Universidade de Aveiro

Computação Paralela (40779)

FAST Corner Detection Em Imagens Usando OpenMP & CUDA

Gonçalo Freitas
NºMEC: 98012

DETI - Departamento de Electrónica,
Telecomunicações e Informática

Docente:
José Nuno Panelas Nunes Lau

22 de maio de 2023

1. Introdução

1..1 Computação Paralela

Computação paralela é uma abordagem utilizada para resolver problemas computacionais dividindo-os em tarefas menores que podem ser executadas simultaneamente em diferentes processadores ou núcleos de processamento. Esta técnica é amplamente utilizada para acelerar o desempenho e lidar com problemas complexos que exigem um processamento intensivo.

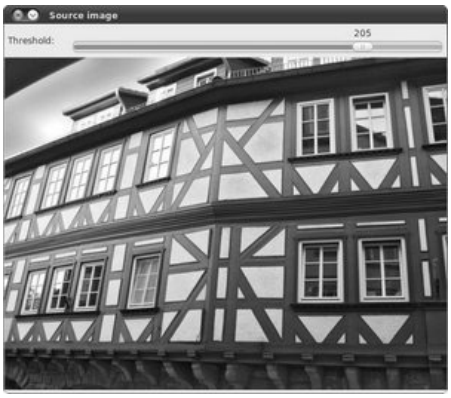
OpenMP, [1], e CUDA [2], são duas tecnologias populares para implementar a computação paralela em diferentes arquiteturas de hardware. A paralelização OpenMP é adequada para sistemas de memória compartilhada (*shared memory*), como CPUs multi-core, enquanto que a paralelização em CUDA é projetada para tirar proveito do poder de processamento das GPUs (unidades de processamento gráfico) da NVIDIA, [3]. Ambas as abordagens podem oferecer um aumento significativo no desempenho e na capacidade de processamento, permitindo a solução mais eficiente de problemas computacionais complexos.

1..2 Problema Proposto

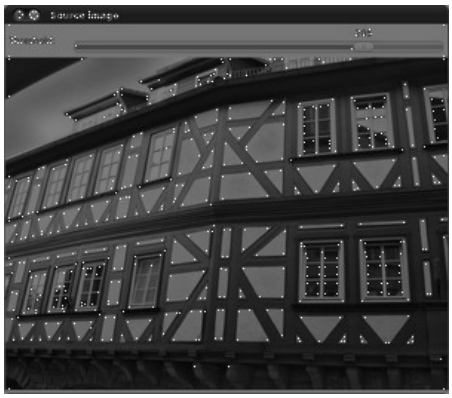
Neste projeto serão processadas imagens, em escala de cinza, com o objetivo de detetar a posição dos cantos. Cada imagem é modelada como uma matriz de inteiros em que cada elemento da matriz corresponde a um pixel na imagem. Os valores da matriz variam de 0 a 255 e especificam a luminosidade do pixel, portanto, um valor de 0 indica um pixel preto e um valor de 255 indica um pixel branco.

A imagem será processada para deteção de cantos utilizando o *FAST Detector*, [4]. Este algoritmo considera como cantos os pixels que têm um certo número de vizinhos consecutivos (ao longo de um círculo de raio 3) que são mais escuros ou mais claros que o canto por uma certa quantidade. O algoritmo pode ser melhorado associando uma pontuação a cada canto detetado e mantendo apenas os cantos cuja pontuação seja um máximo local. O *FAST Detector* tem três parâmetros: o número de pixels consecutivos a contar, a diferença mínima para ser mais escuro (ou mais claro) e se os máximos não locais devem ser eliminados ou não.

Posto isto, o objetivo deste trabalho é resolver este problema usando computação paralela, realizando implementações em OpenMP e CUDA de forma a poder comparar o resultados obtidos, com especial análise para os tempos de execução de cada implementação.



(a) Imagem original



(b) Imagem com o cantos detetados

Figura 1: Aplicação do *FAST Corner Detector* na imagem (a), obtendo-se a imagem (b)

2. Implementações

2..1 OpenMP

Analisando o código fornecido pela docência, é possível verificar que as funções suscetíveis a paralelização são a **fastDetectCorners** e a **nonMaximumSupression**. Na primeira função vemos que existem 2 *for loops* e na segunda existe 1.

É usada então a diretiva `#pragma omp parallel for` para indicar que o *for loop* seguinte deve ser executado em paralelo. É também utilizado a opção que especifica o número de threads que serão usadas, `num_threads(NUM_THREADS)`. O valor da variável `NUM_THREADS` está definido globalmente no inicio do código.

A cláusula `private(i,j)` declara que as variáveis i e j são privadas para cada thread, ou seja, cada thread terá sua própria cópia dessas variáveis e as alterações feitas por uma thread não afetarão o valor dessas variáveis noutra thread. Esta cláusula é utilizada em todos os *for loops*.

A cláusula `reduction(+:count)` é usada para que a variável *count* seja compartilhada entre todas as threads e cada thread possa atualizar a sua cópia local dessa variável, sem que haja conflitos. No final da execução do *for loop*, o valor de *count* de cada thread é somado em uma operação de redução + e o resultado final é armazenado na variável global *count*. Já esta cláusula é apenas utilizada no *for loop* da função **fastDetectCorners** e no 2º *for loop* da função **nonMaximumSupression**. Esta cláusula não era, no entanto, necessária pois a variável global *count* apenas serve para contar o número de *features* e este não é relevante para verificar se o resultado (imagem com os cantos assinalados) está certo ou não.

Assim, estas cláusulas garantem que não haja conflitos no acesso, pelas múltiplas threads, às variáveis i, j e *count*. Após estas alterações ficamos com versões paralelizadas das funções **fastDetectCorners** e **nonMaximumSupression**, denominadas no código como **fastDetectCornersOPENMP** e **nonMaximumSupressionOPENMP**, respetivamente.

Para além destes ciclos, foi também realizada a paralelização do *for loop* que se encontra na função **fastDetectorOpenMP**, responsável por adicionar a imagem original desfoca como fundo, usando a cláusula `private(i,j)` e especificando o número de threads que serão usadas, tal como referido em cima. De resto, esta função é um cópia da **fastDetectorHost**.

2..2 CUDA - Global Memory

Tal como na implementação da paralelização usando OpenMP, as funções **fastDetectCorners** e **nonMaximumSupression** precisam de ser adaptadas e é preciso criar uma função **fastDetectorDevice** de raiz. Contudo, nesta implementação em CUDA é também necessário antes da declaração das funções **fastCorner** e **fastScore**, usar `__host__ __device__`, de forma a permitir que estas sejam executadas tanto no *host* (CPU) quanto no *device* (GPU), isto pois, a ausência destes *function qualifiers* limita a execução da função apenas ao *host*. Foi também adicionado o modificador `__managed__`

aquando da definição da variável `offset` (linha 28) de forma não ser preciso realizar a alocação e desalocação manual da memória para a esta variável.

Visto as funções **fastDetectCorners** e **nonMaximumSupression** serem as que queremos executar em paralelo, temos que criar versões *kernel* CUDA destas, sendo o *kernel* da primeira denominado como **fastDetectCorners_CUDA**. Já a segunda função, foi dividida em dois *kernel* denominados por **nonMaximumSupressionKernel_p1** e **nonMaximumSupressionKernel_p2**. Antes da definição destes 3 *kernels* é usado o modificador `__global__` para indicar que a função será executada na GPU e tem acesso ao espaço de memória global.

Nestes 3 *kernels*, a cada *thread* CUDA é atribuída uma posição específica na imagem, ou seja, cada thread do *kernel* é responsável por processar um pixel da matriz de entrada. A posição da thread atual ao longo do eixo x dentro da *grid* pode ser calculada multiplicando `blockIdx.x` por `blockDim.x` e adicionando `threadIdx.x`. Esta posição é atribuída à variável `x`. De forma análoga, os mesmos cálculos são aplicados para as variáveis `blockIdx.y`, `blockDim.y` e `threadIdx.y` de forma a obter a posição da thread ao longo do eixo y. Sendo esta posição atribuída à variável `y`. Estas variáveis `x` e `y` são usadas para calcular o índice correspondente, `idx`, das matrizes respetivas a cada *kernel*.

Relativamente a **fastDetectCorners_CUDA**. Após o cálculo do `idx`, é aplicada uma condição para verificar se o pixel atual está dentro de uma região válida para a deteção de cantos (3 pixels de margem em cada borda da imagem). Se a condição for verdadeira, a função **fastCorner** é chamada com o pixel atual e os parâmetros de deteção. Caso o pixel seja classificado como um canto, este é marcado como o valor `MAX_BRIGHTNESS` na matriz `h_odata`.

Já em relação aos *kernels CUDA* da função original **nonMaximumSupression**, o primeiro, **nonMaximumSupressionKernel_p1**, é responsável por verificar se o pixel é um canto brilhante, isto é, com o valor `MAX_BRIGHTNESS` e, em caso afirmativo, calcula o *score* correspondente usando a função **fastScore**, armazenando o resultado no *array* `corner_score`. Já o segundo, **nonMaximumSupressionKernel_p2**, realiza a etapa de supressão de não-máximos, isto é, se algum dos vizinhos tiver um *score* maior ou igual ao *score* do pixel atual, o pixel atual é suprimido (atribui-se o valor 0) e a *thread* passa para o próximo pixel. Caso contrário, o pixel atual é mantido como um canto e o valor `MAX_BRIGHTNESS`.

Quanto à função **fastDetectorDevice**, esta, ao contrário da implementação em OpenMP já não é uma cópia exata da **fastDetectorHost** e pode ser descrita como de seguida. Inicialmente, aloca-se memória na GPU para os dados de entrada, utilizando a função **cudaMalloc**. Estes são, posteriormente, copiados da memória do *host* (CPU) para a memória do *device* (GPU) com a função **cudaMemcpy**.

As dimensões dos blocos e da *grid* de execução do *kernel CUDA* são calculadas, levando em consideração as dimensões dos blocos e a cobertura da imagem pela *grid*. Em seguida, o *kernel CUDA* **fastDetectCorners_CUDA** é lançado e, após sua execução, a função **cudaDeviceSynchronize()** é chamada para garantir a sincronização das operações na GPU.

Se a opção $-m$ for selecionada durante a execução, memória adicional é alocada

na GPU para arrays auxiliares, `dev_aux` e `corner_score`, sendo o primeiro copiado da memória do *host* para a memória do *device*. Os *kernels CUDA* **nonMaximumSupressionKernel_p1** e **nonMaximumSupressionKernel_p2** são lançados para realizar a supressão de não máximos. Novamente, a função **cudaDeviceSynchronize()** é chamada entre os lançamentos destes *kernels* e, após a execução, a memória alocada para os *arrays* auxiliares é liberada com **cudaFree**.

Por fim, os dados de saída, `d_odata` são copiados da memória do *device* para a memória do *host* e, assim como na **fastDetectorHost**, a imagem original desfocada é adicionada como plano de fundo.

3. Resultados

De maneira a obter resultados de *speedups* mais corretos, foi desenvolvido 1 ficheiro *bash*, para cada implementação, que, para cada imagem, execute o código 10 vezes. Assim, os valores de *speedups* apresentados e referidos neste projeto representam a média destes 10 valores. Estes ficheiros foram devidamente anexados [1].

3..1 OpenMP

Como executar

Para executar o código desenvolvido para a implementação em OpenMP primeiramente escreve-se, na linha de comandos, o comando `make`, que faz uso do ficheiro **MakeFile**, fornecido pela docência, para compilar o ficheiro **fastDetectorOpenMP.cpp**. De seguida, utiliza-se o ficheiro *bash*, **run_OpenMP.sh**, para executar o ficheiro executável criado pelo comando `make`. Para guardar o *output* para um ficheiro de texto escreve-se, na linha de comandos, `./run_OpenMP.sh > "full_output_OpenMP.txt"`. Caso se queira usar a opção $-m$ o ficheiro *bash* a correr é o **run_OpenMP_m.sh** procedendo, de resto, de igual forma a quando não se considera esta opção.

Discussão

Usando os valores de interesse, `Host` e `OpenMP processing time`, guardados no ficheiro de texto, acima referido, calculando a média para cada imagem, foi possível obter os valores apresentados na Tabela 1. A partir desta vemos que a implementação desenvolvida em OpenMP apresentou sempre melhoramento do tempo de execução tendo sido, em média, obtido um speedup de 3.1250 (ms), o que apesar de ser pouco já se pode considerar como um resultado positivo. De referir que, para a execução do código definiu-se a variável `NUM_THREADS` como 16. Em relação a razão pela qual a imagem *chessRotate1* obteve um *speedup* mais baixo, esta será discutida mais a frente.

Por fim, é necessário comparar as imagens resultantes entre a implementação em OpenMP e a solução correta. Para isto é preciso executar, na linha de comandos,

[1] No Appendix encontra-se explicado, resumidamente, o que se pode encontrar no ficheiro `.zip` que foi enviado

Tempo (ms) / Imagem	Host	OpenMP	*Speedup*
building	14.6058	2.8576	5.1112
chessBig	57.7468	16.4553	3.5093
chessRotate1	1.8639	1.4923	1.2490
house	4.4351	1.2116	3.6605
squares	2.3158	1.1053	2.0952

Tabela 1: Tempos de execução (médios) e respetivos *speedups* (ms) para cada imagem, `NUM_THREADS` = 16, sem supressão de não máximos

o seguinte, `./testDiffs referenceOpenMP.pgm resultOpenMP.pgm`. Ao fazer uso do ficheiro *bash* referido anteriormente, `run_OpenMP.sh`, este comando é também executado pelo que podemos então analisar o ficheiro de texto **full_output_OpenMP.txt**, para verificar o resultado deste, onde vemos que, para cada imagem, em todas as 10 *runs* o resultado é **"images are identical"**, o que significa que a solução obtida com esta implementação vai de acordo com a solução correta.

Já se considerarmos a opção em que é realizada a supressão de não máximos, é esperado que os tempos de execução, tanto do *host* como da implementação em OpenMP aumentem, devido ao algoritmo ser computacionalmente mais expendioso com esta opção. Com este aumento do tempo de execução é esperado com que o speedup também aumente pois o *overhead* da criação de threads torna-se menos significativo. Isto é exatamente o que se pode verificar pela Tabela 2, tendo-se obtido um *speedup* médio, 3.6976 (ms), maior do que o valor obtido quando não se considerou esta opção. Em relação aos resultados obtidos ao usar esta opção, ao analisar o ficheiro **full_output_OpenMP_m.txt**, obtido através do ficheiro *bash* `run_OpenMP_m.sh`, reparamos que o resultado é, para todas as imagens e para todas as *runs*, **"images are identical"**, pelo que podemos concluir que a implementação esta correta, também quando se considera esta opção.

Tempo (ms) / Imagem	Host	OpenMP	*Speedup*
building	17.3022	4.1099	4.2099
chessBig	65.8781	19.3483	3.4049
chessRotate1	2.4604	0.9776	2.5168
house	6.2016	1.2808	4.8420
squares	6.2016	1.2808	2.5142

Tabela 2: Tempos de execução (médios) e respetivos *speedups* (ms) para cada imagem, `NUM_THREADS` = 16, com supressão de não máximos

Posto isto, visto a imagem resultante ser sempre igual ao resultado correto e existir sempre *speedup*, podemos concluir que a nossa implementação em OpenMP está a funcionar corretamente e pode ser utilizada em vez da implementação normal, não paralelizada.

Influência do número de *threads*

Tal como referido anteriormente, foi-se optado por definir a variável `NUM_THREADS` como 16. No entanto, usar o maior número *threads* possível nem sempre é vantajoso e geralmente escolhe-se um valor com base no número de iterações sendo esse, neste caso em particular, a altura da imagem, visto estarmos a paralelizar o *for* correspondente a esse valor. Ou seja, caso o número de iterações for pequeno deve-se considerar um número mais baixo de *threads*, caso contrário, deve-se optar por um número mais elevado.

Na Tabela 2, encontra-se os *speedups* (médios) para cada imagem, bem com o seu tamanho, variando o parâmetro `NUM_THREADS` como $3, 8, 16, 24$. Olhando primeiramente para os *speedups* obtidos quando se usa 24 *threads*, reparamos que apenas nas imagens *building* e *chessBig* é que a implementação obteve melhoramentos no tempo de execução, tendo-se isto devido a serem imagens de grande altura. Relativamente as restantes imagens, visto serem imagens de menor altura, o *overhead* da criação de threads torna-se significativo, ou seja, o tempo de criação e coordenação das *threads* é tão elevado que torna a paralelização menos eficiente do que fazer uso de *single thread*, o que não acontece quando consideramos uma valor mais baixo para `NUM_THREADS`, como 3 e 8, onde vemos que foi sempre obtido um *speedup*. Assim, podemos concluir que, tal como referido anteriormente, a escolha do número *threads* deve ter em conta a complexidade computacional do problema em si e não só o maior número *threads* disponíveis.

Imagem \ Nº *threads*	3	8	16	24	Tamanho
building	2.2980	4.7187	5.1112	1.5932	700x466
chessBig	2.0991	3.7149	3.5093	3.2953	1550x1336
chessRotate1	2.3221	4.0902	1.2490	0.1348	200x200
house	2.4933	5.0484	3.6605	0.4565	300x262
squares	2.2059	3.8319	2.0952	0.2795	320x240

Tabela 3: *Speedups* (médios) para cada imagem, variando o parâmetro `NUM_THREADS`, sem supressão de não máximos

3..2 CUDA

Como executar

Tal como a implementação em OpenMP, foram criados dois ficheiros *bash*, **run_CUDA.sh** e **run_CUDA_m.sh**, que executam o código, repetindo 10 vezes para cada imagem, apenas sendo preciso escrever primeiro, na linha de comandos, o comando `make`.

É dado a opção de usar duas GPU's para se executar o código, sendo estas definidas pelo `deviceId`, que pode tomar o valor 0 ou 1. A valor 0 está associado uma gráfica `GeForce GTX 1660 Ti` e ao valor 1 uma gráfica `GeForce GT 710`. Desta forma, sabendo que o poder de processamento da gráfica cujo `deviceId` $= 0$ é maior, é de esperar que esta obtenha tempos de execução menores e, por sua vez, maiores *speedups*. Esta informação foi retirada a partir do comando `nvidia-smi`. Em relação as dimensões dos blocos de *threads* assumiu-se uma dimensão de 16x16.

Discussão

Nas tabelas seguintes temos representados os tempos de execução (médios) e respetivos *speedups* (ms) para cada imagem, obtidos em ambas as gráficas sem, Tabelas 4 e 5, e com supressão de não máximos, Tabelas 6 e 7.

Comparado a Tabelas 4 e 5 e as Tabelas 6 e 7, vemos que, tal como referido acima a gráfica cujo `deviceId` = 0 obteve tempos de execução menores e, por sua vez, maiores *speedups*. Também é possível reparar que, para a imagem *chessRotate1* quando `deviceId` = 1 não foi possível obter *speedup* com e sem supressão de não máximos. Isto permite-nos concluir que o *speedup* obtido varia consoante a GPU utilizada, tal como seria de esperar.

Por fim, se compararmos as Tabelas 4 e 6 e as Tabelas 5 e 7, podemos concluir, tal como na implementação em OpenMP, que os tempos de execução (médios) no *host* e no *device* aumentaram quando se realiza a supressão de não máximos.

Tempo (ms) / Imagem	Host	CUDA	*Speedup*
building	14.8733	1.5143	9.8219
chessBig	65.3904	7.1319	9.1687
chessRotate1	1.5307	0.4011	3.8165
house	4.5291	0.5920	7.6511
squares	2.4411	0.5007	4.8753

Tabela 4: Tempos de execução (médios) e respetivos *speedups* (ms) para cada imagem, `deviceId` = 0, sem supressão de não máximos

Tempo (ms) / Imagem	Host	CUDA	Speedup
building	11.6843	7.0977	1.6462
chessBig	59.4909	38.6257	1.5401
chessRotate1	1.26996	1.4545	0.8730
house	3.3455	2.2152	1.5102
squares	2.0682	2.0058	1.0311

Tabela 5: Tempos de execução (médios) e respetivos *speedups* (ms) para cada imagem, `deviceId` = 1, sem supressão de não máximos

Tempo (ms) / Imagem	Host	CUDA	Speedup
building	16.7460	1.9780	8.4660
chessBig	72.6846	7.3454	9.8952
chessRotate1	1.9953	0.4810	4.1485
house	5.4555	0.6683	8.1638
squares	2.4549	0.5792	4.2382

Tabela 6: Tempos de execução (médios) e respetivos *speedups* (ms) para cada imagem, `deviceId` = 0, com supressão de não máximos

Imagem \ Tempo (ms)	Host	CUDA	Speedup
building	15.0055	10.3997	1.4429
chessBig	72.8277	37.7092	1.9313
chessRotate1	1.6897	1.8157	0.9306
house	4.5249	3.4469	1.3127
squares	2.5507	1.9671	1.2966

Tabela 7: Tempos de execução (médios) e respetivos *speedups* (ms) para cada imagem, `deviceId` = 1, com supressão de não máximos

Em relação as imagens que se obtém ao usar executar este código sem e com supressão de não máximos, podemos analisar os ficheiros de texto resultantes da execução dos ficheiros *bash*, para cada `deviceId`, onde vemos que o resultado é, para todas as imagens e para todas as *runs*, **"images are identical"**.

Desta forma, visto a imagem resultante ser sempre igual ao resultado correto e existir quase sempre *speedup*, podemos concluir que a nossa implementação em CUDA está a funcionar corretamente e pode ser utilizada em vez da implementação não paralelizada.

Influência da dimensão dos blocos

Semelhante ao que foi visto em OpenMP, a partir da Tabela 8, vemos que nem sempre optar por tamanhos maiores de blocos é positivo, principalmente para imagens mais pequenas. Desta forma, podemos concluir que, para a escolha das dimensões do blocos, devemos não só ter em consideração o tamanho máximo dos blocos disponível mas também a complexidade computacional do problema em si.

Imagem \ Dimensão dos blocos	4x4	8x8	16x16	32x32	Tamanho
building	7.5890	8.5153	9.8219	9.9203	700x466
chessBig	8.5990	9.8632	9.1687	9.8146	1550x1336
chessRotate1	3.5671	4.1130	3.8165	3.4592	200x200
house	7.6992	6.9347	7.6511	6.8577	300x262
squares	4.6782	4.9892	4.8753	5.0092	320x240

Tabela 8: *Speedups* (médios) para cada imagem, variando as dimensões dos blocos, sem supressão de não máximos

3..3 Comparação entre implementações

Comparando ambas as implementações, podemos concluir que, a implementação em CUDA, com `deviceId` = 0, foi a que apresentou melhores resultados, em termos de *speedup*, o que nos leva a afirmar que, quando o poder de processamento da GPU é superior à capacidade de processamento paralelo do CPU, a paralelização em CUDA mostra-se vantajosa. No entanto, caso estejamos perante um problema em que não seja necessário obter um *speedup* elevado, como o caso deste, em que o tempo de execução era

na ordem de grandeza dos milissegundos, a escolha de que implementação utilizar poderia recair na implementação em OpenMP devido a sua mais fácil e direta implementação.

4. Conclusão

Com base nos resultados obtidos na implementação em OpenMP foi possível concluir que esta trouxe vantagens no tempo de execução mas que é preciso ter em consideração a complexidade computacional do problema, de forma a escolher um número de *threads* que maximize o *speedup* obtido. Já em relação a implementação CUDA a conclusão que se pode fazer é semelhante, ou seja, apesar de se mostrar um método vantajoso para diminuir o tempo de execução do código, quando o poder de processamento da GPU é superior ao do CPU, é também preciso fazer uma escolha da dimensão dos blocos informada.

Comparando ambas as implementações podemos concluir que, a implementação em CUDA, com `deviceId` = 0, apresentou os melhores resultados em termos de *speedup*, pelo que podemos considerar a paralelização em CUDA vantajosa, quando o poder de processamento da GPU é superior à capacidade de processamento paralelo do CPU. No entanto, caso estejamos perante um problema em que não seja necessário obter um *speedup* elevado, ou seja, em que a ordem de grandeza do tempo de execução seja baixa, a implementação em OpenMP pode ser considerada a escolha mais vantajosa devido a sua mais fácil e direta implementação.

Por fim, com este projeto, foi possível concluir que a paralelização é algo vantajoso no desenvolvimento de código pois permite obter tempos de execução mais baixos, fazendo uso dos recursos disponíveis que, por muitas vezes, não são utilizados.

Referências

[1] OpenMP. The OpenMP API specification for parallel programming. https://www.openmp.org/

[2] CUDA. https://developer.nvidia.com/cuda-toolkit

[3] NVIDIA. World Leader in Artificial Intelligence Computing. https://www.nvidia.com/en-us/

[4] Machine Learning for High-Speed Corner Detection. Rosten, E., Drummond, T. (2006). In: Computer Vision – ECCV 2006. ECCV 2006. LNCS, vol 3951. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11744023_34

[5] NVIDIA CUDA C++ Programming Guide, v12.1, NVIDIA

[6] CUDA C++ BEST PRACTICES GUIDE, v12.1, NVIDIA

Appendix

Conteúdo do ficheiro enviado

No ficheiro `.zip` enviado, para além das imagens `.pgm` e o código **testDiffs.cpp**, são anexados os ficheiros dos códigos alterados das implementações em OpenMP e em CUDA, denominados por **fastDetectorOpenMP.cpp** e **fastDetectorCuda.cu** , respetivamente. Os ficheiros *bash*, **run_OpenMP.sh**, **run_OpenMP_m.sh**, **run_CUDA.sh** e **run_CUDA_m.sh** também se encontram no ficheiro `.zip`. Estes, tal como referido anteriormente servem para executar o código com e sem supressão de não máximos para cada imagem 10 vezes, de forma a obter resultados mais corretos. A pasta `Common`, fornecida pela docência, também se encontra anexada. Por fim é também anexada duas pastas, `Dados OpenMP` e `Dados CUDA`, onde se pode encontrar os ficheiros `.txt` com os dados utilizados para calcular os valores apresentados nas tabelas deste projeto.

`Dados OpenMP`

Nesta pasta encontramos um total de 8 pastas. Os nomes destas são do tipo `xT` e `xT_m`, onde `x` representa o número de threads e as pastas com `_m` no nome significa que foi usada a opção de supressão de não máximos.

`Dados CUDA`

Nesta pasta encontramos 2 sub-pastas.

`Sem -m`

Tal como o nome indica, nesta sub-pasta encontram-se os valores obtidos quando não se considera a opção de supressão de não máximos. Nesta, encontra-se um total de 5 ficheiros de texto, cujo nome é do tipo `devX_Y_full_output_CUDA`, onde `X` representa o *id* do *device* (0 ou 1) onde foi executado o código e `Y` a dimensão dos blocos utilizados, sendo estes sempre de dimensão `YxY`.

`Com -m`

Já nesta sub-pasta encontram-se os valores obtidos quando se considera a opção de supressão de não máximos. Nesta, encontram-se 2 ficheiros `.txt`, cujo nome é do tipo `devX_full_output_CUDA`, onde `X` representa o *id* do *device* onde foi executado o código.