PPCES 2011: GPGPU Programming – Lab (Solution)

25. März 2011

https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/gpgpu-lab.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 (using CUDA 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 compute cluster and practice dealing with our GPU batch system. This is also a good preparation for the usage of the GPU-Cluster (comprising approx. 50 NVIDIA Fermi graphics cards) that will be established to power the new CAVE of the RWTH Virtual Reality Group and for high performance GPGPU computing at the end of Q2 2011.
Before you start, download the archive from the link above 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. On Windows, navigate to `All Programs -> NVIDIA Corporation -> NVIDIA GPU Computing SDK 3.2`. *(Encountering problems, you may also use our Linux cluster, copy this archive from* `/rwthfs/rz/SW/nvidia` *to your home directory and make the CUDA examples in folder* `C`*.)* 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

Table 1: Output of *deviceQuery*

Feature	Value
Device number and name	Device 0: Quadro NVS 160M
Number of cores	8
Max. number of threads per block	512
CUDA version[1]	3.2
CUDA compute capability (cc)[2]	1.1

[1] The CUDA version corresponds to the version of the CUDA Toolkit which comprises the CUDA compiler or CUDA libraries (e.g. CUBLAS). A more recent toolkit version often leads to performance improvements.
[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.

Before you start programming GPGPUs, you should 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.

2. SAXPY using CUDA C

During this task, you will write your first simple CUDA program, i.e. SAXPY. The idea of this program is to get to know the basic concepts of GPGPU programming rather than to create a highly tuned application.

SAXPY = Scalar Alpha X Plus Y: $\vec{y} = \alpha \cdot \vec{x} + \vec{y}$
Thus, a serial implementation looks like the following:

```
for (int i=0; i<n; ++i) {
   y[i] = a * x[i] + y[i];
}
```

2.1. Getting started with Visual Studio + CUDA

Using Visual Studio IDE, 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.

2.2. Write CUDA C source code

Open the Visual Studio 2008 (!) project `Saxpy_cuda` (in the directory `<archive_path>\gpgpu-lab\C-cuda-saxpy`). Examine the CUDA file `saxpy.cu` and work on the "TODOs" in the source code. The slides from the morning session 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.

Also compare the source files in the solution folder.

```
// GPU kernel function
__global__ void saxpy_parallel(unsigned int n, float a, float *x, float *y)
{
    // TODO: Compute index i using thread and block indices
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

    if (i < n){
      y[i] = a*x[i] + y[i];
     }
}
```

```
int main(int argc, char* argv[])
{
  // malloc and initialize

  // TODO: Allocate d_x and d_y on the device
  cudaMalloc(&d_x, n* sizeof(float));
  cudaMalloc(&d_y, n* sizeof(float));

  // TODO: Copy h_x into d_x, and h_y into d_y
  cudaMemcpy(d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice);
```

```
  // TODO: Define 1D thread blocks of length blockDim
  dim3 threadsPerBlock(blockDim);
  dim3 blocksPerGrid((n%blockDim==0)? n/threadsPerBlock.x : n/threadsPerBlock.x +1);

  // TODO: Invoke parallel SAXPY kernel
  saxpy_parallel<<<blocksPerGrid, threadsPerBlock>>>(n, a, d_x, d_y);

  // TODO: Copy d_y into h_y
  cudaMemcpy(h_y, d_y, n * sizeof(float), cudaMemcpyDeviceToHost);

  // TODO: Free memory on device (d_x, d_y)
  cudaFree(d_x);
  cudaFree(d_y);

  // free
}
```

2.3. Compile your CUDA code

The CUDA compiler is called `nvcc` and is shipped with the CUDA Toolkit from NVIDIA. You can use it with Custom Build Rules in Visual Studio (project is already set up correctly) or on the command line. We recommend using Visual Studio, but a description for the command line is also included.

Visual Studio: For Compiling, select the `Release x64` configuration and (re-)build the project. *(If you have problems during compilation, make sure that the CUDA compiler is set up correctly. Therefore, go the project -> Custom Build Rules. Verify that a CUDA Runtime API Build Rule is selected. Furthermore, choose the Linker menu in the project's properties -> General -> Additional Library Directories. There, an entry such as $(CUDA_PATH)\lib\$(PlatformName) should be denoted. In addition, check that cudart.lib is given in Linker -> Input -> Additional Dependencies.)*

Command line: 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`. Compile your program:
`nvcc [-arch=sm_<cc[2]>] saxpy.cu -Xcompiler "/D WIN32 /EHsc /W3 /nologo /O2 /Zi /MT"`

After a successful compilation, run your program (Ctrl+F5 in Visual Studio). Verify in the output that the error check is passed.

In the following tasks, you may also denote two optional parameters: The first parameter specifies the vector size and the second one defines the number of threads per block. To assign them using Visual Studio, go to the project's `Properties -> Configuration Properties -> Debugging` and set them up in `Command Arguments`. You might get runtime errors depending on your configuration as the GPU hardware is restricted in memory and number of threads (cf. output of `deviceQuery` in Task 1.1).

Output of the program: runtimes and GFlops may differ

```
Vector size n:  262144
Scalar a:       2.000000
#Threads/block: 256

Running on device 0
Error check passed
                                | Runtime [s]    | GFlops         | Speedup        |
================================================================================
CPU (serial)                    | 0.0002198219   | 2.385058       | 1.000000       |
GPU (kernel)                    | 0.0001030855   | 5.085952       | 2.132423       |
GPU (kernel + data transfer)    | 0.0013240265   | 0.395980       | 0.166025       |
```

3. SAXPY & GPU batch system

To leverage the processing power and features of high-end GPGPUs, such as NVIDIA's Fermi GPU, you can use our GPU batch system. In this section, you will practice using our GPU batch system under Linux and you will investigate qualities and trade-offs of GPGPUs. Compare the slides GPUs@CCC from the demo for further information of our GPU batch system.

3.1. GPU batch system

First, copy the GPGPU Lab archive including your SAXPY sources written in Task 2 to your home directory on the Linux cluster by aid of the SSH Secure File Transfer Client. Then, log into one of our Linux cluster frontends `cluster-x` or `cluster-x2` using X-Win32. In the directory `<archive_path>/gpgpu-lab` you can find an example GPU batch script (`gpuBatchScriptExample.sh`) which would run the SDK `deviceQuery` in batch *(assuming you copied the NVIDIA SDK and made the examples as described in Task 1 on Linux)*. Modify this script file for the execution of the SAXPY program. First, change the name of the job and of the output file. Switch the module to use a compiler appropriate for the program (`module switch intel gcc/4.5`), adjust the program path and call the program using the Makefile.

```
make CCAP=20 # for Fermi
make run [N=<vector size> [TB=<#threads per block>]]
```

Notice that the flag `gpu_slots` is obligatory and that `gtype` can be set either to `fermi` or to `gt200` (denoting the GPU architecture to run on). Furthermore, notice that the environment variables `PATH` and `LD_LIBRARY_PATH` must be extended for the usage of CUDA. *(If you run CUDA interactively, you can source the provided file `cuda.init` to set up the CUDA developing environment.)*
Now, you can submit your job to the GPU batch system:

```
qsub <script name>
```

Type `qstat` to see the status of you batch job (qw=waiting or r=running). `qdel` can be used to delete the job if necessary. After the job was scheduled and executed, you can find the results in the output file as specified above. Continue with the next task, while waiting for the results.

Example for a batch script

```
#!/usr/bin/env zsh
#$ -o $HOME/temp/SAXPY_CUDA.out      # stdout to file
#$ -j y                # merge stderr to stdout
#$ -N CUDA_Saxpy       # job name
#$ -l gpu_slots=1      # requested #GPUs (required)
#$ -l gtype=fermi      # GPU architecture type
#$ -l ostype=linux     # requested OS
#$ -l h_rt=00:15:00    # requested real time
#$ -l h_vmem=512M      # requested memory

module switch intel gcc/4.5

LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
PATH=/usr/local/cuda/bin:$PATH

cd $HOME/gpgpu-lab/C-cuda-saxpy
make CCAP=20
make run N=16776960
```

3.2. Analyzing the impact of changed GPU parameters

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). Changing GPU parameters may affect the program performance. Therefore, examine the impact of the issues below by comparing runtimes and GFlops.

To gain appropriate results for a performance analysis, running the tasks in batch mode is essential. However, our GPU batch system may take longer to schedule the jobs of all lab participants. We have two recommendations to tackle this problem. (1) Execute the following subtasks using as less submitted compute jobs as possible. In addition, continue with Task 4 while waiting for the results. Thus, analyze the impact of changed GPU parameters at the end of the lab session. (2) Log into the machine (`linuxnc005`) that comprises two NVIDIA C1060 GPUs which are the predecessor architecture of Fermi. There you can interactively run your program. However, the usage of each GPU is restricted to one user that means if both GPUs are occupied, either you get an error message *("All CUDA-capable devices are busy or unavailable")* or the error check fails. In both cases, you have to manually try it later again. Be also aware that runtimes and GFlops of the serial CPU program run are not reliable if several users work on the same machine.

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 Figure 1 and **Fehler! Verweisquelle konnte nicht gefunden werden.**. What can you conclude?
2. **Comparison of GPU architectures**
   Use the same vector sizes for measurements on your laptop and complete Table 2 (or Figure 1/Figure 2). How big is the performance difference? Considering the (single precision) peak performances of these GPUs, i.e. 34.8 GFlops for the laptop GPU and 1030 GFlops for Fermi, and memory bandwidths of 11.2 GB/s and 144 GB/s, respectively, 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
Fermi	**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
Laptop	**CPU**	**GFlops**	0.558612	0.763966	0.970336	0.901568	0.859316	0.865673
	GPU (kernel)	**GFlops**	0.00089636	0.810052	1.03926	1.05299	1.09224	1.08217
	GPU (kernel+[3])	**GFlops**	0.00052012	0.146417	0.167361	0.182872	0.185372	0.185746

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

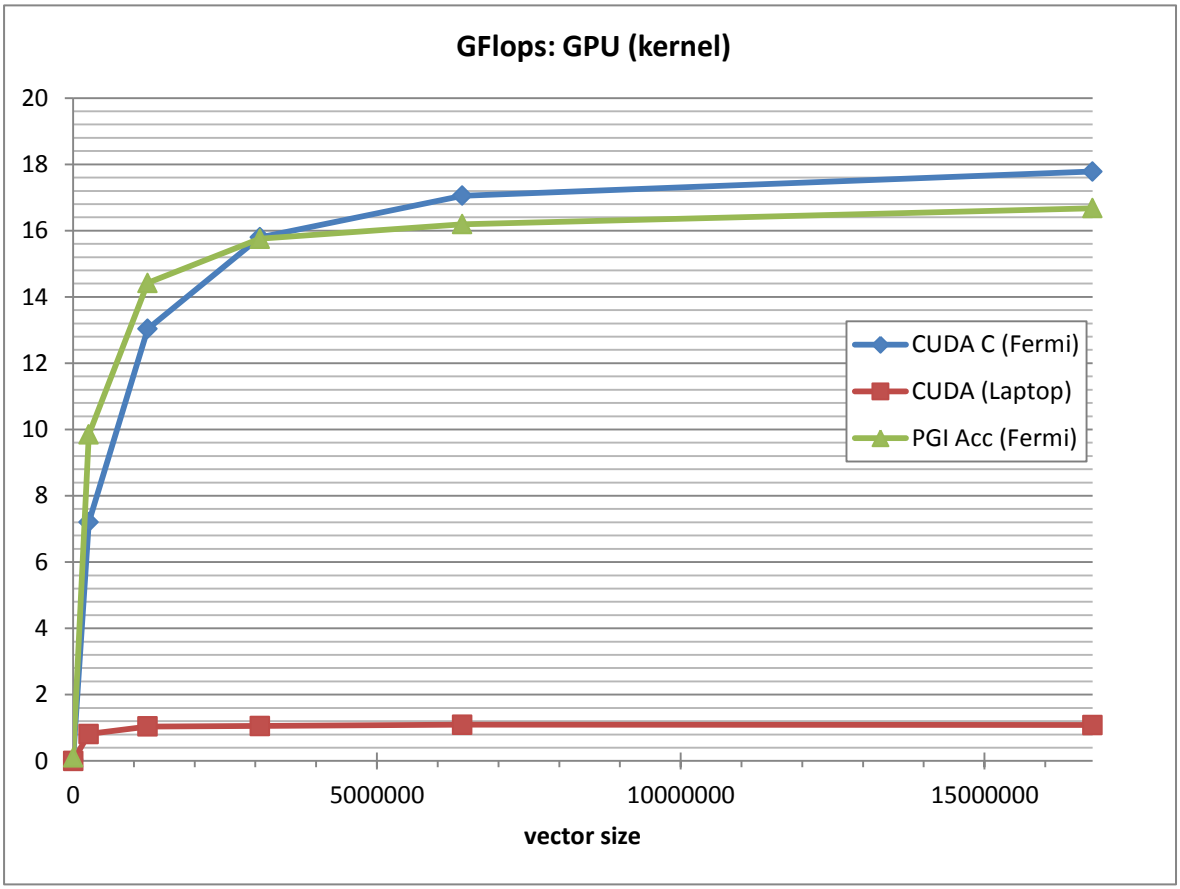

Figure 1:GFlops of GPU kernel

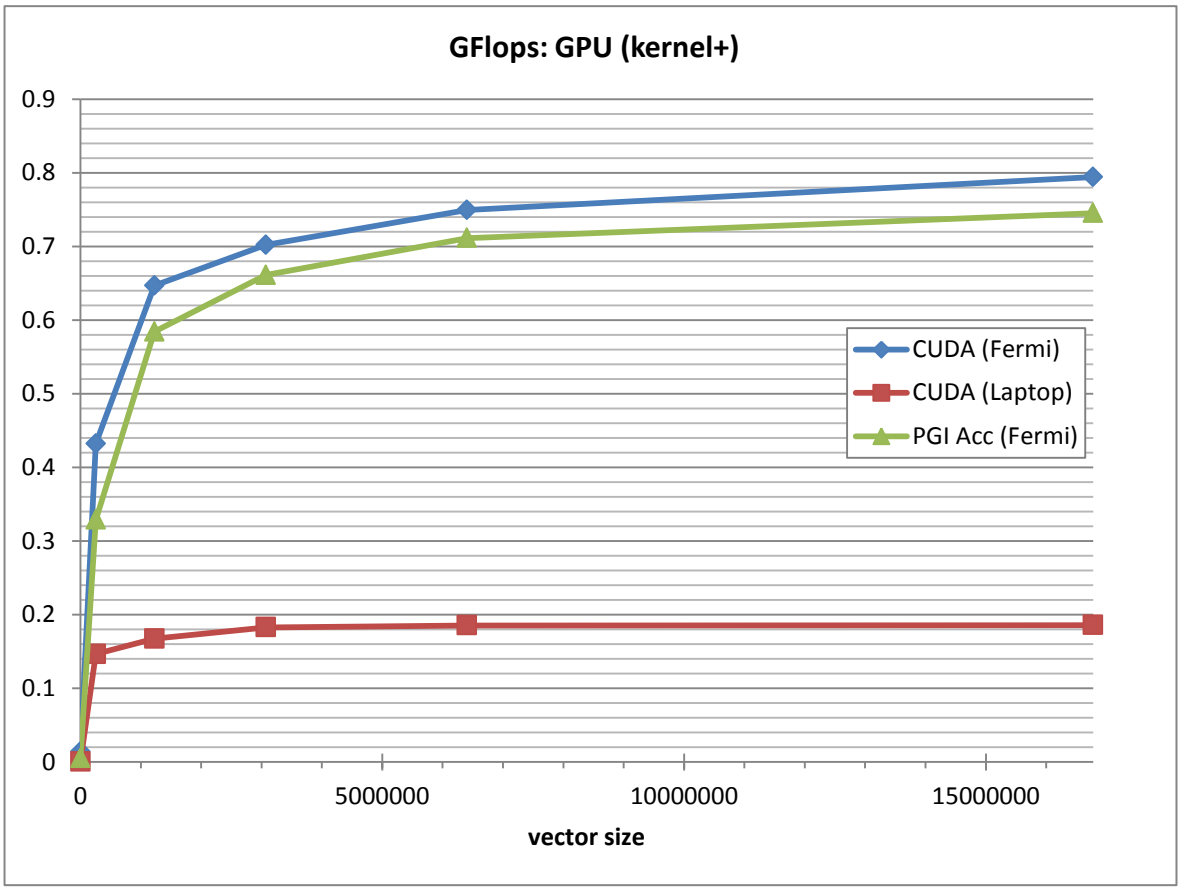

Figure 2: GFlops of GPU kernel (including CPU-GPU data transfer)

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

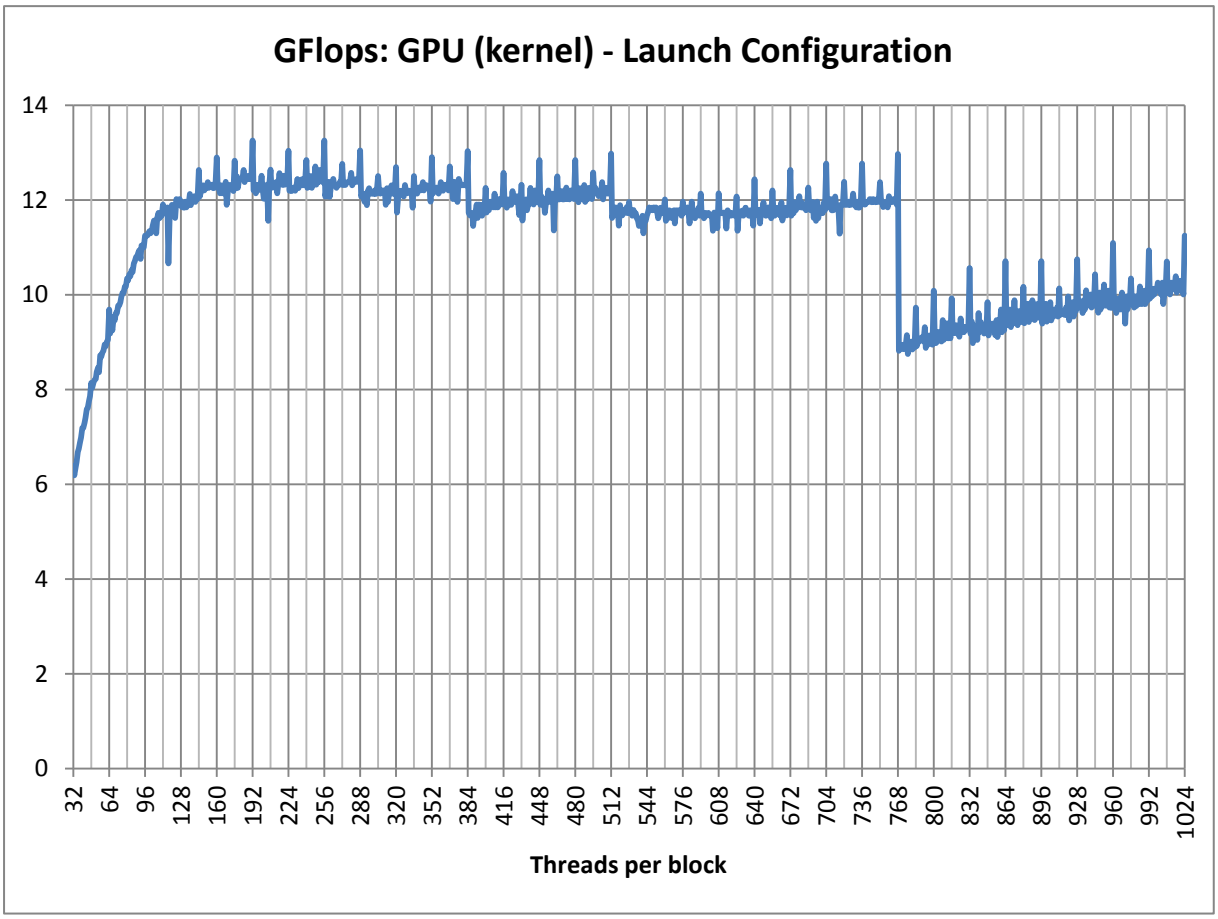

Figure 3: GFlops of GPU kernel on Fermi for different launch configurations

3. **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 Table 1Table 3**Fehler! Verweisquelle konnte nicht gefunden werden.** for writing down your results.

   See Figure 3 for results.

4. **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 a *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. 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;
      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;
}
```

4. SAXPY using PGI Accelerator

During this task, you will move the SAXPY computations to the GPU using the directives-based PGI Accelerator Programming Model. The PGI compilers are available on our Linux cluster (not on your laptop). Therefore, remain on `cluster-x` and load the pgi module:

```
module switch <lastLoadedCompiler (default: intel)> pgi/11.1
```

Program compilation will work on every cluster frontend, however, for execution you must either be logged into one GPU machine or use the GPU batch system.

4.1. Accelerator region

Go the directory `<archive_path>/gpgpu-lab/C-pgiacc-saxpy` and open the file `saxpy.c`. At first, just add an accelerator region (cf. the "TODO" in the source code). Compile the program using the provided Makefile (`make CCAP=20` for Fermi). 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.

4.2. Data clauses

Now, have a look at the data copy statements in the compiler feedback. Which variables are copied from the host to the device and which ones are copied from device to host? Which data amounts are copied? Try to specify the corresponding data clauses manually in your source code. Check the compiler feedback if your implementation is (still) correct.

4.3. Loop mapping

Third, examine the used loop schedule referring to the compiler feedback. What does "`parallel, vector(256)`" mean? How is the work distributed onto the GPU? Refer to Michael Wolfe's slides or have a look at the PGI Accelerator programming guide (http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf). Add a loop directive with the same schedule to your SAXPY program. Verify the correctness by the compiler feedback.

```
int main(int argc, char* argv[])
{
  // malloc and initialize

  // TODO 1: Define accelerator region
  // TODO 2: Add data clauses
  // TODO 3: Define loop scheduling
#pragma acc region for copy(y[0:n-1]), copyin(x[0:n-1]) parallel,vector(256)
  {
    for (unsigned int i=0; i  < n ; ++i) {
      y[i] = a*x[i] + y[i];
    }
  }
  // free
}
```

4.4. Analyzing the performance

To execute your SAXPY PGI Accelerator program, we provide the batch script `pgiBatch.sh`. Adjust the program path in the script and submit it to our GPU batch system. The output can be found in `$HOME/temp/SAXPY_PGI.out`. Notice that additionally to the program output, the kernel timing data is displayed (since we used the compiler option "`time`"). It gives you an overview of the time spent in the GPU kernel (`->kernels`) and for data transfer (`->data`). Note the runtimes for

kernel execution and the whole accelerator region, i.e. kernel execution and data transfer, into Table 4 or add the results to Figure 1 and Figure 2.
Compare the GFlops of the kernel execution using PGI Accelerator to the corresponding GFlops using CUDA C.

Table 4: Performance metrics for Fermi using PGI Accelerator

		Data size n	1 024	256 000	1 225 728	3 072 000	6 404 864	16 776 960
Fermi	**GPU (kernel)**	**Time**	0.000019	0.000052	0.00017	0.00039	0.000791	0.002012
		Flop	2 048	512 000	2 451 456	6 144 000	12 809 728	33 553 920
		GFlops	0.10778947	9.84615385	14.4203294	15.7538462	16.1943464	16.6768986
	GPU (kernel+)	**GFlops**	0.00533536	0.32886431	0.58408052	0.66113722	0.71141112	0.74531376

Notice that the timing data also indicates the launch configuration chosen by the PGI compiler (cf. statements of `block` an `grid` size). Now, modify the loop scheduling in your program. First, just change the width of the vector schedule (choose a multiple of 32) and examine the launch configuration and the difference in performance. Also, try completely different loop schedules.

This document gives an overview of possible solutions for tasks 1 to 4. In the directory `<archive_path>/gpgpu-lab/solution` You can also find the basic source files and batch scripts.