem que esta seja realizada em diferentes componentes de um computador pessoal ou outro, podendo aproveitar assim os recursos de *hardware* disponíveis e permitir computar problemas de tamanho considerável em tempo útil.

As medidas de desempenho de paralelização são de extrema importância para predizer o aumento de desempenho e a eficiência de determinado algoritmo paralelo. Estas permitem desenhar e projetar o melhor algoritmo para a máquina em questão, ou por outro lado escalar a máquina para se adaptar da forma mais eficiente ao tamanho do problema.

Anexo 1 – caracterização da plataforma utilizada e manual do utilizador

A plataforma oneAPI da Intel é uma API, standardizada, que pode ser utilizada para correr código em diferentes dispositivos, sejam eles processadores, aceleradores ou outros. Tem o objetivo final de permitir aos desenvolvedores criar, testar e manter código destinado a ser corrido em diferentes tipos de processadores, eliminando assim a necessidade de códigos fonte diferentes, diferentes linguagens de programação e diferentes ferramentas e *workflows* para cada arquitetura.

Para aceder à plataforma será necessário seguir o *link* seguinte:

https://devcloud.intel.com/oneapi/?uuid=44ef7e72-c4c6-432a-9a5f-bb231595d721

De seguida proceder ao *Sign In* com as seguintes credenciais:

username: raulmanuelviana@gmail.com

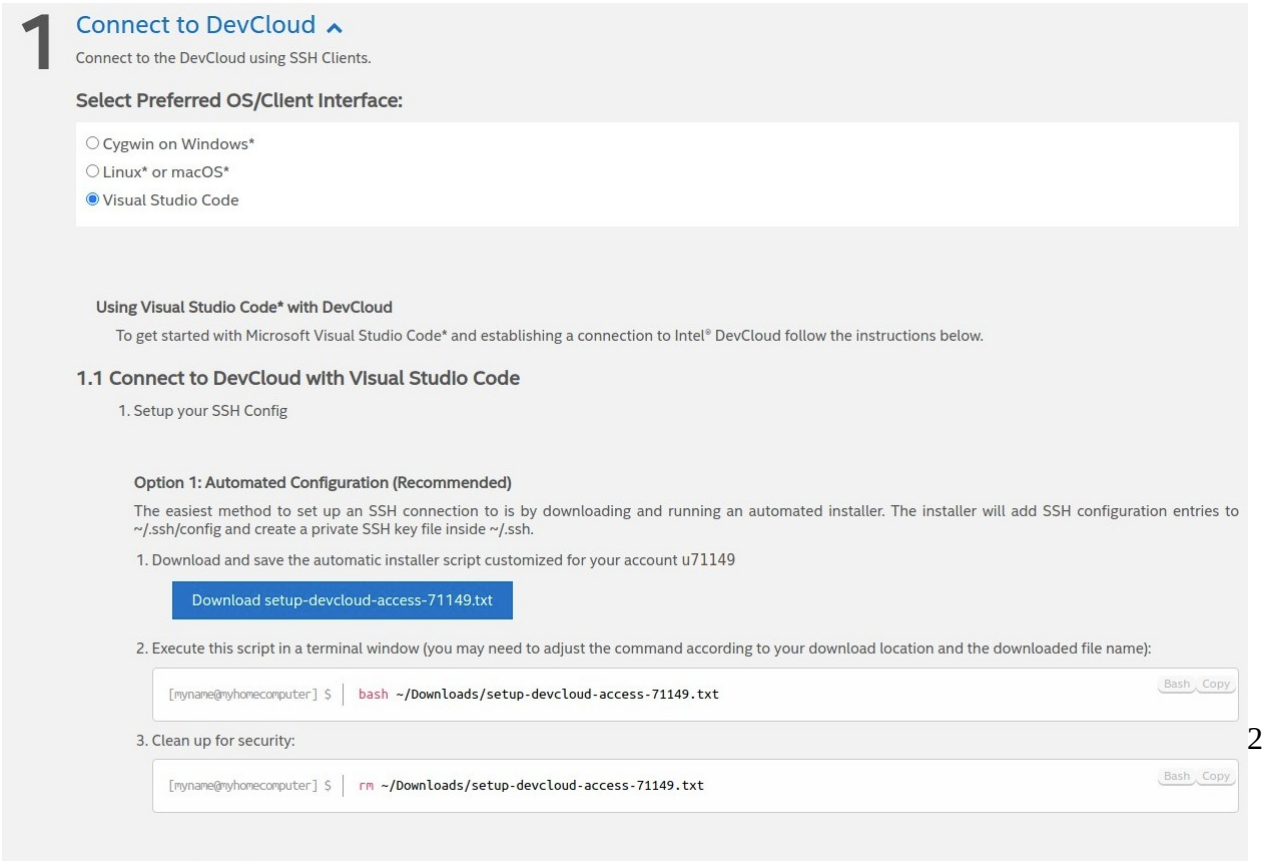
password: feupCpar2021!

Da primeira vez que o utilizador utilizar a plataforma deverá configurar o seu computador para que o acesso se faça da forma mais prática e segura. As imagens seguintes demonstram a melhor forma de poder utilizar a oneAPI.

O manual *online* de onde foram retiradas estas instruções pode ser ecnontrado em:

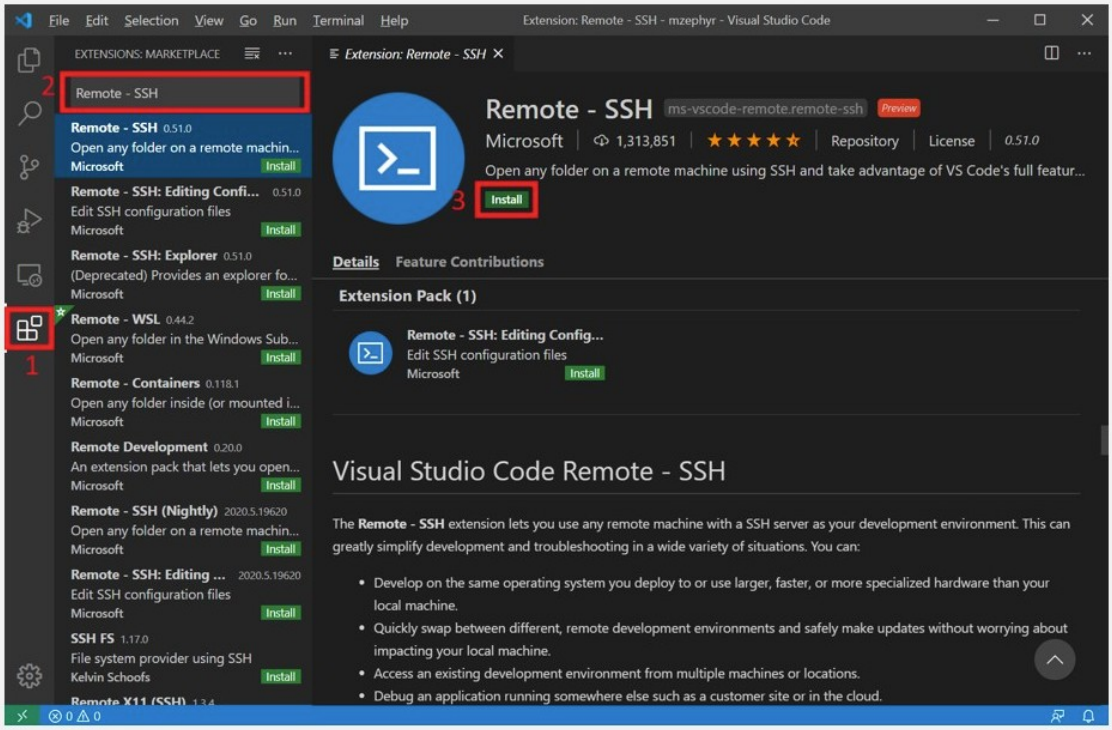
https://devcloud.intel.com/oneapi/get_started/baseToolkitSamples/

Configuração automática das chaves ssh:

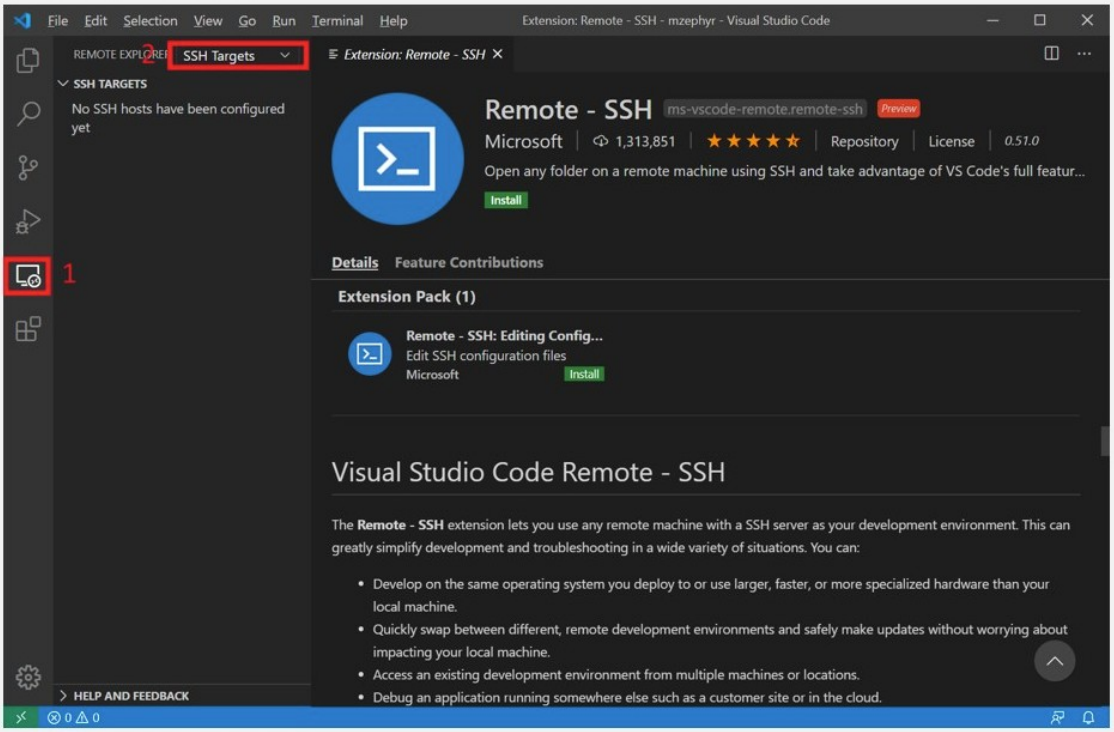


Configuração do VSCode para conexão remota:

3. Install Remote – SSH Extensions from the Visual Studio Marketplace*
   - Click the left menu icon for Extensions
   - Search for **Remote — SSH**
   - Install the **Remote — SSH** Extension



4. Set Remote Connection Panel to SSH Targets
   1. Go to icon on the left side bar for Remote Connections
   2. Remote Explorer should be set to **SSH Targets**



Requisição de um nó computacional que contenha GPU:

Num terminal digitar: **“qsub -I -l node=1:gpu:ppn=2 -d .”**

Port-Fowarding:

7. Port Forwarding
   - Open a new terminal
   - Execute the following command:

```
[myname@myhomecomputer] $ | ssh <compute-node-name>.aidevcloud
```

Bash Copy

Example: If your compute node hostname is `s001-n059`, you would execute the following command:

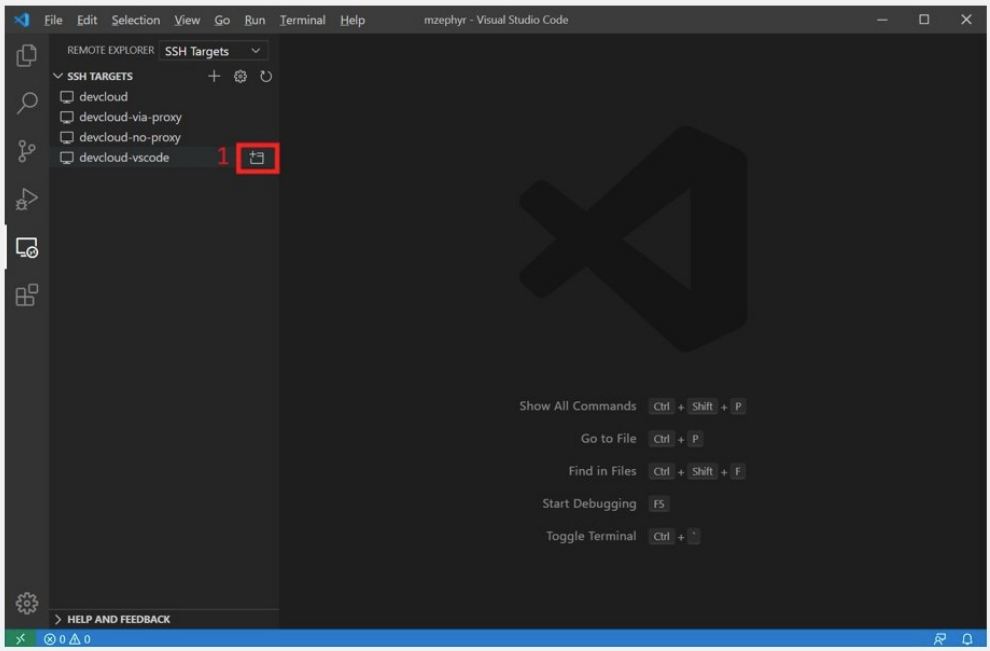
```
[myname@myhomecomputer] $ | ssh s001-n059.aidevcloud
```

Bash Copy

```
localhost:~
$ ssh s001-n059.aidevcloud
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.18 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch
Last login: Tue Sep  1 00:00:00 2020 from 0.0.0.0

u00000@s001-n059:~
$
```

Conetar o VSCode à oneAPI:

8. Connect VS Code* to Intel DevCloud
   - Hover over the **devcloud-vscode** server entry (as shown below) and on the far right of the highlighted entry there should be an icon that will allow you to connect to the server



   - Click the icon to connect. By Default, a new VS Code* Instance should open and connect you to the server

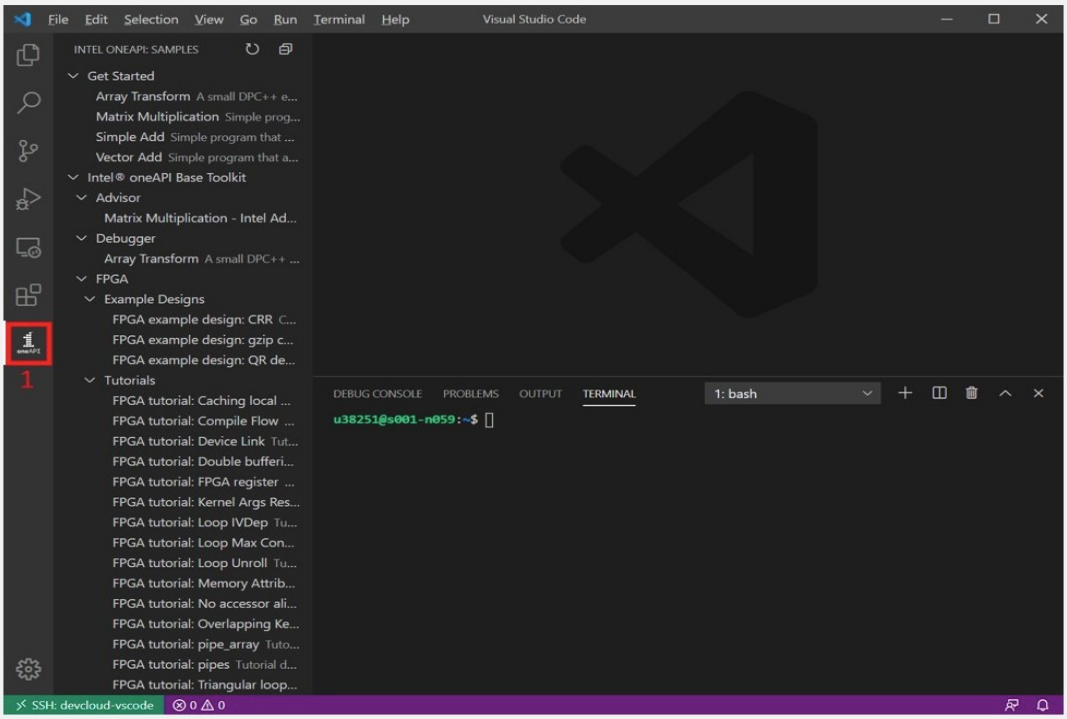
9. Connected to Intel DevCloud
   - You are now connected to the login node of the DevCloud. You can verify this by checking for the SSH Config name, **SSH:devcloud-vscode**, in the lower-left corner.

Utilizar a plataforma e *log out:*

1.2 Sample Browser for Intel® OneAPI Toolkit Extension on DevCloud

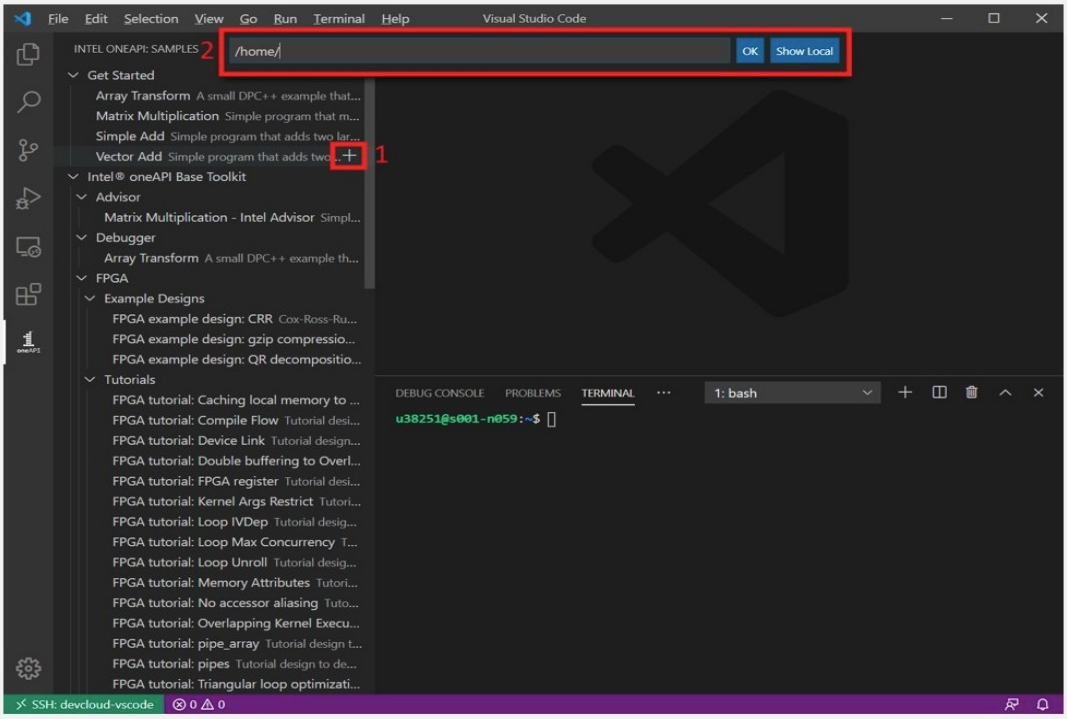
The Sample Browser for Intel OneAPI Toolkits Extension allows you to view code samples directly in Visual Studio Code*. Learn how to install the extension and download your first sample below.

1. In the Extensions Panel, search for **OneAPI** and you will find several extensions
2. Install the **Sample Browser for Intel OneAPI Toolkits** extension by clicking the **Install** button
3. This extension requires a reload of VS Code*, you can do this by clicking on the **Reload Required** button that appeared next to the install button
4. Click on the OneAPI extension icon on the left sidebar shown below:



5. Download your first sample
   Let's choose **Get Started -> Vector Add** sample by following the sequence 1 and 2 shown below
   1. Hover over the sample name and on the far right of the name you will see a plus button appear.
   2. Click on the plus button and you will see a prompt for where you want the sample to be located in.

   Tip: Whichever folder you select, the files will be placed directly in that folder. It is helpful to create a folder with a relevant folder name before selecting a sample so your home directory doesn't become cluttered.



Disconnect Visual Studio Code* from DevCloud

Once completed working on DevCloud you can close the remote connection by selecting **File -> Close Remote Connection** from the VS Code* menu. Alternatively, click the remote-ssh notification in the lower-left corner of the VS Code* window that says **SSH:devcloud-vscode** and select **Close Remote Connection** from the list of Remote-SSH commands

Anexo 2 – dados recolhidos

Decomposição LU

Sequencial		
Tamanho	**Tempo (s)**	**Desempenho (GFLOP/s)**
1024	0,265035	8,10265
2048	4,62945	3,71099
3072	15,1252	3,83349
4096	35,7279	3,84682
5120	69,0606	3,88696
6144	118,688	3,90818
7168	179,702	3,72575
8192	271,596	3,77067

OpenMP CPU		
Tamanho	**Tempo (s)**	**Desempenho (GFLOP/s)**
1024	0,331547	30,1515
2048	4,93907	9,46247
3072	16,2685	6,23145
4096	37,2674	5,96772
5120	72,3223	6,00846
6144	128,994	6,2886
7168	208,475	6,31199
8192	311,356	6,30938

Shared OpenMP CPU		
Tamanho	**Tempo (s)**	**Desempenho (GFLOP/s)**
1024	0,80885	26,5498
2048	0,609485	28,1875
3072	2,24835	25,788
4096	5,56068	24,7162
5120	14,8977	18,0185
6144	57,7711	8,02921
7168	94,7303	7,77562
8192	93,905	11,7088

Multiplicação de matrizes

UHD Graphics P630		
Tamanho	**Tempo (s)**	**Desempenho (GFLOP/s)**
1024	0,83	2,57
2048	3,51	4,89
3072	11,81	4,91
4096	28,23	4,87
5120	54,94	4,89
6144	94,84	4,89
7168	150,49	4,89
8192	225,54	4,88

DPC++		
Tamanho	**Tempo (s)**	**Desempenho (GFLOP/s)**
1024	0,4	5,35
2048	0,95	18,06
3072	3,17	18,28
4096	7,61	18,05
5120	14,63	18,35
6144	25,55	18,15
7168	40,11	18,36
8192	60,71	18,11

Sequencial			
Tamanho	**Tempo Parte Sequencial (s)**	**Tempo Parte Paralelizável (s)**	**Desempenho (GFLOP/s)**
1024	2,57834	2,59131	0,83
2048	20,6218	20,6481	0,83
3072	69,4487	69,5058	0,83
4096	165,166	165,262	0,83
5120	321,731	321,88	0,83
6144	556,48	556,78	0,83
7168	1090,26	1090,55	0,68
8192	1503,33	1503,79	0,73

OpenMP		
Tamanho	**Tempo (s)**	**Desempenho (GFLOPS/s)**
1024	1,82	1,18
2048	7,2	2,39
3072	23,77	2,44
4096	45,43	3,03
5120	82,13	3,27
6144	135,24	3,43
7168	269,12	2,74
8192	359,53	3,06

Eficiency			
Threads	**Tamanho**	**Tempo (s)**	**Desempenho (GFLOP/s)**
1	1024	5,03204	0,426762
2	2048	20,2781	0,847212
4	4096	83,8355	1,63939
8	5120	131,87	2,0356
16	6144	180,381	2,57154