# Lab: Getting Started Guide

## 1. Introduction

This document describes information for GPU programming for the course. Please, read it carefully.

## 2. MinoTauro GPU cluster

### 2.1. Change your password

Use an SSH client program to login to **dt01.bsc.es** using the training account login information provided to you. **This machine is only used for password setup**, the machines you will have to connect during the hands on labs of the course are detailed below. Once logged in, follow the steps that appear in the terminal to change your default password for a new one of your choice (If no steps appear after you login, execute the command *passwd).* Note that there could be some restrictions to ensure that passwords have a minimum strength level (a minimum number of characters, use of numbers and/or punctuation symbols, etc.). Once you finish changing you password, you can logout from this machine and connect to the cluster login node.

**Note:** the password change may take ‘a couple’ of minutes to be effective.

### 2.2. Cluster usage

With your SSH client login to **mt1.bsc.es** using the provided username and your new password. If you just changed your password, it may take some time (up to 10 minutes) for the change to be effective, thus it is possible that you cannot login during this time.

In this section, we present a brief description of the main commands needed to work in the cluster. For further information about the MinoTauro cluster, take a look to the **MinoTauro User's Guide**. This guide contains detailed information about the MinoTauro Cluster, its job management system, etc.

The MinoTauro cluster uses a job management system to control access to the computing nodes. To run the labs, you will need to use the commands provided by the system to submit jobs for execution, check their state, or cancel them if necessary:

#### Submitting a job

To submit a job, use the command *sbatch <job_script>*, like in the following example:

```
$ sbatch ./job.sh
Submitted batch job 170798
```

After you execute the command, you shall see a message with the ID of the newly created job (170798 in the example above). The argument of *sbatch* (<job_script>) is the path of a shell script which contains a set of directives required by the job queue system, and the command to execute the program (see the example below). Some labs already provide a job.sh file, just note that to run the different programs in a lab, you will have to edit the last line of the script to execute the corresponding program. If a lab doesn’t provide a job script, make a copy of a previous one and edit it accordingly for the current lab.

```
#!/bin/bash
```

```
#SBATCH --job-name=MyJob
#SBATCH --workdir=.
#SBATCH --output=%j.out
#SBATCH --error=%j.err
#SBATCH --time=00:05:00
#SBATCH --ntasks=1
#SBATCH --tasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:1

<some command(s)>
```

### Query job status

To query job status use the *squeue* command. You shall see all your submitted jobs that are waiting or running in the cluster.

A job has the PENDING state when is waiting in the queue:

```
$ squeue --long
JOBID  NAME  USER     STATE   TIME TIMELIMIT CPUS NODES NODELIST(REASON)
170798 MyJob nct00002 PENDING 0:00 10:00     6    1     (Priority)
```

A job has the RUNNING state when it is executing. When a job is in the RUNNING state, the TIME column shows the elapsed time since the job started executing, and the NODELIST (REASON) column indicates on which cluster's node/s the job is running.

```
$ squeue --long
JOBID  NAME  USER     STATE   TIME TIMELIMIT CPUS NODES NODELIST(REASON)
170798 MyJob nct00002 RUNNING 0:04 10:00     6    1     nvb118
```

When the job has finished and the job queue system has finished cleaning the job resources, the job won't appear anymore in the list:

```
$ squeue --long
JOBID NAME USER STATE TIME TIMELIMIT CPUS NODES NODELIST(REASON)
```

### Canceling a job

To delete a job you can use the command *scancel <job ID>*

```
$ scancel 170798
```

When your job is finished, you can check the program output in the files “XXX.out” and “XXX.err”, where “XXX” is the job ID number. Files with “.out” extension store the execution output log (stdout). Files with “.err” extension store the error or warning messages printed during the execution (stderr).

**Note: Be aware** that interruptions to your SSH connection may cause you to lose unsaved work. If you have reason to believe your connection to the cluster may be unstable, you may wish to use an FTP program with the SFTP protocol and your provided login information to retrieve the source files to your local machine for editing and upload them back to the cluster to compile and execute the programs.

## 3. Parboil Framework & Benchmarks

3.1. Obtain and setup the labs framework

Use an SSH client program to login to **mt1.bsc.es** using the training account login information provided to you. Your home directory can be organized in any way you like. To unpack the labs framework including the code for all the lab assignments, execute the following command in the directory you would like the labs framework to be extracted:

```
$ tar -zxf ~nct00021/mv19_cuda.tgz
```

Check that the package directory has been created and populated.

```
$ cd mv19_cuda; ls
```

There should be several directories and files, including:
- `benchmarks`
- `datasets`

### 3.2. Parboil basic usage

The course labs use the Parboil Benchmarks Suite framework as its base. This framework contains a script named *parboil* which is used to do all common tasks like compilation, clean up, or execution of the labs.

Here is a list with the basic parboil's commands. For further information execute ./parboil --help

To start, you can list all available programs:

```
> ./parboil list
```

To get more information about a given program, like the available program's versions and datasets:

```
> ./parboil describe <program>
```

To compile a program:

```
> ./parboil compile <program> <version>
```

To run a program:

```
> ./parboil run <program> <version> <dataset>
```

**NOTE: Do not run the programs directly on the login nodes of the cluster. Use the job queue system to run them on the computing nodes.**

To run a program under CUDA memcheck (to debug errors when a kernel execution fails):

```
> ./parboil memcheck <program> <version> <dataset>
```

**NOTE: Do not run the programs directly on the login nodes of the cluster. Use the job queue system to run them on the computing nodes.**

To clean a program's temporary files:

```
> ./parboil clean <program> [<version>]
```

You will noticed that the *run* command also executes the *compile* command in advance. This will reduce the number of typing for testing and decrease the turnaround time. However, the *compile* command is still meaningful when you are writing significant code and expecting lots of errors, as it would not bother you from trying to run the program.
In general, when a parboil benchmark is run, the script will also read the output of the benchmark and compare it against a known correct output, to then print a message saying whether the comparison succeeded or not.

### 3.3. Understanding the Parboil framework

The Parboil framework is developed by the IMPACT Research group at the University of Illinois, and is meant to provide an easier interface to manipulate benchmarks and measure performance on a GPU platform. Since many of the assignments and code examples are benchmarks used by the group, we will be using the parboil infrastructure in this course to help make some of the details of standardizing compilation and execution easier.

**Primary Parboil infrastructure components**

- *parboil*: This is the main script to compile and run the labs.
- Directory *benchmarks*. This directory stores each benchmark. Each subdirectory of the benchmarks directory contains all the files for a single benchmark. A benchmark directory contains the following directories:
    - *build*: Stores compiled executables for the benchmark. Is created during compilation, and may not be present initially.
    - *run*: Stores the results of most recent run of each version of the benchmark. Is created when needed, and may not be present initially.
    - *src*: Stores the source files of the benchmark. Each subdirectory of src is a different version of the benchmark: possibly with different optimizations, or separate CPU and CUDA versions, etc.
    - *tools*: Stores custom scripts to perform the output comparison for the benchmark.
- Directory *datasets*. This directory stores each benchmark's datasets. A benchmark dataset directory is composed of the directories input, and output.
    - *input*: Stores a set of parameters and input data sets. Each subdirectory of the input folder represents a different input.
    - *output*: Stores the “golden” output data sets for comparison.
- Directory *common*: This directory contains libraries shared by all labs.
    - *src*: This directory contains the main parboil library source files.
    - *include*: Common include files.
- Directory *driver*: Python scripts to control the compilation and execution of the labs.

The output log of each parboil benchmark should include a report printed by the parboil timer library, which lists the execution time taken for each of the following sections: IO, GPU, Copy, Compute, and others. Their meanings are listed below.

- **IO**: (CPU) Time spent in input/output.
- **Kernel**: (GPU) Time spent computing on the GPU, recorded asynchronously from the CPU.
- **Copy**: (CPU/GPU) Time spent synchronously moving data to/from GPU and allocating/freeing memory on the GPU.
- **Driver**: (CPU) Time spent in the host interacting with the driver, primarily for recording the time spent queueing asynchronous operations.
- **Copy_Async**: (GPU) Time spent in asynchronous transfers
- **Compute**: (CPU) Time for all program execution on the host CPU other than parsing command line arguments, and all other operations otherwise accounted for above.
- **CPU/GPU Overlap**: Time double-counted in asynchronous GPU and CPU activity: automatically filled in, not intended for direct usage. Total execution time should be the sum of all previous timers minus this overlap time.

# Lab: Vector Addition

In this introductory hands-on lab, you will perform a simple vector addition that will help you to understand the structure of a CUDA program. You will complete key portions of the program to compute this widely-applicable kernel. In this lab you will learn:

- How to allocate memory in the GPU
- How to copy data between host and GPU memories
- How to use the thread and block indexes to write CUDA kernels
- How to invoke GPU kernels

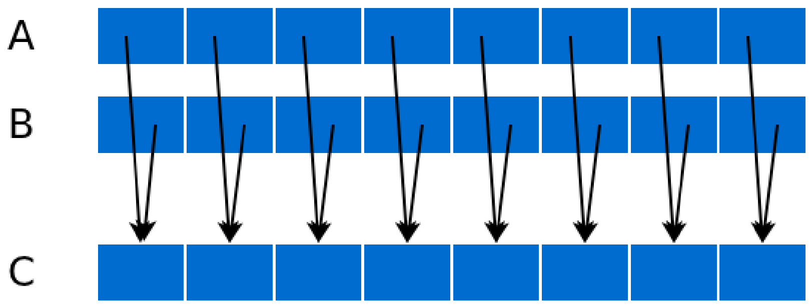
The directory for this lab is:

*`benchmarks/vecAdd/src/cuda`*

**NOTE: In its initial state, the code does not compile. You must fill in the blanks or lines to make it compile.**

## Background

In this implementation each thread will compute a single element of the output vector, as shown in the picture;



## Instructions

Step 1: GPU memory allocation
Allocate memory for each of the input/output vectors. Use the variables named `A_d`, `B_d` and `C_d`. The number of elements is the same for all the vectors and is stored in the variable `vec_size`.

*File: `main.cu`*

Step 2: Copy input vectors to GPU memory
Copy the input vectors from host memory (`A_h`, `B_h`) to GPU memory (`A_d`, `B_d`).

*File: `main.cu`*

Step 3: Initialize thread block and kernel grid dimensions
Define the grid dimensions for the kernel. You can assume a size of 512 threads for the thread block. Remember that you should use a thread per computed element. Take into account that the number of elements of the vectors might not be divisible by 512.

*File: `main.cu`*

Step 4: Invoke CUDA kernel
Launch the CUDA kernel using the computed grid and block dimensions. You have to pass the allocations in the GPU memory and the number of elements of the vectors. You can see the signature of the kernel function in the `kernel.cu` file.

*File: `main.cu`*

NOTE: the `__restrict__` attribute tells the compiler that the memory pointed to by the pointer is not referenced by any other pointer argument. This may allow the compiler to better optimize the code. However, is the programmers responsibility to make sure no other pointer argument points to the same memory, otherwise, the results may be wrong.

Step 5: Implement vector addition kernel
Use the block and thread identifiers to compute the element to be computed by each thread. Take into account that the number of elements of the vectors might not be divisible by the thread block size. Read the input elements to be added. Write the sum of the two input values into the output vector.

*File:* `kernel.cu`

Step 6: Copy output vector to host memory
Copy the input vectors from GPU memory (`C_d`) to host memory (`C_h`).

*File:* `main.cu`

Step 7: Free GPU memory allocations
Free the vectors allocated in GPU memory (`A_d`, `B_d`, `C_d`).

*File:* `main.cu`

Step 8: Compile and run the program
If you are using a cluster, modify the job script to execute this lab, and submit it with the corresponding command. The program must give correct results with all the provided datasets.

# Lab: Simple Matrix-Matrix Multiplication

Your goal for this lab is to get used to the CUDA kernel language. For that, you are required to implement a simple dense matrix multiplication in CUDA. In this lab you will learn:

- How to create and work with 2D grids.

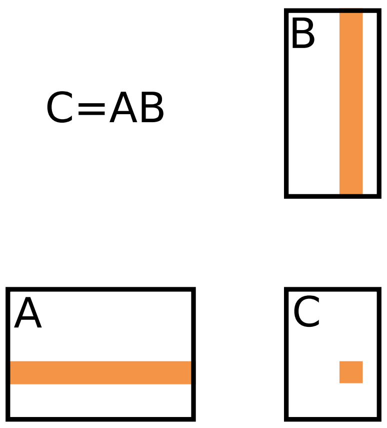
The directory for this lab is:

*benchmarks/sgemm/src/cuda_noblas*

## Background

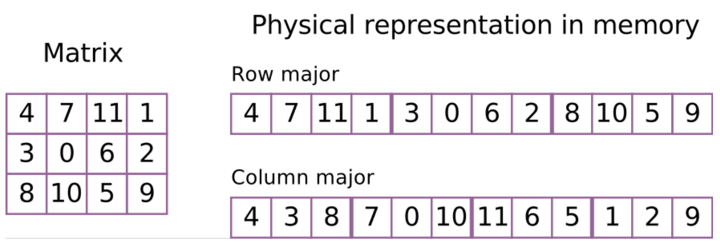
The routine you have to implement expects A and C matrices to be in Column Major layout[1] and B to be in Row Major layout[2]. Each element of the Matrix-Matrix multiplication result (matrix C) is computed with the dot product of a row of the A matrix and a column of the B matrix:

```
for (unsigned i = 0; i < ROWS_C; ++i) { // Row
  for (unsigned j = 0; j < COLUMNS_C; ++j) { // Column
    float sum = 0.0f;
    // Traverse row 'i' of A and column 'j' of B
    // Remember that A is in Column Major and
    // B is in Row Major layout
    for (unsigned c = 0; c < COLUMNS_A; ++c) {
      sum += A[<linear idx for A>] * B[<linear idx for B>];
    }
    C[<linear idx for C>] = sum;
  }
}
```



**[1] Column Major layout: elements in the same column are placed into contiguous memory locations.**
**[2] Row Major layout: elements in the same row are placed into contiguous memory locations**

Since the kernel receives the matrices as pointers, we cannot access them using the row and column indexes directly (e.g. A[row][col] // won't work). We need to compute the linear index, using the corresponding row and column indexes, to make the access into the “flattened” matrix. The formula used to compute the linear index depends on which representation is used to flatten the matrix, as depicted in the following figure:



## Instructions

Step 1: Invoke the kernel
Create a 2D grid with as many threads in the `x` dimension as rows in C, and as many threads in the y dimension as columns in C. Add the code to invoke the `sgemmNT_naive` kernel.

*File: `sgemm_kernel.cu`*

Step 2: Implement the kernel
Complete the `sgemmNT_naive` kernel in which each thread computes a single element of the output `C` matrix.

*File: `sgemm_kernel.cu`*

Step 3: Run the program
If you are using a cluster, modify the job script to execute this lab, and submit it with the corresponding command. The program must give correct results for the datasets ***data0***, ***data1***, ***data8*** and ***data9***.

# Lab: Tiled 7-point 3D Stencil

Tiling is a common technique used to improve data locality and minimize accesses to main memory. In this lab you will extend a naive implementation of the 7-point 3D stencil kernel to use tiling to reduce accesses to global memory. In this lab you will learn:

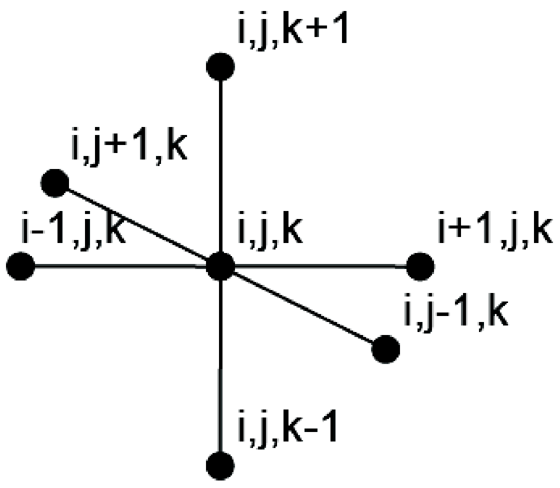
- How to use shared memory to store values that are read by many threads within a thread block. (Shared memory tiling)
- How to use registers to store values that are used many times by the same thread. (Register tiling)
- Why the size of thread blocks is important when using shared memory.

The directory for this lab is:

*`benchmarks/stencil/src/cuda`*

## Background

Stencil computations are widely used in scientific simulations. In a *k*-order stencil computation, the value of each cell of the output array is computed as a linear combination of the central value and the values of the *k* neighbouring cells of the input volume in each dimension. The figure below shows the 7 values of the 3D input array used to compute the output value for the *(i, j, k)* cell in a 1-order stencil computation (also called 7-point stencil).



The code you will be writing will iterate through the z dimension of 3D volume, applying the stencil above to the elements of one plane (X-Y) at a time. The naive implementation is provided in the code as the `simple2D_stencil`. Your task will be to optimize it using shared memory tiling and register tiling techniques. The idea behind these techniques is to temporarily store a part of the input set that is reused for several operations in a memory local to the SM (either registers or shared memory). Since these local memories have shorter access times, subsequent loads of the data are faster, improving the overall kernel performance.

## Instructions

Step 1: Initialize grid and block dimensions

Create a 2D grid with as many threads as elements in one Z plane. Block size should correspond to the tile size (given in the code).

*File: `kernels.cu`*

Step 2: Write the shared memory tiled kernel

Complete the `block2D_stencil` kernel in which each thread computes a single element of the output volume. To simplify boundary conditions, the outer boundary of each X-Y plane (topmost and bottommost rows, and leftmost and rightmost columns) is initialized to zeros, and the thread blocks process the elements within this boundary. Thus, the thread blocks that process any of the outer boundaries of the X-Y planes will have a fraction of threads that are idle. For your convenience, `Index3D(i, j, k)` macro is provided to help you accessing the 3D array.

**NOTE**: remember to change the host code to launch your implementation of `block2D_stencil` instead of the already provided `simple2D_stencil`.

*File:* `kernels.cu`

Step 3: Extend the kernel to perform register tiling over the Z dimension

Extend the shared memory tiled kernel to perform register tiling. The idea here is to reuse the already loaded values that are also used in the next iteration (x-y plane). Keep them in local variables (registers).

*File:* `kernels.cu`

# Lab: Tiled Matrix-Matrix Multiplication

Tiling is a common technique used to improve data locality and minimize accesses to main memory. In this lab you will extend a naive implementation of the Matrix-Matrix multiplication kernel to use tiling to reduce accesses to global memory. In this lab you will learn:

- How to use shared memory to store values that are read by many threads within a thread block. (Shared memory tiling)
- How to use registers to store values that are used many times by the same thread. (Register tiling)
- Why the size of thread blocks is important when using shared memory.

The directory for this lab is:

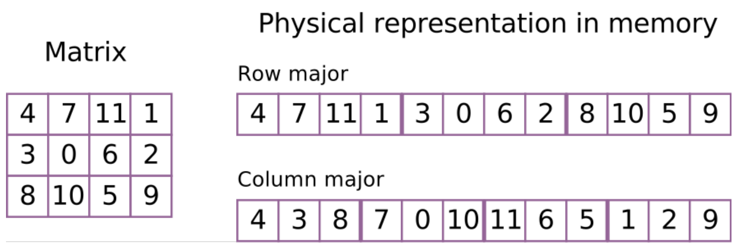
*benchmarks/sgemm/src/cuda_noblas*

## Background

The routine you have to implement expects `A` and `C` matrices to be in Column Major layout[1] and B to be in Row Major layout[2]. In this case, the kernel implementation have to apply two common optimization techniques: shared memory tiling and register tiling. Basically, both techniques temporarily store a part of the input set that is reused for several operations in a memory local to the SM (either registers or shared memory). Since these local memories have shorter access times, subsequent loads of the data are much faster, improving the overall kernel performance.

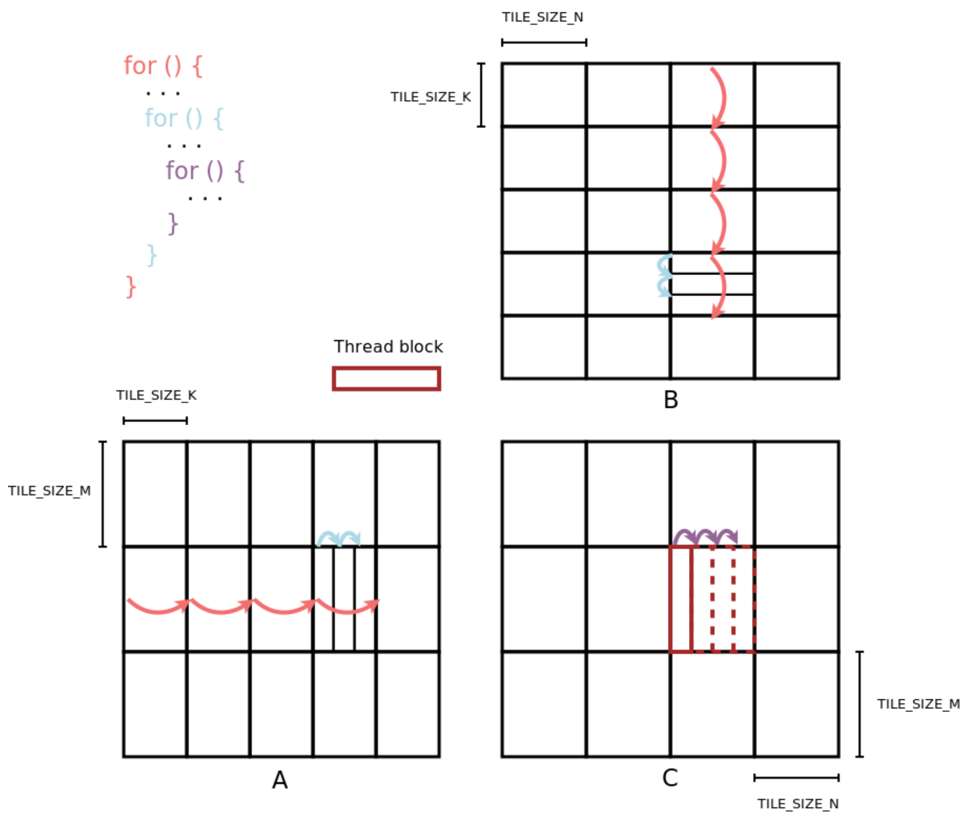
**[1] Column Major layout: elements in the same column are placed into contiguous memory locations.**

**[2] Row Major layout: elements in the same row are placed into contiguous memory locations.**



**Volkov scheme**

In this lab you will use the Volkov scheme to implement a highly efficient matrix multiplication. The scheme of the algorithm performed by one thread block to compute its output matrix tile is shown in the next figure (shared memory and register used for the tiling are not pictured). Note that in this case the thread block is smaller than matrix C's tile, and the whole tile is computed with a kernel that has three nested loops.



The tile dimensions are defined by the TILE_SZ_x constants. Matrix C tile dimensions are `TILE_SZ_M` rows per `TILE_SZ_N` columns. `TILE_SZ_K` is the number of columns of A's tiles, and the number of rows of B's tiles. The expected sgemm implementations must be able to handle matices of any size, even if they don't evenly divide by the tile sizes.

## Instructions

Step 1: Invoke the kernel
Comment the *#define SIMPLE* line (located at the top of the file) to select the tiled kernel version. Create a 2D grid with as many thread blocks as tiles has matrix C. Take into account that the matrix dimensions may not evenly divide by the tile dimensions. In this version of sgemm, each thread block has as many threads as rows has a C tile, all in a single dimension. During the sgemm computation, all the threads in a block iterate over the corresponding C tile columns, with each thread computing one row of the tile.

*File: `sgemm_kernel.cu`*

Step 2: Implement the kernel
Complete the `sgemmNT_tiled` kernel. The body of the kernel already contains a skeleton of the expected code, with some comments explaining the computation steps. Since the matrices dimensions may not evenly divide by the tile dimensions, the last row an column of tiles may have less columns and rows, respectively. The kernel code must check this to implement a correct sgemm and to avoid accesses to invalid memory addresses.

*File: `sgemm_kernel.cu`*

Step 3: Run the program
Ensure that the *#define SIMPLE* line in *sgemm_kernel.cu* is commented. If you are using a cluster, modify the job script to execute this lab, and submit it using the corresponding command. The program must give correct results for all the datasets (*data[0-9]*).

# Lab: Histogramming

Your goal for this lab is to learn how to implement efficient histogram using CUDA. In this lab you will learn:

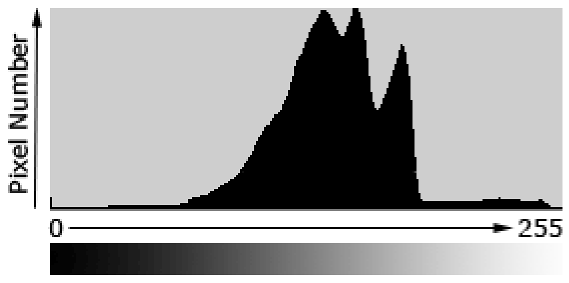
- How to use atomic operations in CUDA
- How to compile CUDA kernels for a specific GPU architecture (compute capability)
- How to use the privatization technique to minimize the access to the shared data

The directory for this lab is:

*benchmarks/histogram/src/cuda*

## Background

Histogram is a graphical representation of the distribution of data where items are on the x axis, and number of occurrences of the given item are on the y axis. In this exercise, we will make a histogram of a digital black and white photograph in which each pixel can have a value between 0 and 255 (inclusive). We will perform a fine grained histograming, with 256 bins where each possible pixel value will have it's own bin. Each bin *b* of the final histogram will contain the number of pixels who's value is *b*. Graphically, histogram of a photograph usually looks something like:



Serial (CPU) code that performs the histogramming operation would look like this:

```
memset(histogram, 0, 256 * sizeof(unsigned int)); //Initialize all bins to 0

for(int i=0; i<image_size; i++) //Loop through all the pixels

    ++histogram[image[i]]; //Increment the values of the bins
```

To perform parallel histogram however, you will need to access the histogram structure concurrently (bin increment). In order to do that, use the atomic addition operation `atomicAdd`. Please note that:

- Atomic functions are only available for devices of compute capability 1.1 and higher.
- Atomic functions operating on 32-bit integer values in shared memory and atomic functions operating on 64-bit integer values in global memory are only available for devices of compute capability 1.2 and higher.
- Atomic functions operating on 64-bit integer values in shared memory are only available for devices of compute capability 2.x and higher.

To check the compute capability of GPUs in Minotauro, you can execute the deviceQuery benchmark (look for Major and Minor revision number). To compile for the compute capability x.y, you need to append the "-arch=sm_xy" to the nvcc compilation line.

## Instructions

Step 1: Write a simple histogramming kernel (*`histo_simple`*)where both input data and histogram are stored in global memory.
Notice that the input data has **not** the same size as the kernel grid dimensions (you will need to take care of this in the kernel). Use the `atomicAdd` function to access the histogram data coherently:

```
unsigned int atomicAdd(unsigned int* address, unsigned int val);
```

*File: `kernel.cu`*

Step 2: Run the program
If you are using a cluster, modify the job script to execute this lab, and submit it using the corresponding command.

Step 3: Write histogramming kernel using privatization.
Uncomment the *`#define PRIVATIZATION`* line to expand the *`histo_kernel`* macro to *`histo_privatization`* and use the correct version.

*File: `main.cu`*

Implement *`histo_privatization`* where each thread block creates a private histogram of it's input partition in the shared memory.  Once private histograms are completed, start merging them into the global histogram. Each bucket of the global histogram is obtained by summing the corresponding buckets of the private histograms.

*File: `kernel.cu`*

Step 4: Run the program
Ensure that the *`#define PRIVATIZATION`* line in *`main.cu`* is not commented. If you are using a cluster, modify the job script to execute this lab, and submit it using the corresponding command.

# Lab: Vector Reduction

Your goal for this lab is to learn how to implement efficient reduction operations using CUDA. For that, you are required to implement a vector reduction in CUDA. In this lab you will learn:

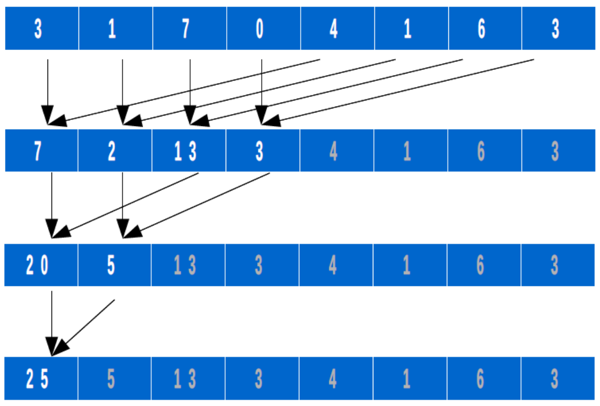
- How to use shared memory to minimize accesses to global memory
- How to use the synchronization primitives provided by CUDA

The directory for this lab is:

*`benchmarks/reduction/src/cuda`*

## Background

A reduction is performed by combining all elements of an array into a single value using some associative operator. Data-parallel implementations take advantage of this associativity to compute many operations in parallel, computing the result in `O(lg N)` total steps without increasing the total number of operations performed.
Each thread block reduces a chunk of the array. The reduction scheme to be used is shown below. First, each thread loads one value into shared memory. When this first step has finished, half of the threads are used to perform a reduction for a pair of elements (separated by a stride of `#threads` elements). In the next step, only a quarter of threads are used. The algorithm continues until a single thread is used to reduce the last two elements. This reduction scheme has benefits over other reduction patterns (e.g. reducing contiguous pairs of elements), in terms of minimizing thread divergence and shared memory bank conflicts, and maximizing memory coalescing when accessing to global memory.

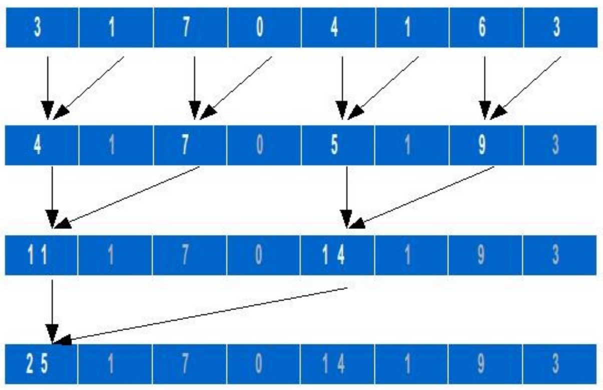


This stage produces an array of intermediate results (one output element per thread block). Therefore, the kernel can be called iteratively until a single thread block is needed to reduce the intermediate results, or the reduction can be finished by a loop executed in the host CPU.

## Instructions

Step 1: Write the naive kernel implementation
Complete the `reduction` kernel in which each thread block reduces a portion of the vector. Implement the reduction using the approach pictured below:

*File: `kernel.cu`*



Step 2: Fill the missing parts in host code

Add the necessary code to allocate the GPU buffers, do the corresponding copies, and invoke the kernel. Assume a block size of `BLOCK_SIZE` threads.

*File:* `main.cu`

Step 3: Run the program

If you are using a cluster, modify the job script to execute this lab, and submit it using the corresponding command.

Step 4: Write the kernel explained above

Complete the `reduction` kernel in which each thread block reduces a portion of the vector. Use the approach explained in the background of this lab. Values of the partial reductions must be written in contiguous memory locations indexed by the thread block index.

*File:* `kernel.cu`

Step 5: Run the program

If you are using a cluster, modify the job script to execute this lab, and submit it using the corresponding command.

# Lab: Prefix scan

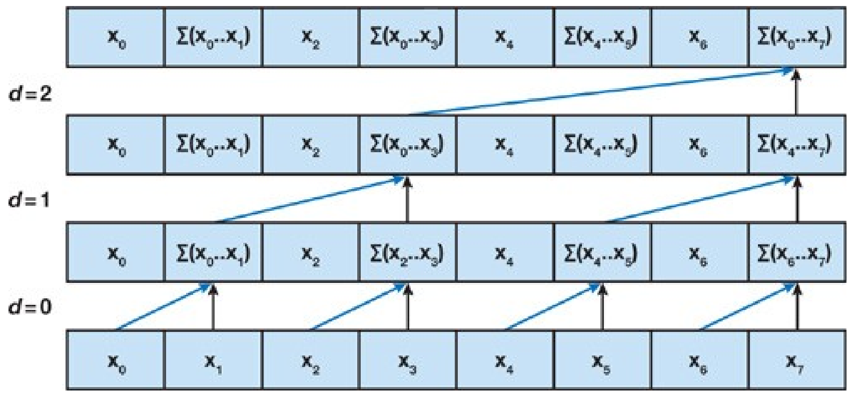
The directory for this lab is:

*benchmarks/prefixScan/src/cuda*

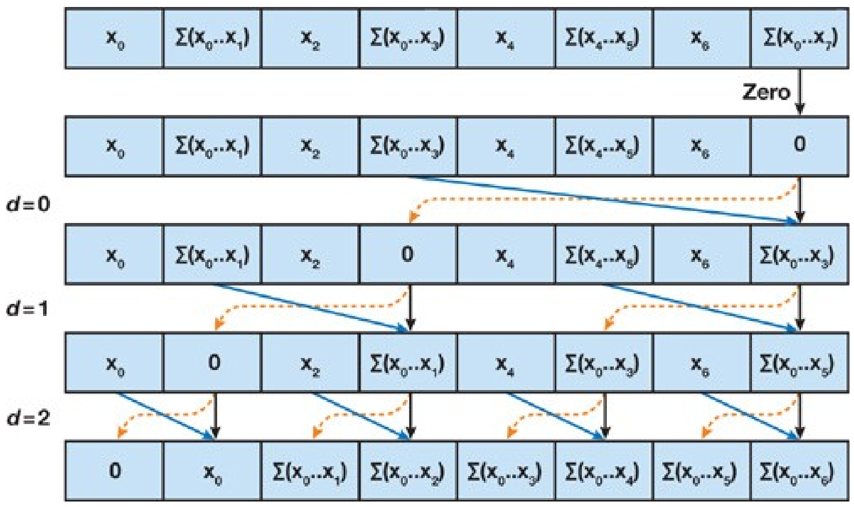
## Background

The all-prefix-sums operation on an array of data is commonly known as *scan*. There are many uses for scan, including, but not limited to, sorting, lexical analysis, string comparison, polynomial evaluation, stream compaction, and building histograms and data structures (graphs, trees, and so on) in parallel. An efficient parallel implementation of a scan operation consists of two phases:

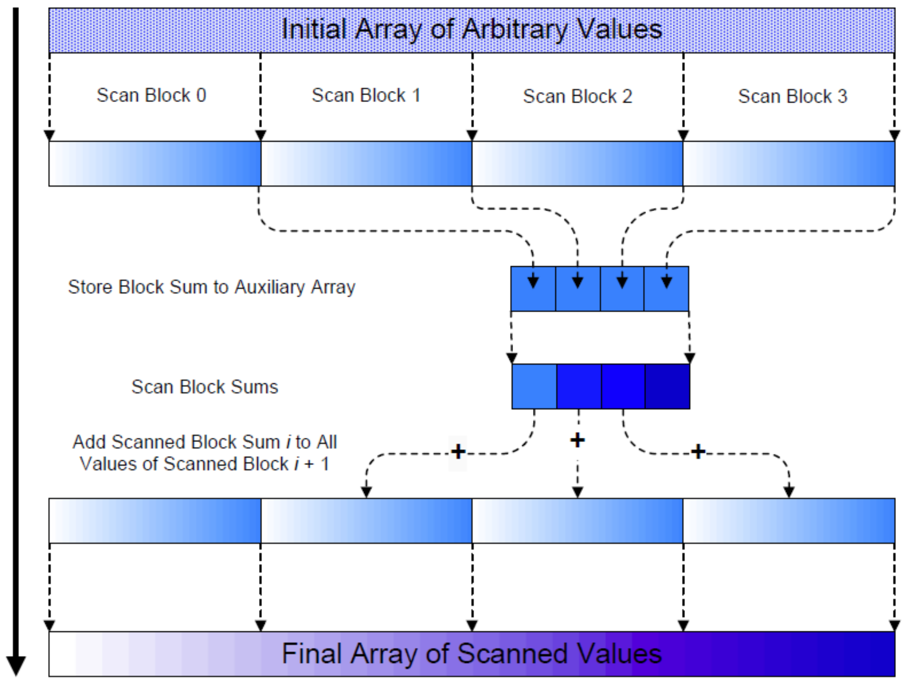
**Reduction phase**
In this phase, a typical reduction is performed to compute the contents of the last element of the array (which contains the sum of all the elements in the array). In this process, partial sums of the intermediate levels of the reduction tree are also stored in the array and are used in the second phase of the algorithm.



**Post scan phase**
When doing an exclusive prefix scan, this phase starts by setting the last element of the vector to 0. Then, in each step, each working thread adds the value *stride* positions away to its element (blue & black arrows), and stores the previous value of its element *stride* positions away (yellow dashed arrows). Note that initial stride is equal to half of the vector length and is halved in each step, while the number of working threads is doubled.



**Arbitrary input vector lenghts**
To handle large input vectors you will have to use several thread blocks. As depicted in the figure below, each thread block performs a partial prefix sum of a `BLOCK_SIZE * 2` segment of the vector. You will need to call the *`preScanKernel`* multiple times to handle large arrays, and also to perform the prefix scan of the partial block sums vector ("Scan Block Sums" in the figure).
Then, you will need a second kernel (*`addKernel`*) to add the partial sum (block sum) of all the previous segments to the elements of each `BLOCK_SIZE * 2` segment, to obtain the final prefix scan output vector.


Initial Array of Arbitrary Values
Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3
Store Block Sum to Auxiliary Array
Scan Block Sums
Add Scanned Block Sum *i* to All Values of Scanned Block *i* + 1
+
+
+
Final Array of Scanned Values

## Instructions

Step 1: Implement the kernels
Complete the kernels required to implement the prefix scan.

*File: `kernel.cu`*

Step 2: Implement the host code of the prefix scan
Complete the *`preScan`* function. It may need several kernel invocations to perform the prefix scan on the input vector.

*File: `kernel.cu`*

Step 3: Run the program
If you are using a cluster, modify the job script to execute this lab, and submit it using the corresponding command. The program must give correct results for the provided dataset (default).