Programming Massive-Parallel Processor - Exercise 1

TECHNISCHE UNIVERSITÄT DARMSTADT

Dominik Marino - 2468378
21. November 2020

Inhaltsverzeichnis

1 Cuda Devices

```cpp
void print_cuda_devices() {
  int nDevice;
  cudaGetDeviceCount(&nDevice);

  for (int i = 0; i < nDevice; i++) {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, i);

    std::cout << "Compute capability (Major/Minor): \t" << prop.major << ", " << prop.minor << std::endl;
    std::cout << "Multiprocessor count: \t\t\t" << prop.multiProcessorCount << std::endl; //return in int
    std::cout << "GPU clock rate: \t\t\t" << prop.clockRate << " kHz (" << prop.clockRate / 1000000.0 << "GHz)" << std::endl;
    std::cout << "Total global memory: \t\t\t" << prop.totalGlobalMem << " bytes ("<< prop.totalGlobalMem / 1024.0  << " MiB)"<< std::endl; //return in bytes
    std::cout << "L2 chache size: \t\t\t" << prop.l2CacheSize << " bytes ("<< prop.l2CacheSize / 1048576.0 << " KiB)" << std::endl;     //return in bytes
    std::cout << "\n" << std::endl;
  }
}
```

Listing 1: Print Cuda Devices (matmul.cc)

2 CPU Multiplication

2.1 Allocation Matrices

```cpp
  // TODO: Allocate CPU matrices (see matrix.cc)
  //       Matrix sizes:
  //       Input matrices:
  //       Matrix M: pmpp::M_WIDTH, pmpp::M_HEIGHT
  //       Matrix N: pmpp::N_WIDTH, pmpp::N_HEIGHT
  //       Output matrices:
  //       Matrix P: pmpp::P_WIDTH, pmpp::P_HEIGHT

  CPUMatrix inMatrixCPU1 = matrix_alloc_cpu(pmpp::M_HEIGHT, pmpp::M_WIDTH);
  CPUMatrix inMatrixCPU2 = matrix_alloc_cpu(pmpp::N_WIDTH, pmpp::N_WIDTH);

  CPUMatrix outMatrixCPU = matrix_alloc_cpu(pmpp::P_WIDTH, pmpp::P_HEIGHT);
```

Listing 2: Allocate CPU matrices (matmul.cc)

2.2 Fill Matrices

```cpp
  // TODO: Fill the CPU input matrices with the provided test values (pmpp::fill(CPUMatrix &m, CPUMatrix &n))
  pmpp::fill(inMatrixCPU1, inMatrixCPU2);
```

Listing 3: using fill method for CPU input matrices (matmul.cc)

2.3 Matrix CPU Multiplication

```cpp
  // TODO (Task 5): Start CPU timing here!
  timer_tp start_cpu = timer_now();

  // TODO: Run your implementation on the CPU (see mul_cpu.cc)
  matrix_mul_cpu(inMatrixCPU1, inMatrixCPU2, outMatrixCPU);

  // TODO (Task 5): Stop CPU timing here!
  timer_tp stop_cpu = timer_now();
  float cpu_comp_time = timer_elapsed(start_cpu, stop_cpu);
```

Listing 4: Start and Stop timer and run the implementation (matmul.cc)

```
void matrix_mul_cpu(const CPUMatrix &m, const CPUMatrix &n, CPUMatrix &p) {

  for (int i = 0; i < m.height; i++) {
    for (int j = 0; j < n.width; j++) {
      float sum = 0.0;
      for (int k = 0; k < n.height; k++) {
        sum = sum + (m.elements[i * m.width + k] * n.elements[k * n.width + j]);
      }
      p.elements[i * p.width + j] = sum;
    }
  }
}
```

Listing 5: Matrix multiplication (mul_cpu.cc)

2.4 Testing CPU

```
// TODO: Check your matrix for correctness (pmpp::test_cpu(const CPUMatrix &p))
  pmpp::test_cpu(outMatrixCPU);
```

Listing 6: Test Output Matrix (matmul.cc)

3 GPU Multiplication

3.1 Set CUDA Device

```
  // TODO: Set CUDA device
  int nDevice = 0;
  cudaGetDeviceCount(&nDevice);
  int userDeviceInput = nDevice - 1;

  if (userDeviceInput < nDevice) {
    cudaSetDevice(userDeviceInput);
  } else {
    printf("error: invalid device choosen\n");
  }
```

Listing 7: Set Cuda Device and check if the choosen device is invalid (matmul.cc)

3.2 Allocate GPU Memory

```
  // TODO: Allocate GPU matrices (see matrix.cc)
  GPUMatrix inMatrixGPU1 = matrix_alloc_gpu(pmpp::M_HEIGHT, pmpp::M_WIDTH);
  GPUMatrix inMatrixGPU2 = matrix_alloc_gpu(pmpp::N_WIDTH, pmpp::N_WIDTH);

  GPUMatrix outMatrixGPU = matrix_alloc_gpu(pmpp::P_WIDTH, pmpp::P_HEIGHT);
```

Listing 8: Allocate GPU Matrices (matmul.cc)

```
GPUMatrix matrix_alloc_gpu(int width, int height)
{
  GPUMatrix m;
  m.width = width;
  m.height = height;

  cudaMallocPitch((void**)&m.elements, &m.pitch, width * sizeof(float), m.height);

  return m;
}
```

Listing 9: Matrix alloc GPU (matmul.cc)

3.3 Upload 2 GPU

```
  // TODO: Upload the CPU input matrices to the GPU (see matrix.cc)
  matrix_upload(inMatrixCPU1, inMatrixGPU1);
  matrix_upload(inMatrixCPU2, inMatrixGPU2);
```

Listing 10: Upload Matrix 2 GPU (matmul.cc)

```
void matrix_upload(const CPUMatrix &src, GPUMatrix &dst)
{
  int size = src.height * src.width;
  cudaMemcpy(dst.elements, src.elements, size * sizeof(float), cudaMemcpyHostToDevice);
}
```

Listing 11: Upload Matrix (matrix.cc)

3.4 Matrix GPU Multiplication

```
// TODO (Task 5): Start GPU timing here!
  // TODO (Task 5): Start GPU timing here!
  cudaEvent_t evStart, evStop;
  cudaEventCreate(&evStart);
  cudaEventCreate(&evStop);
  cudaEventRecord(evStart, 0);

  // TODO: Run your implementation on the GPU (see mul_gpu.cu)
  matrix_mul_gpu(inMatrixGPU1, inMatrixGPU2, outMatrixGPU);

  // TODO (Task 5): Stop GPU timing here!
  cudaEventRecord(evStop, 0);
  cudaEventSynchronize(evStop);
  float elapsedTime_ms;

  cudaEventElapsedTime(&elapsedTime_ms, evStart, evStop);
  printf("CUDA processing took: %f ms\n", elapsedTime_ms);
  cudaEventDestroy(evStart);
  cudaEventDestroy(evStop);
```

Listing 12: Matrix Multiplication with timer for calculation (matmul.cc)

```
// TODO (Task 4): Implement matrix multiplication CUDA kernel
__global__ void multiplication_kernel(GPUMatrix m, GPUMatrix n, GPUMatrix out) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  float pValue = 0;

  for (int k = 0; k < m.width; k++) {
    float Melement = m.elements[ty * m.width + k];
    float Nelement = n.elements[k * n.width + tx];

    pValue += Melement * Nelement;
  }
  int out_index = ty * out.width + tx;
  out.elements[ty * out.width + tx] = pValue;
}

void matrix_mul_gpu(const GPUMatrix &m, const GPUMatrix &n, GPUMatrix &p)
{
  // TODO (Task 4): Determine execution configuration and call CUDA kernel
  dim3 grid(1, 1);    //div_up(m.height, n.width);
  dim3 dimBlock(m.width, n.height);

  multiplication_kernel<<<grid, dimBlock >>>(m, n, p);

  cudaError_t err = cudaGetLastError();

  if (err != cudaSuccess)
    printf("Error: %s\n", cudaGetErrorString(err));

  cudaDeviceSynchronize();
}
```

Listing 13: Kernel and Set up (mul_gpu.cu)

3.5 Download 2 CPU

```
// TODO: Download the GPU output matrix to the CPU (see matrix.cc)
CPUMatrix dOutputMatrixfromGPU = matrix_alloc_cpu(pmpp::P_WIDTH, pmpp::P_HEIGHT);
matrix_download(outMatrixGPU, dOutputMatrixfromGPU);
```

Listing 14: downlaod Matrix (matmul.cc)

```
void matrix_download(const GPUMatrix &src, CPUMatrix &dst)
{
  int size = src.height * src.width;
  cudaMemcpy(dst.elements, src.elements, size * sizeof(float), cudaMemcpyDeviceToHost);
}
```

Listing 15: Download Matrix from GPU (matrix.cc)

3.6 Testing GPU

```
// TODO: Check your downloaded matrix for correctness (pmpp::test_gpu(const CPUMatrix &p))
pmpp::test_gpu(dOutputMatrixfromGPU);
```

Listing 16: Testing GPU Matrix (matmul.cc)

3.7 Compare CPU Matrix with GPU Matrix

```
// TODO: Compare CPU result with GPU result (see matrix.cc)
matrix_compare_cpu(outMatrixCPU, outputMatrixGPU);
```

Listing 17: Compare CPU Matrix with GPU Matrix (matmul.cc)

```
void matrix_compare_cpu(const CPUMatrix &a, const CPUMatrix &b)
{
  int j = 0;
  for (int i = 0; i < a.height * a.width; i++) {
    if (a.elements[i] == b.elements[i]) {
      j++;
    }
  }
  if (j == (a.height * a.width)) {
    std::cout << "CPU Matrix and GPU Matrix are equal" << std::endl;
  } else {
    std::cout << "Matrices are not equal" << std::endl;
  }
}
```

Listing 18: Compare CPU Matrix with GPU Matrix (matmul.cc)

3.8 Print compution time from CPU and GPU

```
//print compution time from cpu and gpu
print_time(cpu_comp_time, elapsedTime_ms);
```

Listing 19: compution time from CPU and GPU (matmul.cc)

```
void print_time(float cpu_comp_time, float gpu_comp_time) {
  std::cout << "\n" << std::endl;
  std::cout << "Computing Time for CPU: " << cpu_comp_time << std::endl;
  std::cout << "Computing Time for GPU: " << gpu_comp_time << std::endl;

  if (cpu_comp_time < gpu_comp_time) {
    std::cout << "CPU is faster" << std::endl;

  } else if (gpu_comp_time < cpu_comp_time) {
    std::cout << "GPU is faster" << std::endl;

  } else {
    std::cout << "Computing time is equal" << std::endl;
  }
}
```

Listing 20: Compare computation time (timer.h)

4 utils

4.1 Print Matrix

```
void print_matrix(CPUMatrix &matrix) {
  int j = 0;
  for (int i = 0; i < matrix.height * matrix.width; ++i) {
    std::cout << matrix.elements[i] << ' ';
    j++;
    if (j == matrix.width) {
      std::cout << std::endl;
      j = 0;
    }
  }
}
```

Listing 21: Print Matrix in console (matmul.cc)

5 Question

- Where do the diffferences come from?
  - The CPU calculation is slower then the GPU calculation. If we look up to the CPU implementation, then we see that the algorithm is $O(n^3)$. In the case of matrix multiplication we can parallelize the computations, because the GPU have much more threads. In our task has every thread to calculate a position/element in the output matrix P.
  - But it really depends on the size of the matrix, but this does not mean that paralleling of a matrix multiplication is always faster then the CPU implementation. It really depends on the size of the data. Small data with a few calculation fits poor on a GPU. Another reason can be memory access for slowness i.e. if we are copying a lot of memory in our algorithm. In my code i want to synchronise all the threads and if we have to wait until all threads are done, then it also can slow down the computation time.