RELATÓRIO DO TERCEIRO EXERCÍCIO DE PROGRAMAÇÃO PARALELA E CONCORRENTE

Relatório por
Guilherme Lopes – 52761

- ## Quais foram as modificações necessária para a execução no GPU?

  Na função atribuída ao GPU foi escrito o código para a inicialização em *loop* do *Kernel*. Após isso, foi criado o *Kernel* (identificado com o rótulo *__global*) com toda a lógica da resolução do problema, agora em paralelo entre todas as *threads* atribuídas. No *Kernel*, ao contrário da versão em *loops* sequenciais no CPU, a abordagem tomada foi de se paralelizar os *loops* de acordo com a organização em blocos que o GPU oferece. Utilizando a mesma "linguagem" de nomes que a versão sequencial, ou seja, variáveis representantes *i* e *j*, foi atribuído a cada *i* o número de um bloco correspondente e a cada *j* uma *thread* desse mesmo bloco e devido ao facto de o problema oferecer a flexibilidade de executar as iterações de *i* e/ou *j* fora de ordem, é possível cada bloco e *thread* funcionarem isoladamente, independentemente da ordem em que cada um executa. Para as iterações de *k* foi utilizado o valor de cada iteração de chamada ao *Kernel*, garantindo a sincronização de blocos, ou seja, que *k* é executado na sua totalidade antes de *k + 1*. É também garantido que cada bloco não fica sem trabalho e que *i* e *j* percorrem ambos todo o intervalo [0, *graph_size – 1*] no caso de *i*, *para* cada iteração *k* e no caso de *j*, para cada iteração *i*.

- ## Quais foram as escolhas de parâmetros para a execução no GPU e o uso/transferência de memória?

  Inicialmente foi necessário a alocação da memória necessária no GPU (*cudaMalloc*) e a transferência de conteúdo do grafo para um vetor a ser utilizado e alterado independentemente dentro do *Kernel (cudaMemcpy)*. O número de *threads* por bloco foi definido como 32, em que os testes demonstram melhores resultados possivelmente por ocorrer uma redução no impacto da divergência de *threads* em cada bloco, e o número de blocos depende do tamanho do grafo sendo então o tamanho do grafo dividido pelo número de *threads* por bloco. Se o tamanho do grafo não for múltiplo de 32, vai ser atribuído um bloco a mais do que o necessário e dentro do *Kernel* será filtrado se certas *threads* devem fazer trabalho ou não. Dentro do *Kernel* foi decidido que apenas se usaria blocos de *threads* de uma dimensão, pois foi verificado que, não apenas era desnecessária a utilização de duas ou mais dimensões para a resolução proposta, como também se demonstrou, ao testar, mais lento do que a resolução de blocos de uma dimensão apresentada. Não foi utilizado o tipo de memória partilhada devido às razões referidas no fim do relatório, apenas memória global. Também não foram utilizadas operações atómicas, pois, em cada *k*, um espaço de memória é apenas acedido uma vez, ou seja, não existe problemas de concorrência. No final do(s) *Kernel* executarem todos, será transferido o resultado para a variável correspondente no CPU (*cudaMemcpy*) e finalmente é realizado *cudaFree* nas variáveis utilizadas no *Kernel* (GPU).

- ## O programa sofre de divergência de *threads*?

  Sim, isto porque existem *threads* que acabam por possivelmente validar uma das duas condição *if* do algoritmo e assim executar linhas de código que outras *threads* não executam na mesma iteração,

sendo no primeiro *if* algo mais visível e problemático porque existirá *threads* sem nenhum trabalho do inicio ao fim. Naturalmente as *threads* por bloco estão sincronizadas e, em casos como o referido, as *threads* que não validam a condição necessitam de aguardar pelas *threads* que validam, porém as *threads* entre *warps* diferentes simplesmente avançam no código, não garantindo que todas as *threads* em um bloco estejam sincronizadas, isto é resolvido com a utilização da função *syncthreads*. No caso da resolução apresentada este problema não ocorre, porque é usado blocos de 32 *threads*, que coincide com o tamanho de cada *warp*, ou seja, cada bloco está naturalmente sincronizado.

- Problemas e descobertas com outras tentativas de resolução

  No ficheiro *fw_gpu2.cu* está uma resolução semelhante do problema, porém agora com o uso de *cooperative groups*, para substituir a abordagem de sincronização de blocos. A razão para não utilizar esta resolução é porque apresenta resultados menos eficientes comparativamente à chamada em *loop* do *Kernel,* e mesmo que esta última seja uma prática inferior, decidi optar por essa abordagem para conseguir um melhor desempenho e aplicar o conteúdo lecionado nas aulas. Em *fw_gpu3.cu* e *fw_gpu4.cu* existe uma tentativa de utilização de *shared memory,* ambas tentativas não funcionais, com o objetivo de tentar aceder menos vezes à memória global, que é consideravelmente mais custoso, e tentar obter resultados mais eficientes, porém tal resolução foi deixada de lado, pois não foi encontrada nenhuma maneira de utilizar *shared memory* eficientemente neste problema. Em ambas as tentativas foram utilizadas blocos de duas dimensões e ao contrário de *fw_gpu3.cu*, o algoritmo de *fw_gpu4.cu* está possivelmente incorreto em geral. Em *fw_gpu5.cu* foi utilizado *cooperative groups* e blocos de duas dimensões. Esta solução foi deixada de lado devido às justificações referidas anteriormente no relatório, no caso, devido à utilização de blocos de duas dimensões e utilização de *cooperative groups* para sincronização de blocos serem abordagens menos eficientes, além da complexidade possivelmente desnecessária. Em resumo, ao longo do meu percurso no trabalho, desisti de imensas das funcionalidades que o CUDA oferece, que em teoria aproveitam melhor o GPU, e com isto restou apenas a resolução mais simples e de mais fácil entendimento, que ainda consegue um *speed-up* bastante positivo em face à versão sequencial no CPU.

- (Tamanho de Grafo: **512**) Speed-up = Tsequencial / Tparalelo =
  = 0,5230 / 0,2606 ≈ 2,0069
  (Tamanho de Grafo: **1024**) Speed-up = Tsequencial / Tparalelo =
  = 4,1512 / 0,5614 ≈ 7,3944
  (Tamanho de Grafo: **2048**) Speed-up = Tsequencial / Tparalelo =
  = 33,2928 / 1,8455 ≈ 18,0399
  Conclui-se que o *speed-up* é exponencial em relação ao tamanho do grafo.

NOTA: O ficheiro com o código a ser avaliado é o *fw_gpu.cu*. Os cálculos do *speed-up* foram feitos com a média do resultado de cinco testes e mesmo que resolução no GPU seja inconsistente em termos de resultados, a média deve dar um resultado aproximado do que deve ser esperado. Testes feitos na máquina *machine-learning* fornecida aos alunos.