PPCES 2012: GPGPU Programming Lab (Solution)

March 2012

https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/gpgpu-lab_ppces2012.zip

Sandra Wienke, wienke@rz.rwth-aachen.de

Abstract

This document guides you through the prepared examples and exercises. The purpose of the following tasks is to make you feel comfortable with the basic concepts of GPGPU programming, tuning and tool usage (using CUDA, OpenCL and PGI Accelerator).
For the first approaches, you will use the provided CUDA-capable laptops. In addition, you will test out the processing power of the available high-end GPGPUs belonging to the RWTH GPU.

Before you start, download the archive from the link above (or go the PPCES webpage www.rz.rwth-aachen.de/ppces and follow the link to the course material) and unzip it. Make sure that you work on your local hard disk drive, i.e. `C:`, on your laptop, since Visual Studio encounters problems accessing data on network drives.

If you need help or have any question please do not hesitate to ask!

1. NVIDIA SDK Examples

NVIDIA provides a CUDA and OpenCL programming and best practices guide, as well as numerous CUDA and OpenCL examples which are a nice starting point for familiarizing yourself with GPGPU programming.
We suggest to experiment with the NVIDIA SDK examples on the laptops provided since the SDK has already been properly setup here. If you rather like to use Linux, please refer to Appendix 10.1.

Using the provided Windows laptops, navigate to `All Programs -> NVIDIA Corporation -> NVIDIA GPU Computing SDK 3.2`. The menu entries `CUDA` and `OpenCL` comprise the corresponding documentation and links to the sources (`src`) and executables (`bin`) of the SDK examples. Furthermore, the `NVIDIA GPU Computing SDK 3.2 Browser` gives an overview of all examples and the possibility to execute them right away.

1.1. DeviceQuery

Before you start programming GPGPUs, it is always a good idea to verify that your available GPU resource is CUDA-capable and set up correctly. To this end, navigate to the `deviceQuery` example using the SDK Browser and execute it. If everything works properly, you will get a list of the most important features of your GPU. Complete Table 1 with your GPU details.

Table 1: Output of *deviceQuery*

Feature	Value: Laptop	Value: GPU-Cluster
Device number and name	Device 0: Quadro NVS 160M	Device 0+1: Quadro 6000
Number of cores	8	448
Max. number of threads per block	512	1024
CUDA version[1]	3.2	4.0
CUDA compute capability (cc)[2]	1.1	2.0

2. CUDA Example

During this task, you will write your first CUDA program, i.e. a Jacobi solver. The idea of this program is to get to know some basic concepts of GPGPU programming rather than to create a highly tuned application.
This **Jacobi** example solves a finite difference discretization (5-point-stencil) of the screened Poisson equation

$$\frac{\partial^2}{\partial x^2}u + \frac{\partial^2}{\partial y^2}u - au = f$$

using the Jacobi iterative method. To this end, the Jacobi method starts with an approximation of the objective function *u(x,y)* and reuses formerly-computed matrix elements to solve the current one. It iterates only about the inner elements of the 2D-grid (see Figure 1) so that the boundary elements are only used within the stencil. The solving process is aborted if either the residual becomes very small or a certain number of iterations is achieved.

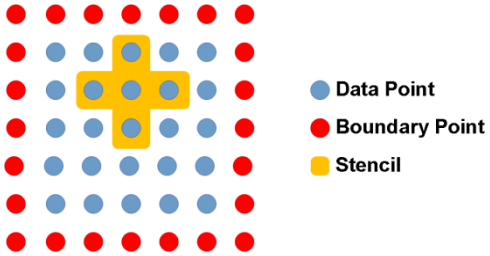


Figure 1: 5-point stencil

Hints for the implementation (see tasks below):

- The size of the arrays *u*, *uold* and *f* is `n = cols * rows`.
- Except the arrays, all kernel arguments (e.g. *ax*, *ay*, *b*) have already been defined within the source code.
- If you have created a correct implementation, the residual will have a value of approximately $6 * 10^{-10}$ and the iteration number will be 20 (for the default medium-sized data set).

Host reference version

Later on, you will compare the performance of a host multi-core architecture to the one of a GPU. To this end, you can use the OpenMP implementation in directory `C-omp-jacobi`.
On **Windows**, open the Visual Studio 2008 (!) project `JacobiOpenMP.sln`. Choose the `Release x64` configuration. Then, open the project's properties and navigate to `Configuration Properties -> Debugging -> Environment`. There, you can specify the number of threads by `OMP_NUM_THREADS=<no>`. Also set the `Command Arguments` to "`< jacobi.input.medium`".
On **Linux**, you can use (after compiling) `make run NTHREADS=<no of threads>` to try different configurations.

2.1. Getting Started

We recommend to use the provided **Windows** laptops for the first CUDA C (NVIDIA) and OpenCL C programming exercises. If so, you might want to use Visual Studio and CUDA syntax highlighting (refer to appendix 10.2). If you are just

[1] The CUDA version corresponds to the version of the CUDA Toolkit which comprises the CUDA compiler or CUDA libraries
[2] The compute capability (cc) corresponds to the core architecture of the GPU and describes the features supported by the CUDA-capable GPU. For instance, you need a device of cc 1.3 or higher to enable double precision floating point operations.

comfortable with **Linux**, you have to go to the RWTH's Linux GPU-Cluster. Follow the instructions in section 4.1 for login and setup. Linux Makefiles will be provided (cf. section 4.2). Refer to `make help` to get more information.

Additionally, you can choose between **CUDA C** (NVIDIA) and **CUDA Fortran** (PGI). See the corresponding subfolders in the GPGPU Lab archive. Note that CUDA with Fortran can only be used on the GPU-Cluster under Linux and not on the laptops! See sections 4.1 and 4.2. Be aware that OpenCL is only implemented as a C extension.

2.2. Writing CUDA Code

Go to the `C-cuda-jacobi` (CUDA C) or `F-cuda-jacobi` (CUDA Fortran) directory in the GPGPU Lab archive. Then, move to the `level01` subfolder.
On **Windows**, open the Visual Studio 2008 (!) project `JacobiCuda01.sln`.

Task

The first approach to parallelize the program should be to distribute the work just of the outer loop (i.e. iterations over rows of matrix) among the threads on the GPU. To simplify the reduction of the residuals, store an interim value for each row on the GPU and reduce the corresponding array of residuals on the host.
Examine the CUDA C file `jacobi.cu` or CUDA Fortran file `jacobi.CUF`, respectively, and work on the "*TODOs*" in the source code. You may have to add, delete or modify some code lines. We suggest to focus first on the data movements between host and device before moving to the kernel implementation. You may use the function `checkErr(err, __FILE__, __LINE__)` to verify that no errors occurred in your CUDA API calls. You can get the error code either by the return value of the CUDA API call or by `cudaGetLastError()`.

The slides from the morning session (including the colored *background slides*) might help you. You can also have a look at the `VectorAdd` example in the NVIDIA SDK or ask one of our team members, if you have any problems.

Solutions (i.e. source code files and a running program) can be found in folder `solution/C|F-cuda-jacobi/level01`.

2.3. Compiling & Executing CUDA Code

On **Windows**, you can use Visual Studio (project `JacobiCuda01.sln` has already been set up correctly) or the command line to compile and execute CUDA programs. We recommend using Visual Studio, but a description for the Windows command line is also included in appendix 10.3.
For Compiling, select the `Release x64` configuration. Then open the project properties (right-click on the project) and navigate to `Configuration Properties -> Debugging -> Command Arguments` and insert "`< jacobi.input.medium`". (Re-)Build the project. After a successful compilation, you can run the program with Ctrl+F5 (or `Debug -> Start Without Debugging`).

On **Linux**, use `make release` and `make run`.

Task

Compile your CUDA source code and execute it. Verify in the output that the solution error is really small (see hints above).

How many MFlops can you achieve on the GPU of the laptop?

122 (laptop)	MFlops
3,180 (cluster)	CUDA 1

As performance comparison of GPU and CPU, run the OpenMP version of the Jacobi program in the folder `C-omp-jacobi` (see "Host reference version" above for more information). First, set the number of OpenMP threads to one to get the serial execution time of the program. Then, increase the number of threads to 2, 4, 8, 12 and 16.

How many MFlops can you achieve at maximum?

1,080 (laptop)	MFlops
12,500 (cluster)	OpenMP

2.4. Profiling CUDA Code

For performance analysis, tool support is quite important. Besides CUDA debugging (see section 6), profiling tools are useful as they enable an analysis based on hardware counters. NVIDIA provides the *Compute Visual Profiler* for CUDA C

and Fortran programs on Windows and Linux operating systems. It can also be used for NVIDIA OpenCL and PGI Accelerator programs. We will see that later.

Task

1. First, create an executable of your Jacobi program.
2. On **Windows**, navigate to `All Programs -> NVIDIA Corporation -> CUDA Toolkit -> v3.2 -> Compute Visual Profiler` and start the Compute Visual Profiler.
   On **Linux**, make sure that the CUDA-module is loaded and execute the command `computeprof`.
3. Then create a new project.
4. In the `session settings`, give the session an arbitrary name and choose the file `profJacobi.sh` in the directory of your executable in the `Launch` dialogue box. This mini script just starts your executable and reads parameters from standard input.
5. For our purpose, it is enough to see same basic profiling results. Therefore, go the register `Profiler Counters` and disable all counters for `Cache` and all `Instructions`. Click on the button `launch`.

After a while, the profiler will list a `Profiler Output`. Have a look at the different columns and their values. Can you see your launch configuration?
Applying a right-click on the *Context* in the left hand-side pane, you can choose from different tables and plots. Have a look at the `Time Height Plot`. If you need help in understanding the plots/tables, ask one of our team members.

6. Save your project.

Solutions can be found in the directory `Profiling -> CudaOpenCL`. These profiles are done on the GPU-Cluster. Thus, you may have longer runtimes if you did profiling on the provided laptop.
In the `Profiler Output`, you can find the grid size [8,1,1] and the thread block size [256,1,1].

3. OpenCL Example

As you are now familiarized with the Jacobi example, you will implement it using OpenCL C (no Fortran available) in this exercise. Therefore, go the directory `C-ocl-jacobi` in the GPGPU Lab archive.

3.1. Writing OpenCL Code

On **Windows**, open the Visual Studio 2008 (!) project `JacobiOpenCL.sln`.
On **Linux**, OpenCL programs can be executed with the provided Makefiles. Here, you can either use the GNU compiler or the Intel compiler (`module switch <current compiler> <new compiler, e.g. intel or gcc>`).

Task

Use the same approach of parallelization like in the CUDA task. That means distribute the work just of the outer loop over the rows of the matrix among the GPU threads. Again, store interim residual values for each row on the GPU and reduce the corresponding array of residuals on the host.
First, open the file `main.c` and go to the function `InitGPU()`. Try to setup and initialize one GPU device by working on the "*TODOS*" within the text and using OpenCL API calls. You can use `checkErr(err, __FILE__, __LINE__)` to check on OpenCL errors. Second, move to the file `jacobi.c`. Allocate memory on the device, transfer data between host and device (`createBuffer()`) and launch the kernel analogously to the CUDA example. Third, open the file `jacobi.cl` which contains the actual kernel. Work on the "*TODOs*".

Solutions (i.e. source code files and a running program) can be found in folder `solution/C-ocl-jacobi`.

3.2. Compiling & Executing OpenCL Code

On **Windows**, as in the CUDA Visual Studio project, specify the `jacobi.input.medium` file as command argument. Then build and execute the project.
On **Linux**, use the provided Makefile (`make release` and `make run`).

Task

Compile the OpenCL source code and execute it. Verify that the solution error is sufficiently small.

How many MFlops can you get with OpenCL?

120 (laptop) 2,870 (cluster)	MFlops OpenCL

3.3. Profiling OpenCL Code

With NVIDIA's *Compute Visual Profiler*, you can also profile OpenCL code for the GPU.

Task

1. Open the profiler project that you have created in the last CUDA exercise.
2. Select `Session` in the menu and open the `Session settings`.
3. Choose the profJacobi.sh file in the directory of the OpenCL executable in the `Launch` section and start profiling.

Have a look at the `profiler output`. Now, right-click on the `Context` of this session. Choose `Comparison summary plot` and select the CUDA session that you have established before. Then, select `GPU time`. How does the runtime differ?

Solutions can be found in the directory `Profiling -> CudaOpenCL`.
In the case stated there, CUDA's GPU runtime was 2 % smaller (faster) than the one of OpenCL.

4. The RWTH GPU-Cluster

In this exercise, you will switch the programming environment and login to the RWTH GPU-Cluster since all following exercises must be done on the cluster. The GPU-Cluster comprises a total of 57 NVIDIA GPUs. These GPUs have compute capability 2.0 (Fermi architecture) and therefore enable higher peak performance than the GPUs embedded in the laptops provided and also enable features like double precision floating-point operations. The GPU-Cluster runs Linux as operating system and works with a module system such as the normal RWTH Compute Cluster. More information can be found on the slides *GPU-Cluster@RZ*.

4.1. Login & Setup

To use the GPU-Cluster, first login to a frontend node of the RWTH Compute Cluster (see handout *Acces to Lab Machines*) and then move to one of the following GPU nodes:

```
ssh -Y linuxgpus[20-24]
```

CUDA C

Now, make NVIDIA's CUDA toolkit available which provides, for instance, all CUDA runtime libraries:

```
module load cuda
```

We also suggest to switch the default Intel compiler to the GNU compiler, since the CUDA C compiler *nvcc* assumes the GNU compiler as default:

```
module switch intel gcc
```

CUDA Fortran

CUDA Fortran is a product of the Portland Group (PGI) in collaboration with NVIDIA. It is only available with the PGI compiler. Therefore, switch the default Intel compiler to the PGI compiler to use CUDA capabilities with Fortran:

```
module switch pgi/12.1
```

4.2. Compiling & Executing CUDA code

Copy the GPGPU Lab archive including your Jacobi sources from the laptop to your home directory on the cluster by aid of the SSH Secure File Transfer Client (cf. handout *Acces to Lab Machines*).

Afterwards, navigate to the CUDA Jacobi example. Compile it and execute it by:

```
make help
make release
make run [dev=<device ID>]
```

Since several users will use the same machine as you do and our GPUs can only be exclusively used by one user, it is possible that the GPU is already occupied and you get a corresponding error message. Try to use the second GPU in the node (`make run dev=1`) or try again after a few seconds.

How many MFlops can you achieve on the GPU-Cluster?

3,180	MFlops CUDA 1

4.3. CUDA Batch Job (optionally)

In the directory `C|F-cuda-jacobi/level01`, you can find an example GPU batch script (`batchCudaJacobi.sh`) that executes your CUDA Jacobi program in our LSF batch system.

Task

Modify this script by adapting the path and the e-mail address (if applicable). Now, you can submit your job to the LSF batch system:

```
bsub  <  batchCudaJacobi.sh
```

Type `bjobs` to see the status of your batch job (e.g. `PEND` for pending). After the job was scheduled and executed, you can find the results in the output file as specified in the batch script. You may want to continue with the next task, while waiting for the results.

5. PGI Accelerator Example

During this task, you will port the basic Jacobi example to the GPU using the directives-based PGI Accelerator Programming Model with C or Fortran. The PGI compiler is available on our Linux (GPU-)Cluster (not on your laptop):

```
module switch intel pgi/12.1
```

Note that you can easily activate collecting simple timing information by a compiler flag (i.e. `-ta=nvidia,time`). The corresponding feedback includes also information about the launch configuration. In our example, you may use a Makefile option to activate it:

```
make time=1
```

5.1. Getting Started

Similar to NVIDIA's Device Query example, PGI provides the `pgaccelinfo` command. Execute it and extend Table 1 by the values of the GPUs on the cluster as far as possible (device revision number = compute capability).

5.2. First Example

Go the directory `C|F-pgiacc-jacobi/level01` and open the file `jacobi.c` or `jacobi.F90`, respectively.
In the previous examples, we have swapped the pointers of *u* and the *uold* to exchange their content on the device. However, since PGI Accelerator does not support pointer assignment within accelerator regions at the moment, you have to use here a slightly different approach than in the CUDA and OpenCL examples. To this end, the content of both arrays shall be copied element-wise on the GPU. The rest of the parallelization concept of the previous examples can be taken for the PGI Accelerator approach.

Task 1

First, add accelerator regions (cf. the "*TODO*" in the source code) around (a) the swapping part of *u* and *uold* and (b) the nested loop for the computations. Manage the data transfer of the arrays using the `copy` clause at both accelerator regions. This is a very naive approach, but will show what can be done wrong. Compile the program using the provided Makefile (`make help` for more information). You will get a nice compiler feedback. Verify that there is a message "`Accelerator kernel generated`" which says that your specified region can be moved to the GPU. Also make sure that the reduction in recognized. Check which loop scheduling is automatically applied. Where is the data copied?

Compiler feedback of first approach:

```
Jacobi:
     58, Generating copy(uold[:n-1])
         Generating copy(afU[:n-1])
         Generating compute capability 2.0 binary
     64, Loop carried dependence of '*(uold)' prevents parallelization
         Loop carried backward dependence of '*(uold)' prevents vectorization
     67, Loop is parallelizable
         Accelerator kernel generated
         64, #pragma acc for seq
         67, #pragma acc for parallel, vector(256) /* blockIdx.x threadIdx.x */
             CC 2.0 : 16 registers; 4 shared, 64 constant, 0 local memory bytes; 100%
             occupancy
     76, Generating copy(afU[:n-1])
         Generating copy(afF[:n-1])
         Generating copy(uold[:n-1])
         Generating compute capability 2.0 binary
     82, Loop carried dependence of '*(afU)' prevents parallelization
         Loop carried backward dependence of '*(afU)' prevents vectorization
     85, Loop is parallelizable
         Accelerator kernel generated
     82, #pragma acc for seq
     85, #pragma acc for parallel, vector(256) /* blockIdx.x threadIdx.x */
         CC 2.0 : 33 registers; 1032 shared, 112 constant, 0 local memory bytes; 50%
         occupancy
         91, Sum reduction generated for residual
```

COPY means that data is transferred from host to device at the start of the accelerator region and data is moved back from device to host at the end of the accelerator region.

The feedback states that the outer loops are executed sequentially, whereas the inner loops are strip-mined on all multiprocessors, each executing the loop on the threads with a block size of 256.

Afterwards, open the *Compute Visual Profiler* again:

```
module load cuda
computeprof &
```

Profile the PGI Accelerator application (use `profJacobi.sh`)and have a look at the `Time Height Plot`. Can you see when the data is copied between host and device?

Solutions can be found in the directory `Profiling -> PgiAcc`. See the session saying "Naive Approach".

Task 2

Since the previous approach contained to many data transfers, enclose the while-loop with an `data region` and adapt the data clauses. Note that *uold* is only `local`ly used on the device.
Furthermore, set explicitly a loop schedule: Use the one from the CUDA and OpenCL examples i.e. map the outer loops to all threads and execute the inner loops `seq`uentially.

How many MFlops do you achieve using this PGI Accelerator version?

2,350	MFlops PGI Acc 1

Note that this version does more work (due to the modified swapping of *u* and *uold* and the implicit reduction) than the CUDA and OpenCL versions.

Since PGI Accelerator is a directive-based approach, you can easily ignore the accelerator directives using compiler flags. Execute your code only on the host by:

```
make [release | debug] ser=1
```

Solutions (i.e. source code files and a running program) can be found in folder `solution/C|F-pgiacc-jacobi/level01`.
An adapted profile can be found under `Profiling -> PgiAcc` and session "CorrectDataMovement".

5.3. Loop scheduling

You have already experimented with different loop schedules. Now, you should add a second level of parallelism to the accelerated regions. That means that the outer loops should be distributed to all blocks (multiprocessors) and work of the inner loops shall be done by one thread each. Look at the compiler feedback for verification.

How many MFlops do you get with this modified work distribution?

21,200	MFlops PGI Acc 2

Solutions (i.e. source code files and a running program) can be found in folder `solution/C|F-cuda-jacobi/level02`.

6. CUDA Debugger

Debugging tools are very important for a productive program development process. However, debugging support for GPUs is still not as popular as for CPUs. For CUDA C, there are the *Totalview* and *DDT* debugger under Linux and *Parallel Nsight* (Visual Studio Plugin) under Windows. In this task, you will get a short overview of the *Totalview* Debugger on our GPU-Cluster environment. Therefore, use the CUDA C Jacobi program in the directory `Debugging` in the GPGPU Lab achieve which contains some minor mistakes.

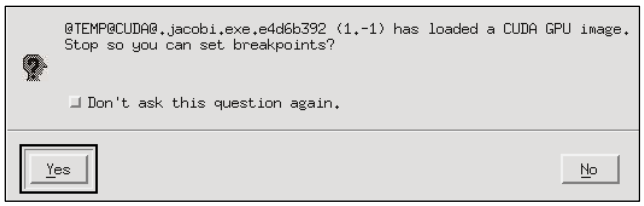
Task

1. Create a debug version of the program with `make debug`. Note that both flags "`-g -G`" are needed to create debug information (see command line).
2. Load Totalview in version 8.9.2-2 (!) from our module system: `module load totalview/8.9.2-2`
3. Start Totalview (specify the Jacobi program to debug): `totalview ./jacobi.exe &`
4. Click `OK` on the popped-up window.

In the symbol bar, you can find buttons for (re-)starting, halting or killing the debug session of your program.

5. Click on `Go` to start your program.
6. If your program contains any CUDA code, you will get a pop-up window in which you are asked whether you want to set any breakpoints in your CUDA code.
7. Click on `Yes`.
8. Now, set two breakpoints:

- Line 56: `if (idx >= 1 && idx <= rows-2)`
- Line 66: end of GPU kernel



9. Continue to run the program by clicking `Go` again. The debugger will stop at the first breakpoint in your CUDA code.
10. In the menu, navigate to `Tools -> CUDA Devices`. Can you see which devices are available on the machine?
11. See the `Stack Frame` and find the variable `idx`. What is its value?
12. Below the symbol bar, you can see fields which contains the block ID and thread ID of the current thread within the grid that was started on the GPU. Change the first dimensions of the block and the thread specifications. Compare the value of the `idx` variable.
13. Then, double-click on the variable (array) name `residuals` in the `Stack Frame`. This will open a window with details of the variable.
14. Double-click on the variable's address in the text field `value`.
15. Modify the type of the variable by adding the size of the array. Then, you will have: `@generic float[2000]` Have a look at the values of the array elements. Leave the window open for the next steps.
16. In the main window of Totalview, click on `Go` again to run to the next breakpoint.
17. Have a look at the values of the array elements again. Did all values change? Have a closer look at the very first element and the last elements of the array. What is wrong? Try to find the error in the program and fix it.:

7. CUDA Advanced

As you have seen in the last PGI Accelerator exercise, the kind of work distribution (besides leveraging low-latency memory) can be crucial to get good performance. In this task, you will implement the loop schedule from section 5.3 with CUDA.

Task

Start either from the CUDA version that you have written in the previous exercises or go to the directory `C|F-cuda-jacobi/level02` and open the Jacobi source code. In the latter, almost everything except the kernel will be given. Work on the "*TODOs*". If you use your own files, distribute the outer loop to the blocks on the GPU and the inner loop to the threads within the blocks. Use the device shared memory to store interim residual results for each thread within the block. At the end of the kernel, reduce these values to one residual value per block. Note that this should be done by only one thread.

How many MFLops do you get with CUDA and the new loop schedule including usage of shared memory?

27,200	MFlops CUDA 2

Solutions (i.e. source code files and a running program) can be found in folder `solution/C|F-cuda-jacobi/level02`.

8. Analyzing the impact of GPU parameters (optionally)

In this section, you will find out that the GPU parameter configuration can have a high impact on the performance (depending on the application). Here, we provide a C CUDA program for the SAXPY computation:

$$\vec{y} = \propto \vec{x} + \vec{y}$$

The output of the program lists performance metrics as runtimes and GFlops. They are obtained for a serial program version on the host computer, for the GPU kernel (just including the SAXPY calculations in the kernel) and for the whole GPU program run (including CPU-GPU data transfer).
You can use the following make targets and options:

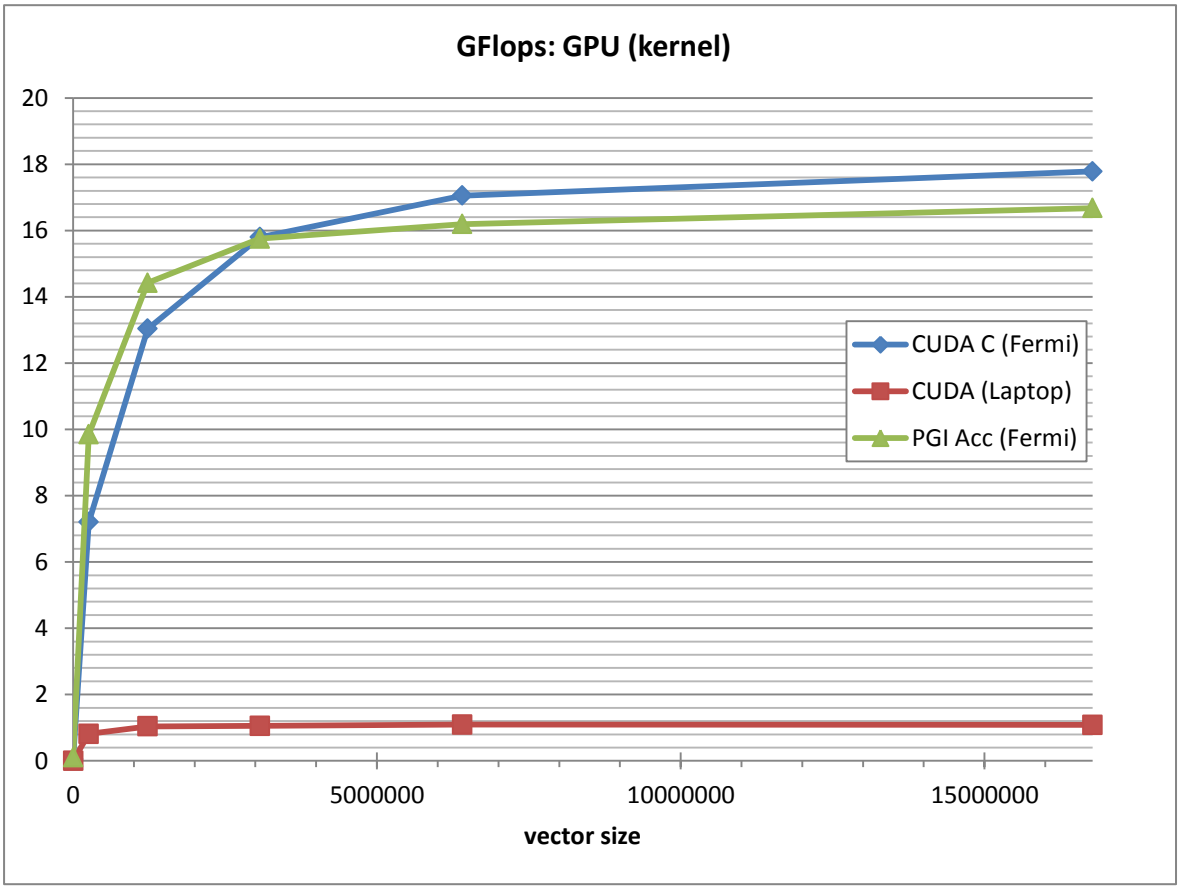
```
make [ release | debug ]
make run [N=<vector size> [TB=<#threads per block> DEV=<GPU id>]]
```

Task

1. **Impact of data size (=total number of threads (here))**
   While leaving the number of threads per block at the constant value of 256, vary the vector sizes (see Table 2) and have a look at runtimes, GFlops and speedups. Write down the GFlops in Table 2 or sketch your results in **Fehler! Verweisquelle konnte nicht gefunden werden.** and **Fehler! Verweisquelle konnte nicht gefunden werden.**. What can you conclude?
   Considering the (single precision) peak performances of the GPU, i.e. 1030 GFlops, and a memory bandwidths of 144 GB/s, are your results reasonable?

Table 2: GFlops of Fermi and laptop GPU (256 threads per block) using CUDA C

			Data size n	1 024	256 000	1 225 728	3 072 000	6 404 864	16 776 960
GPU-Cluster	CPU	GFlops		2.12845	2.56019	2.14846	1.948	2.00027	1.93227
	GPU (kernel)	GFlops		0.0499219	7.1999	13.0359	15.7973	17.0551	17.7826
	GPU (kernel+[3])	GFlops		0.0136468	0.432058	0.646805	0.701926	0.749237	0.794025



[3] „+“ denotes that the time for data transfer between CPU and GPU is included.

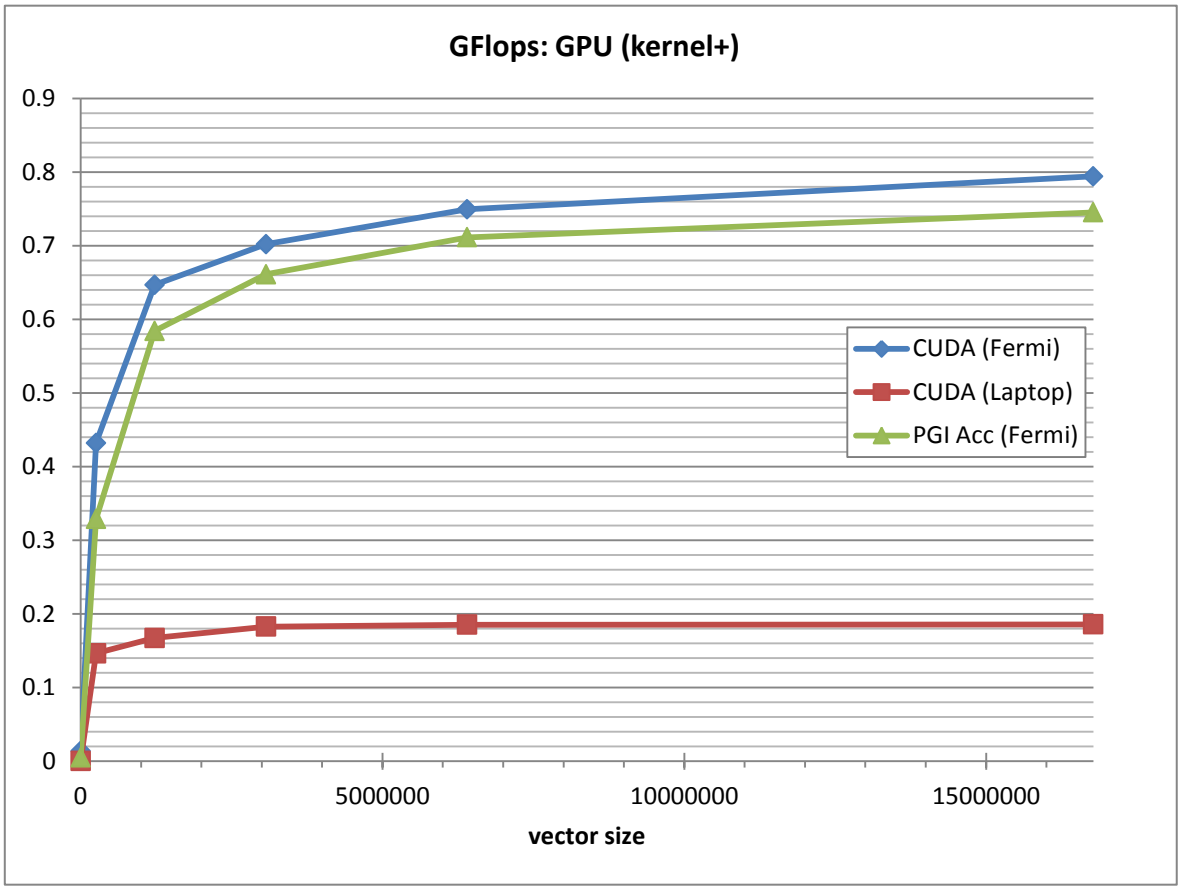


Figure 3: GFlops of GPU kernel (including CPU-GPU data transfer)

2. **Impact of launch configuration**

   Now, set the vector size constantly to 1 225 728 and vary the number of threads per block. Always chose a multiple of 32, as NVIDIA threads are executed in groups of 32 (called a *warp*) internally. For Fermi the maximal number of threads per block is 1024. Which is the best launch configuration? If you want, you can use **Fehler! Verweisquelle konnte nicht gefunden werden.** for writing down your results.

Table 3: Results for different launch configurations on Fermi using CUDA C

Threads per block	32	64	128	256	384	512	1024
GPU (kernel) GFlops	6.38343	9.68655	11.9549	13.2562	13.0348	12.9789	11.2513

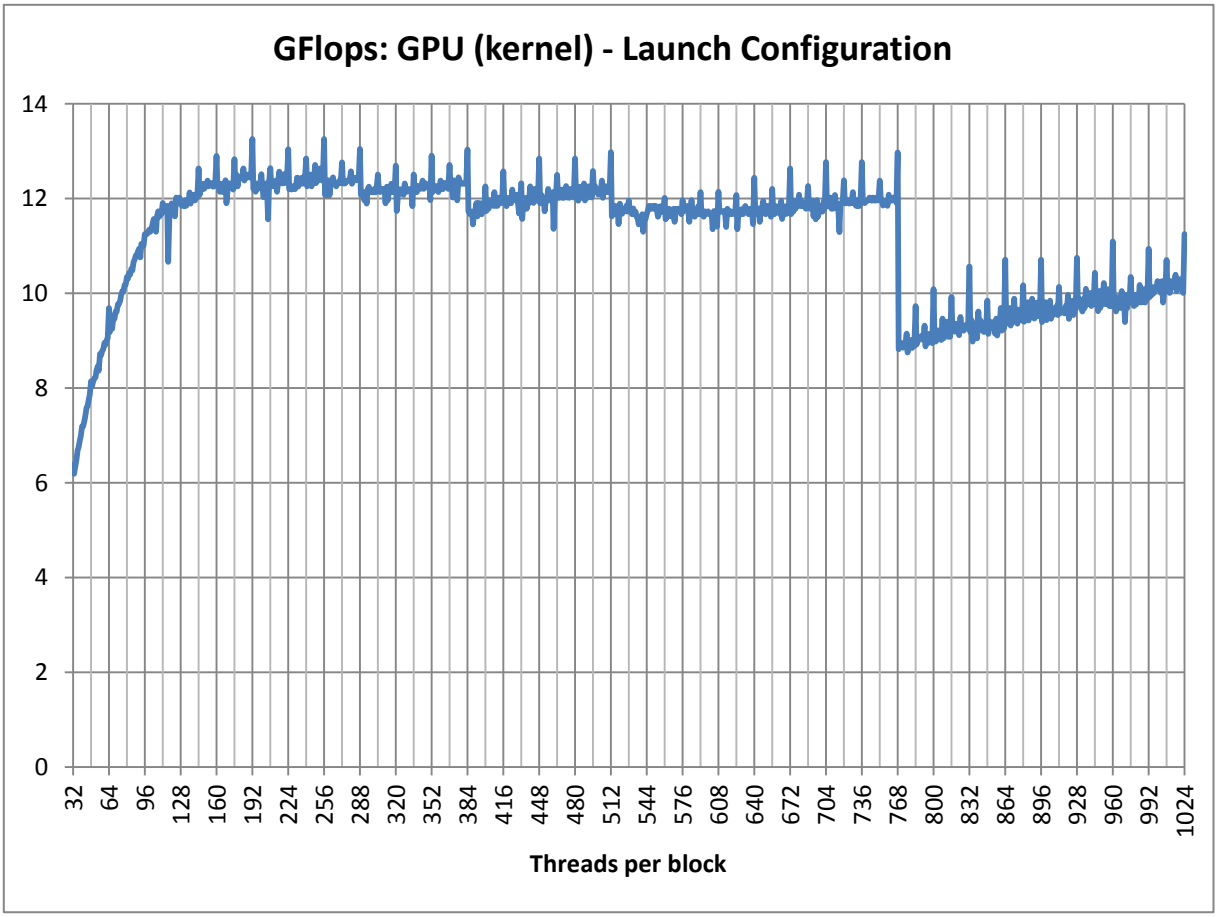


Figure 4: GFlops of GPU kernel on Fermi for different launch configurations

3. **Impact of memory throughput and number of floating point operations**

   Notice that our SAXPY program has only 2 floating point operations per 3 memory accesses and that one float is represented by 4 Bytes. NVIDIA's Fermi GPU has a theoretical peak memory bandwidth of 144 GB/s. Thus, we might get at *theoretical* maximum of (144 GB/s) / (4 B) * 2/3 Flop = 24 GFlop/s. Compare this value to your measured values. In general, a good way to improve the number of Flops is using shared memory. However, this is not applicable to our program. Instead, we investigate the impact of more floating point operations per memory access. Therefore, add "senseless" or even result-distorted operations to your GPU kernel (see comments in code). Make the same changes to the serial SAXPY computation and adjust the getGFlops() method. What is the impact on GFlops and speedup?

   One possible solution contains 20 floating point operations per (still) 3 memory accesses. Thus, the theoretical *peak of "SAXPY" moves to (144 GB/s) / (4 B) * 20/3 Flop = 240 GFlop/s. This example achieves about 165 GFlops for the GPU kernel and 7.3 GFlops for the GPU kernel+ using a vector size of 4793344 and 256 threads per block on Fermi.*
   In general, more floating point operations can usually better hide memory latency.

```
// GPU kernel function
__global__ void saxpy_parallel(unsigned int n, float a, float *x, float *y)
{
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n){
      float yy = y[i], xx = x[i];
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
```

```
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      y[i] = yy;
     }
}
```

```
// Compute SAXPY on CPU
void saxpy_serial(unsigned int n, float a, float* x, float* y) {
  for(unsigned int i=0; i<n; ++i) {
    float yy = y[i], xx = x[i];
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      yy = a*xx + yy;
      y[i] = yy;
  }
}
```

```
// Get performance metric gflops (time in sec)
float getGflops (double time, unsigned int n) {
        float gf = 0;
        unsigned long int operations;
        operations = (unsigned long int) (2 * n);
        gf = operations / time;
        gf /= 1000; // Kilo
        gf /= 1000; // Mega
        gf /= 1000; // Giga
        return gf*10;
}
```

9. Solutions

Solutions for all tasks are provided in the GPGPU Lab archive under `solutions`. There you can find a document comprising the tasks and their solutions, the basic source files and batch scripts.

10. Appendix

10.1. NVIDIA GPU SDK under Linux

If you want to use NVIDIA's GPU Computing SDK under Linux, you can login to our Linux GPU-Cluster and setup the CUDA environment (see section 4.1). Copy the SDK from `/rwthfs/rz/SW/nvidia` to your home directory and make the CUDA and OpenCL examples by:

```
make CUDA_INSTALL_PATH=$CUDA_ROOT
```

10.2. Getting started with Visual Studio & CUDA

Using Visual Studio IDE on the Windows laptops provided, it is a good idea to enable syntax highlighting for CUDA files (*.cu) first. Open Visual Studio 2008 (!) and select `Tools -> Options`. Then, open `text editor` in the tree view on the left, and click on `File Extension`. Type `cu` in the extension-box, set the editor to `Microsoft Visual C++` and click `Add`. Click `Ok` on the dialog box. Restart Visual Studio and the CUDA syntax will now be highlighted.

10.3. Compiling CUDA C Source Code on the Command Line (Windows)

For compilation without using Visual Studio IDE, go to the Windows command line (cmd) and set up the CUDA compiler manually. For the latter, notice that `nvcc` needs a supported host compiler which is the Microsoft Visual Studio compiler `cl.exe` on Windows. Therefore, you have to run the batch script `vcvarsall.bat` with the argument `amd64` for setting up the corresponding environment on the laptop. `vcvarsall.bat` can be found under `C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC`.

Then, you can compile your program:

```
nvcc [-arch=sm_<cc2>] saxpy.cu -Xcompiler "/DWIN32 /EHsc /W3 /nologo /O2 /Zi /MT"
```