Universidade de São Paulo
IME-USP

Relatório EP2
MAC0219

Tiago Martins Napoli - 9345384

1 Implementação

Na implementação de CUDA feita, a entrada com n matrizes 3×3 foi considerada como 9 vetores com n elementos cada, sendo que cada vetor carrega todos os elementos de determinada posição das matrizes. Daí basta aplicar a operação de redução de mínimo em cada um desses vetores e é possível montar a matriz resultante.

O problema foi, assim, reduzido a como fazer a redução de mínimo de um vetor. Para isso, cada bloco da GPU, com 32 threads cada um, ficou responsável por fazer a redução de um trecho de 64 posições seguidas no vetor. A posição inicial que cada bloco fica responsável é determinada pelo `Id.x` do bloco, de modo que a união das responsabilidades dos blocos resulte no vetor completo. Os resultados conseguidos para cada bloco são guardados e posteriormente transferidos para o host. Quando no host, os resultados dos blocos são fundidos e o resultado para o vetor é calculado.

Note que para isso é necessário que a quantidade de matrizes, ou seja, de posições em cada vetor, deve ser múltiplo de 64, mas, isso foi facilmente resolvido aumentando o vetor o quanto fosse necessário e colocando valores dummy nas posições adicionadas. (os valores dummy podem simplesmente ser algum valor que já havia aparecido no vetor original). Na implementação abaixo, a função `solve_for_position` recebe um vetor com todos os elementos na posição (x, y) de todas as matrizes que se quer resolver, e resolve a redução para esse vetor.

A implementação foi baseada na quarta implementação apresentada no seguinte documento NVIDIA - Optimizing Parallel Reduction in CUDA.

```
1  __device__ inline int min_cuda(int a, int b) {
2      if(a < b) {
3          return a;
4      }
5      return b;
6  }
7
8  __global__ void reduce(int *in, int *out) {
9      //vetor com sizeof(int) * BLOCK_SIZE
10     extern __shared__ int sdata[];
11
12     int tid = threadIdx.x;
13     int i = blockIdx.x * 2 * blockDim.x + threadIdx.x;
14     sdata[tid] = min_cuda(in[i], in[i + blockDim.x]);
15     __syncthreads();
16
17     //do reduction in shared memory
18     for(int s=blockDim.x/2;s > 0; s /= 2) {
```

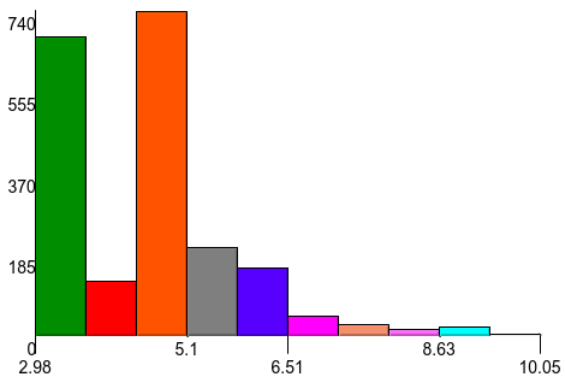
```
19          if(tid < s) {
20              sdata[tid] = min_cuda(sdata[tid], sdata[tid+s]);
21          }
22          __syncthreads();
23      }
24
25      if(tid == 0) {
26          out[blockIdx.x] = sdata[0];
27      }
28  }
29
30  int solve_for_position(int n, int blocks, int *h_x, int *h_block_min, int *d_x, int *d_bloc
31
32      checkCudaErrors(cudaMemcpy(d_x,h_x,n*sizeof(int),cudaMemcpyHostToDevice));
33      reduce<<<blocks,threads,threads * sizeof(int)>>>(d_x,d_block_min);
34      cudaDeviceSynchronize();
35
36      checkCudaErrors(cudaGetLastError());
37      checkCudaErrors(cudaMemcpy(h_block_min,d_block_min,blocks*sizeof(int),cudaMemcpyDevice
38
39      int res = h_block_min[0];
40      for(int i=0;i<blocks;i++) {
41          res = min_seq(res, h_block_min[i]);
42      }
43      return res;
44
45  }
```

2 Testes

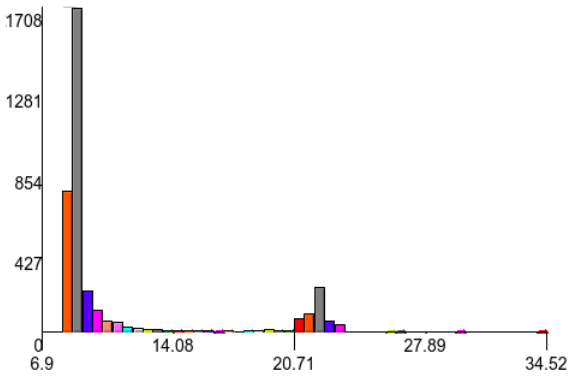
Os testes foram realizados na rede Linux, com a placa Tesla K20c. Para os testes foi feito um algoritmo sequencial, que servia como gabarito.

2.1 131072 matrizes

Os dados coletados foram os tempos de 2000 execuções dos algoritmos. Para cada execução foram geradas matrizes aleatórias, que foram usadas de entrada. Os tempos foram então coletados e foram criados os seguintes histogramas:



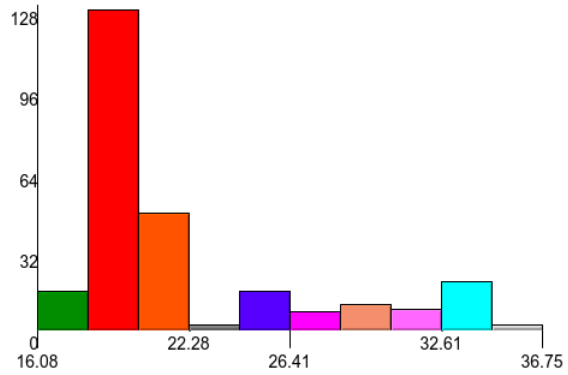
Tempos CUDA (ms) - 131072 matrizes



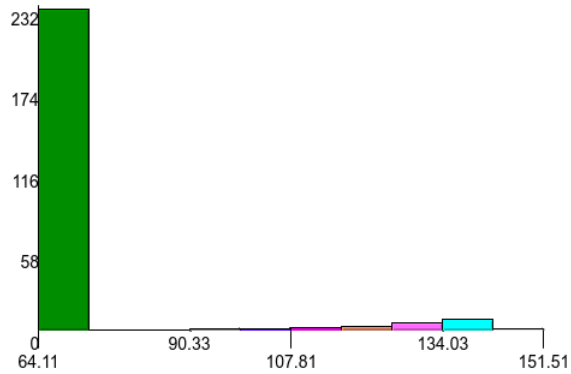
Tempos Sequencial (ms) - 131072 matrizes

2.2 1048576 matrizes

Foram coletados os tempos de 250 execuções e criados os seguintes histogramas:



Tempos CUDA (ms) - 1048576 matrizes



Tempos Sequencial (ms) - 1048576 matrizes

Os resultados mostraram que houve, de fato, melhora significativa. A melhora nas médias foi cerca de 3×.