TÉCNICO LISBOA

Assignment 4

- Tutorial -

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2024/2025

Overview

In this assignment, you will write and analyze some parallel programs in CUDA, the main difference being that we will now introduce and focus on Tensor Cores and their applications. You will need to develop some simple CUDA codes running on regular GPU cores (some using cuBLAS), but you will also learn about how Tensor Cores can make GPUs more efficient and how to extract their potential. In the first part, we will focus on Matrix Multiplication (once again), and then we will switch our focus to efficient implementations of convolution.

Deadlines and Submission

NO SUBMISSIONS! This work serves as a tutorial and preparation – **NO EVALUATION**. It is intended for your practice and learning via examples and tutorials. This assignment provides a set of exercises for you to practice! You can resolve them at our lab session or at home. Surely, you can discuss your solutions and doubts with the teacher of your labs in our next lab session. Again, there is no submission for this part of the work and your solutions will not be graded!

You can consider these examples as a representative set of problems to get you prepped for the work to be developed in the class (which will be evaluated). Those assignments are not yet provided to you!

Environment Setup

For this assignment, you will need to run (the final version of) your codes on LSD2 machines, which contain NVIDIA RTX 4070 Ti GPUs. The assignment starter code is available on the course webpage (section: Labs).

Part 1: Warm up! Going back to Matrix Multiplication

The starter code for this part of the assignment is located in the `/gemm` directory of the assignment zip file. In the `main.cpp` file you can find a simple implementation of General Matrix Multiplication (GEMM) in the CPU (see function `gemm_cpu`) and several declarations of the GPU `cublas_gemm_*` functions that use CUBLAS and different floating-point data precisions. We will perform GEMM in FP32 in CUDA cores and FP16 and TF32 in Tensor Cores. The Tensor Core precisions will take FP32 inputs and outputs and work with their respective precision during computation. To compare the results of the mat mul in the CPU to the ones obtained in the GPU (with different precisions), we will use a function to calculate the Euclidean distance between matrices (see `euclidean_distance` function). Here, you will also be able to perform multiple calls to GPU GEMM that you will need to complete in the `gemm.cu` file. It is important to note that to ensure correct performance measurements from the GPU, we have to launch the same kernel twice: the first as a warmup, and the second to get the timings.

As a reminder, we are dealing with the operation C=AxB, where A is an M by K matrix, B is a K by N matrix, and finally C is an M by N matrix of single floating-point precision. This operation is formally defined as `D=alpha*A*B+beta*C`, and to perform the operation above we have to define our scale factors `alpha` and `beta` to 1 and 0, respectively.

We can now focus on the `gemm.cu` file. For your first assignment, **you will need to complete the call to the `cublas_gemm` function** provided. This is the general function that implements the matrix

multiplication which will be wrapped by the wrapper functions to perform the GEMM operation in the data precisions we want. For this purpose, you will need to complete the gaps described in the comments of the starter code. We will be using the `cublasGemmEx()` function to perform our matrix multiplication, and you can find all the information you need about it in the slides from our theoretical lectures and the cuBLAS documentation. Remember that the default way of accessing matrices in cuBLAS is column-major, and the CPU implementation of the matrix multiplication already has taken this into consideration.

After completing this function, hit `make` to compile the program and run it for a problem size of 128x128x128 (M by N by K) with the command:

```
./gemm
```

The Euclidean distance between the two results (CPU and FP32) should be very small (around 0.02). Now, **complete the wrapper functions** `cublas_gemm_fp16` and `cublas_gemm_tf32` by using the `cublas_gemm_fp32` as an example and choosing the correct cuBLAS **compute type** so that the functions will execute the GEMM in the tensor cores using FP16 and TF32 multiplicands, respectively. We want to produce our result in FP32, so you should look at the compute types which will automatically downconvert our FP32

Now hit `make` once again and run the code for different problem sizes using, for example:

```
./gemm -a 512
```

where you can replace 512 for any value you want to set the MNK dimensions to. Execute the code for different problem sizes.

The program will output the time it takes to run the cuBLAS kernel, the resulting performance in GFLOPS/s and the total time it takes to execute the whole function with memory transfers and kernel calls. Furthermore, it also shows the Euclidean distance between the CPU output and the different outputs of the GPU implementations.

Fill the table below with the problem sizes and the **GFLOPS/s** these implementations achieved:

Sizes (N)	128	256	512	1024	2048
FP32 [GFLOPS]					
FP16 [GFLOPS]					
TF32 [GFLOPS]					

How do you explain the obtained results?

Some questions you should think about: For small matrices, why is the performance nearly identical? What happens to performance as the problem size increases? What is responsible for the difference in performances? Why can we observe it as the problem size increases? What can you observe when you compare the Euclidean distance between the output matrices of the different implementations? What happens to these differences as the problem size increases? Why? Do all implementations have the

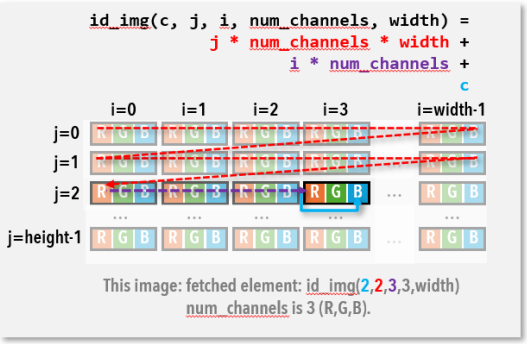
same compute roof? Feel free to try bigger problem sizes (optional!) (4096, 8192) to explore this question, but keep in mind that the CPU implementation will take a long time to execute!

Part 2: GPU Convolution with CUDA

The starter code for this assignment can be found in the `conv/` directory of the assignment zip file. Here, you will have to implement your own version of an image convolution using CUDA cores.

In the `main.cpp` file you can find some code responsible for the preprocessing of the `input.png` image, reading it to an array of floats with the dimensions `input_h*input_w*num_channels`. Here you can consider that the data is arranged in a row major grid, where each element contains the values of the 3 RGB channels. You are encouraged to access the image through the `id_img()` macro. In this file, all of the filters are also populated, and all the necessary arrays are already handled for you. You don't need to alter anything in this file, but you are encouraged to have a look at it. There is a CPU implementation of the convolution code we discussed in the class (see `convolve_cpu`), which will work as our reference implementation. Finally, you can also find the `reduce_channels` function that sums all the activations in the output channels into one so that we can produce our output into a black and white image.

There is also an additional macro defined to help you access the 4D filter tensors without any issues. This will help you ensure correct strides between the 4 dimensions, and you can use it with the `id_filter()` macro. Together with the `id_img()` macro, these allow you to easily access the tensors, without worrying about how the data is organised within them. You can have a look at the diagrams below, which tell you how the data is organised and how you can access each element.



In `convolution.cu`, you can find a host function `convolution_gpu_cuda_cores` and a device function `convolution_kernel`. Here, you will have to implement the convolution kernel to perform the same convolution operation with the use of the CUDA cores, for a variable number of filters and filter dimensions. The function `convolution_gpu_cuda_cores` is almost entirely defined, all the memory allocations and transfers are once again handled for you and you just have to set a value for your `threadsPerBlock` and `numBlocks` according to the needs of your convolution implementation. In the **`convolution_kernel`** function you have the main task of this part: you must **complete this function**. Take inspiration from the CPU implementation and think about how you can cleverly parallelize this operation. Remember the very useful macros at your use.

Now, hit `make` (you can ignore the warnings that appear for `main.cpp`) and run it for the default parameters with the command:

```
./conv
```

You can now compare the outputs of the CPU implementation (`output.png`) and your GPU implementation (`output_gpu.png`), and you should see the same resulting image in both. By executing the previous command, you are performing a convolution on the input image with 4 edge detection filters which are capable of highlighting all edges of the original shapes. The edge detection filters only work with 4 filters and a filter dimension of 3. Alternatively, you can also run the convolution for random filters which independently of the filters will always result in the same output, as they are accumulated into a single channel. You can do so with:

```
./conv -r
```

This will execute the convolution with the default parameters (4 filters of `3x3`) but with random filters. When executing with random filters you can also alter the number of filters and the dimensions of the filters, but you cannot change the number of input channels as the image will always be RGB. You can alter these two parameters with, for example:

```
./conv -r -n 8 -f 5
```

This will execute a convolution with 8 filters of `5x5`. It is important that your kernel is correctly working for the next part of this assignment.

Compare the results and performance obtained for the CPU and GPU implementations for the default configuration of parameters.

Here are some questions you can think about: Why do we have to reduce the different output channels activations to a single channel? How can you explain the output of the random filters? (hint: black pixels have all channels set to 0). How can you explain the difference in the kernel time and the GPU time (time taken for the whole function) for the GPU implementation of convolution? How could you further optimize your GPU implementation? How did you parallelize your workload, and what alternative ways could you have partitioned it?

Part 3: GPU Convolution with cuDNN

The starter code for the final part of the assignment can be found in the `./cudnn` directory of the zip file. Here, you can find an implementation of the convolution with the use of the cuDNN library in the `main.cu` file.

In the main.cu file you can find all the code relevant to this task and while you won't need to alter any code for this part, you are still encouraged to have a look at the code to understand how you can work with cuDNN. You can find the `build_graph` function that performs all the operations necessary to declare a cuDNN graph; its tensors and operations are described, and all the necessary steps are taken to build it to a single fused computation node.

The rest of the file is very similar to the code we worked with in the previous part, as it loads the input image and declares and populates different filters according to the parameters we pass as command line arguments. There is some additional pointer manipulation to allow us to use different datatypes for our convolution, such as half precision for the input and filter. Here, we measure the graph execution time, which gives us our performance, but also the total time taken for all the operations in the GPU and the

build time, which is the time the runtime compiler takes to transform the graph into a single optimized computation node. Finally, we reduce all the channel activations and then produce the output image, similarly to the previous task.

We are using cuDNN to explore the performance of the convolution kernels when using the "implicit" GEMM algorithm we discussed in class. By changing our computation mode, we will also be able to use the Tensor Cores to perform the convolution. We can now compare the performance of our previous implementation and the cuDNN implementation for different configurations of parameters and execution types (CUDA cores vs Tensor Cores).

To do so, hit `make` (you can ignore the compiler warnings for the `stb_image.h` file) and execute the application with the default parameters (edge detection) with:

```
./cudnn-conv
```

Or for a different number of filters and dimensions of the filters with:

```
./cudnn-conv -r -n 8 -f 5
```

Remember that you can only use different parameters for random filters. To use different modes of computation, you should use the following command where you can choose `FP32`, `FP16_32` or `TF32` as the mode for, respectively, computing in CUDA cores, or Tensor Cores with `FP16` multiplicands and `FP32` accumulation and `TF32` multiplicands.

```
./cudnn-conv -m FP32
```

Now, we are going to compare the performance between implementations for different configurations. Fill the tables below with the performances these implementations achieved.

With the default size of the filter fill:

Number of Filters	4	32	64	128	256
Your Part 2 implementation [GFLOPS]					
cuDNN FP32 [GFLOPS]					
cuDNN FP16_32 [GFLOPS]					
cuDNN TF32 [GFLOPS]					

Now, using 32 filters (-n 32), fill the table:

Size of filter	3	5	7	9	11
Your Part 2 implementation [GFLOPS]					
cuDNN FP32 [GFLOPS]					
cuDNN FP16_32 [GFLOPS]					
cuDNN TF32 [GFLOPS]					

Looking at the results from the two tables above, what can you observe? Are the cuDNN implementations significantly faster than your previous implementation? When are the differences accentuated? How does the performance change as we increase the number of filters and the size of them? Why does this happen? Why is there such a discrepancy between the GPU time and the kernel execution time in the cuDNN implementation? What is the build time and why does it take so long? Do you notice differences when using CUDA cores vs Tensor Cores in cuDNN? Is the performance difference what you expected? How can we better utilize the Tensor Core and achieve better performance?