Projeto 4 PPC 2025

Pedro Lopes 58196

Notas Prévias:

À semelhança das outras fases do projeto, esta parte contou com assistência de um LLM (Claude Sonnet 4.5). Em concreto, foi usado para implementar o algoritmo de parsing (mais detalhes disponíveis no código fonte) e para duvidas gerais (por exemplo, necessidade de forward declarations, funcionamento dos eventos de Cuda ou questões sobre o pointer partilhado entre a GPU e CPU (mse))

Resumo do trabalho realizado:

Pergunta 1: How did you parallelize your program?

- Para paralelizar o algoritmo, foi necessário passar os dados e as funções para a CPU através de cudaMalloc e cudaMemcpy. Estas foram usadas porque, pelo que consegui pesquisar, eram mais eficientes que cudaMallocManaged. Esta ultima função foi usada apenas para o pointer dos resultados, uma vez que este tinha que ser acedido tanto pela GPU como pela CPU, logo beneficiava de um método que não necessitasse de copiar os conteúdos de um lado para o outro.
- Foram depois criados dois kernels: No primeiro (vamos chamar de K1), são calculados os valores de SE (erro ao quadrado) para cada conjunto de dados numa função. O segundo kernel, K2, calcula a media (MSE) de cada uma das funções.
- Para executar os kernels, foi necessário copiar todos os dados para a GPU de forma prévia (as funções precisam de todos os dados para serem testadas). De seguida, foram criadas 4 cudaStreams. Cada stream copia para a memoria as funções que vai executar (ou seja, um quarto das funções totais) e executa o K1 com um grid de blocos 2D de tamanho X por Y, com X a equivaler ao numero de pontos em data.csv a dividir por 256 (que é o numero de threads por bloco), e Y igual ao número de funções a serem analisadas naquela stream. Dessa forma, cada thread calcula o resultado da aplicação de uma função a uma única linha.
- O calculo da função é feito durante o parsing da string que representa a função, com o resultado a ser calculado ao mesmo tempo que o parsing é feito. Isto impediu uma possível otimização, onde o parsing de string para função aritmética era feito por uma só thread e guardada em memória partilhada dentro do bloco.
- Após isto, é executado K2, ou seja, o cálculo do MSE. Este é feito através de um grid com uma dimensão de tamanho X, onde X representa o numero de funções. Cada um destes X blocos possui 256 threads. No Kernel, é executado um algoritmo de Tree Based Reduction, que evita conflitos no acesso aos resultados ao mesmo tempo que paraleliza o calculo da soma de erros, que vai servir para calcular o MSE para cada função.
- O resto do programa funciona de maneira similar à versão sequencial, com a exceção de que é necessário fazer sincronização na CPU para esperar que a GPU termine o seu trabalho.

Pergunta 2: How many kernels and memory copies did you use?

Foram usados dois kernels, que foram chamados um total de 8 vezes (2 por cada stream). A memória dos dados foi copiada na sua totalidade 1 vez, antes da criação das streams, e as funções foram copiadas 4 vezes (uma vez para cada stream, com cada stream a copiar uma porção diferente). O pointer mse foi criado em memória unificada entre CPU e GPU, pelo que não foi preciso copiar manualmente, logo, não inclui na soma do numero de copias.

Pergunta 3: How did you minimize the number of kernels called?

Ao incluir todos os dados num só grid em vez de os dividir, foi possível diminuir o número de kernels. Foi também importante calcular várias funções por kernel, de modo a evitar chamadas repetitivas. O numero de chamadas poderia ser reduzido ainda mais se não implementasse as streams, mas isso prejudicaria a eficácia do código.

Pergunta 4: How did you minimize the number of data transfers required?

Os dados necessários a todas as streams (aka os features e targets em data.csv) foram todos copiados de uma só vez. O uso de memoria unificada no pointer mse retira a necessidade de cópia manual entre device e host, sendo que este é inicializado vazio, e só é lido pelo host assim que a GPU termina a sua execução. Para além disso, não há transferência de dados entre a execução dos kernels. Poderia também diminuir o numero de copias feitas ao remover a implementação das threads e passando as funções todas de uma vez, mas isso prejudicaria a performance do programa.

Pergunta 5: How did you choose the number of threads and blocks (for each kernel)?

- Todos os blocos foram inicializados com 256 threads por ser um expoente de 2,logo, um valor recomendado para hardware moderno.
- Em K1, a grelha possui 2 dimensões, com X a ter um número suficiente de blocos para cada thread ter uma linha de dados correspondente (ou seja, (linhas+256-1) : 256) e com Y a ser equivalente ao numero de funções, ou seja, cada thread aplica uma única linha de dados a uma única função.
- Em K2, a grelha possui apenas uma dimensão, com um numero total de blocos igual ao numero de funções. Cada bloco calcula a media de uma função, com cada thread a calcular a soma de 4 valores de cada vez, e reduzindo o numero de threads ativas para metade a cada passo.

Pergunta 6: How did you use shared_memory?

A anotação __shared__ só foi utilizada uma vez, no calculo do MSE. Essa variável guarda um array com as somas de vários resultados da função anteriores, somando estas somas a cada passo, até obter o resultado final em shared_sum[0]. Tendo em conta que todas as threads teriam que ler e escrever no mesmo array, fez sentido guardar este em memória partilhada para acelerar o acesso.

Pergunta 7: How did you take into account branch conflicts in your project?

O principal motivo de divergência no programa é o parser, que calcula o resultado de uma função ao mesmo tempo que a lê. Infelizmente, não consegui encontrar uma alternativa que resolvesse este problema.

No resto de K1, praticamente todas as threads realizam as mesmas operações. O mesmo para K2, que inclui o algoritmo de redução, onde a única divergência se encontra no passo final, onde a thread 0 calcula a média final para a função correspondente.

Pergunta 8: After which combination of number of functions and number of rows in the CSV does it make sense to use the GPU vs the CPU?

Através de um teste pouco exaustivo, realizado mantendo a proporção de linhas/funções em 100/1, chegamos à conclusão que a partir das 5000 linhas e 50 funções, o overhead necessário para executar na GPU passa a compensar quando comparado com a execução da CPU, com ambas as unidades de processamento a demorarem cerca de 18ms a executar. Há porém que lembrar que este teste foi executado num ambiente Colab específico para testes de GPU que utiliza uma Nvidia Tesla 4, sendo que não tenho garantia que a CPU usada possa ser considerada de nível/geração equivalente, pelo que há que ter cuidado na veracidade desta conclusão

Pergunta 9: If you were to use this program within a Genetic Programming loop (with evaluation, tournament, crossover, mutation, etc...) for several iterations, how would you adapt your code to minimize unnecessary overheads?

- Acredito que a melhor maneira de cumprir tal objetivo seria colocar dados os dados necessários na GPU no início e evitar ao máximo a transferência de dados entre a CPU e a GPU, provavelmente com a exceção da pontuação do melhor indivíduo.
- Assim que uma geração acabasse, a sua população teria que permanecer armazenada na GPU para ser processada na geração seguinte, evitando ser passada para a CPU a não ser que estritamente necessário. Tendo isto em conta, não haveria benefício em implementar um sistema de streams onde umas stream executa uma porção das tarefas enquanto a outra copia os dados, uma vez que o tempo necessário para copiar os dados é muito menor do que o tempo de execução de todas as gerações do algoritmo.
- Devido ao facto de não ser possível executar criar uma barreira de threads dentro de um grid inteiro, seria necessário que cada passo do algoritmo tivesse o seu próprio kernel.
- A exceção a esta ideia seria a sequência entre Tournament/Crossover e Mutation, que, como foi visto em fases passados do projeto, é a única parte do algoritmo que beneficia de Task Paralelism, pelo que esta sequência poderia ser executada dentro do mesmo kernel. Mais uma vez, não seriam necessárias múltiplas streams devido a todos os dados já estarem dentro da GPU.