The Art of Parallel Processing using General Purpose Graphical Processing units

–Hardware, CUDA introduction, and Software architecture

Technical Report

Tino Lourens
TiViPE
Kanaaldijk ZW 11
5706 LD Helmond
The Netherlands
tino@tivipe.com

June 23, 2009

Contents

Abstract

The aim of this report to elaborate on general purpose graphical processing unit (GP-GPU) programming and provide a cookbook for programming NVIDIA's compute unified device architecture (CUDA). CUDA contains a programming language that is very similar to C, and thus easy to program. Programming CUDA appeared to result in algorithms that are at least one order of magnitude faster than a best effort multi core SSE implementation. When normal C/C++ code was used the differences where 2 to 3 orders in magnitude.

The architecture of the GPU is elegant which makes it easy to construct a hybrid model of data-transfer and pure instruction (or floating point) processing that is much easier to understand than a SSE or cache optimized CPU model.

The GPUs as computational units rapidly evolving compared to the CPU, the gap between CPU on both data-transfer and computational power is more than 10 fold, making the GPU an excellent candidate for real time parallel data processing. The latest generation GPUs has become powerful enough to make a single PC solution to become sufficient to control a for instance a medical imaging system. It implies size reduction, cost reduction, energy reduction, programming effort reduction, and lifting on off-the-shelf consumer technology.

A follow up report on CUDA parallel computing using TiViPE is available [2]. In this report TiViPE programs using parallelism will be discussed together with the respective computational times for different datatypes.

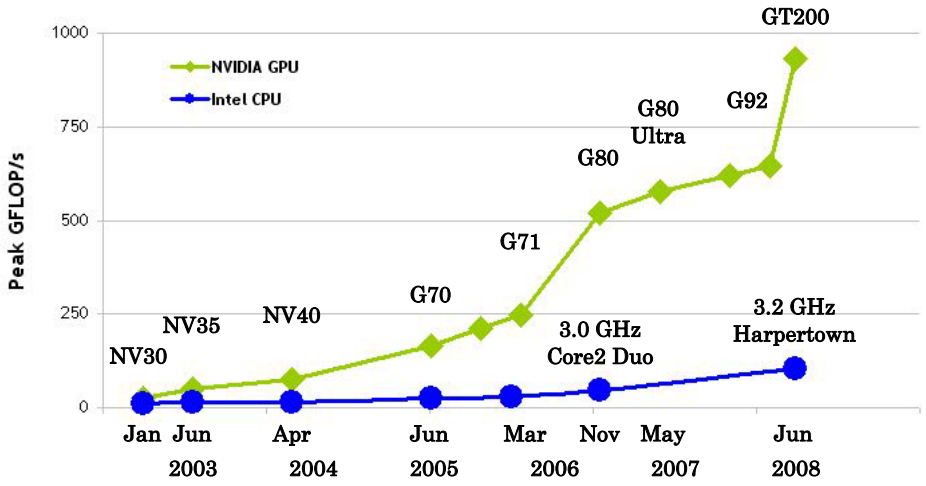

Figure 1: *CPU and GPU performance development. In Q2 2008, the high-end Intel CPU processor is an octo (dual-quad) core with a peak performance of around 100 GFLOPS while the latest GeForce GPU have a peak performance of 1 TFLOPS.*

1 Introduction

Hardware is rapidly evolving and has made a shift towards parallel computing by producing multiple cores on a single processing unit. This holds for both a central processing unit (CPU) with 2, 4, or 8 processing units and a graphical processing unit (GPU) with at most 16 multiprocessors having each 8 to 24 stream processors. The computational speed of processors has increased substantially over the past decades. The current CPUs are in a 10-20 giga (10^9) floating point operations per second (GFLOPS) range per core, yielding a total of 80 GFLOPS for a high-end quad core. GPUs have staggered to a 200-600 GFLOPS range (583 GFLOPS for an NVIDIA 8800 Ultra), and it is a fact that in Q2 2008 the GPUs have hit the tera (10^{12}) floating point operations per second (TFLOPS) domain, since NVIDIAs 9800GX2 and the 280GTX with peak performances of 1.152 and 0.933 TFLOPS, respectively. In Q1 of 2009 a shift was made form 65 to 55 nm technology resulting in a 285GTX and a 295GTX with peaks of 1.062 and 1.788 TFLOPS, respectively. The competition between AMD/ATI and Nvidia is the force between this technology development. A third competitor is Intel's Larrabee, and it is likely to enter the market by 2010.

In a matter of just a few years, the programmable graphics processor unit has evolved into an absolute computing workhorse, as illustrated in Figure 1. With multiple cores driven by very high memory bandwidth, today's GPUs offer incredible resources for both graphics and non-graphics processing. The main reason behind such an evolution is that the GPU is specialized for compute-intensive, highly parallel computation.

Currently the market has voted for Blue-Ray DVD and full HD, and the LCD screen standard is set to 24 inch. It implies that the number of pixels per display has increased by a factor 2 to 4, demanding GPUs to evolve rapidly.

Also software developments are evolving. In 1992, Silicon Graphics Incorporation (SGI) introduced the open graphical language (openGL) standard that is still widely used. In 1995, Microsoft acquired RenderMorphics, its RealityLab engine was integrated into Direct3D. In 2001, NVIDIA corporation introduced Cg "C for graphics" to bridge the gap between OpenGL and Direct3D. A recent

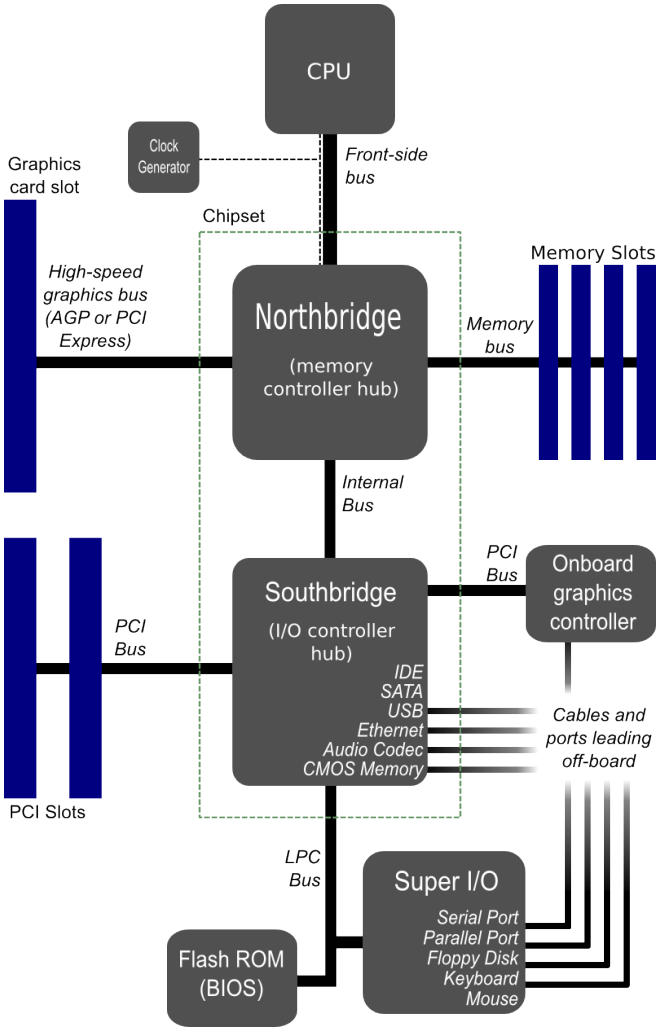

Figure 2: *Intel PC architecture.*

trend in computer science in stream computing is general purpose computing on GPUs (GP-GPU). In 2007, NVIDIA started to supply this trend by the compute unified device architecture (CUDA) together with an easy to use software development kit (SDK).

This report is outlined as follows: Section 2 gives an overview of the Intel architecture and its bottlenecks with regard to data transfer (Section 4). Its followed by a a section on GPU hardware. Section 5 provides an introduction to CUDA programming. The following section, provides an in depth explanation on GPU programming. It also gives a general framework for both CPU and GPU processing. Section 6 elaborates on a model that comprises of a data transfer and a computational part to predict the time required to process an algorithm.

2 CPU architecture

Figure 2 illustrates the PC architecture. Let us consider a data stream from I/O devices to display step by step, and assume that a video stream with images of 1MB of data travels from one place to another.

The commonly used I/O devices are USB 2.0 with a bus-speed of 480 Mb/s, SATA 3.0 Gb/s, ethernet at 1Gb/s and recently 10 Gb/s, and PCI-express solutions like infiniband at speeds of 10, 20, and 40 Gb/s.

Given a giga bit ethernet we can process at most 1Gb/s is equivalent to 125MB/s and in practise due to the TCP-IP/UDP protocol where data is added to a package of 1500 or 9000 bytes the performance is between 70 and 90 percent of this 125MB/s, so it implies that around 100MB can be processed per second. Latter is equivalent to 100 frames per second. Hence if more data needs to be processed, it is important to use 10 giga bit ethernet. It is able to process 10 times as much data as the giga bit ethernet yielding a performance of 1GB/s.

The speed of the internal bus has a direct media interface (DMI), it is approximately a PCI-Express x4 that has the speed of 1 GB/s. If disc I/O needs to be performed parallel to sending data by ethernet this internal bus might become a bottleneck. A solution to this problem is to bypass the internal bus and connect directly to the memory controller using the so-called infiniband connection that is available at speeds of 1, 2, and 4 GB/s. However, connecting several PCs will result in multiple hubs, since the number of PCI-Express slots is limited to at most four on most motherboards. In 2009, ASUS came with a "P6T7 WS SuperComputer" motherboard, that contains 7 PCI-express slots with 16 lanes. The term supercomputer refers to the GPU potential of 4 CUDA cards, leaving still space to install high-bandwidth interfaces like infiniband.

The CPU is a calculator able of processing instructions in parallel, currently up to 8 cores. To obtain data for processing it is coupled to a front side bus (FSB) that is able to process up to 12.8GB/s. A dual lane with a speed just half that of the front side bus couples memory to the CPU. In case when 8 cores are used and these cores would all need the maximum data transfer around 800MB/s for a 10.6GB/s FSB per core could be achieved. For 1MB of image data per core it would imply that 1.25 ms are needed to get data to a core. Its is seems to be a huge bandwidth, but when several intermediate results are processed this data trafficking could quickly increase. Just consider spatial and temporal filtering of a single image by construction of for instance a Gaussian and a Laplacian pyramid that each contain around 6MB (32 bit float) of data, suppose that around 5 intermediate results of data are necessary for temporal integration. It becomes obvious that computing data on a CPU might lead to trafficking times of more than 30 ms per image as just sketched. Some real time CPU models need to avoid data transportation for this reason by keeping data as much a possible in the memory caches, and thus in a more complicated software architecture. Section 4 elaborates on this topic in more detail.

An alternative would be to send data to the PCI-Express bus at speeds of 250 or 500 MB/s (PCI express version 2.0 of late 2007) per lane. Commonly 16 lanes are used for graphics cards resulting in speeds of up to 4 or 8 GB/s. The in practise due to overhead, these values are lower, see also http://en.wikipedia.org/wiki/PCI_Express. Data-transfer rates will be checked extensively in Sections 4.

Performing a data-transfer once over the PCI-Express bus, result in data to reside on the GPU. Being there computational power and memory bandwidth are a factor 10 higher, and thus a good alternative when it comes to processing data in real time.

Another advantage is that when data needs to be displayed on the screen binding of the data that resides on GPU will be sufficient. Hence no data transportation to the CPU by PCI-Express is necessary. On the other hand the resulting CPU data will need to be moved to the GPU for visualization anyway.

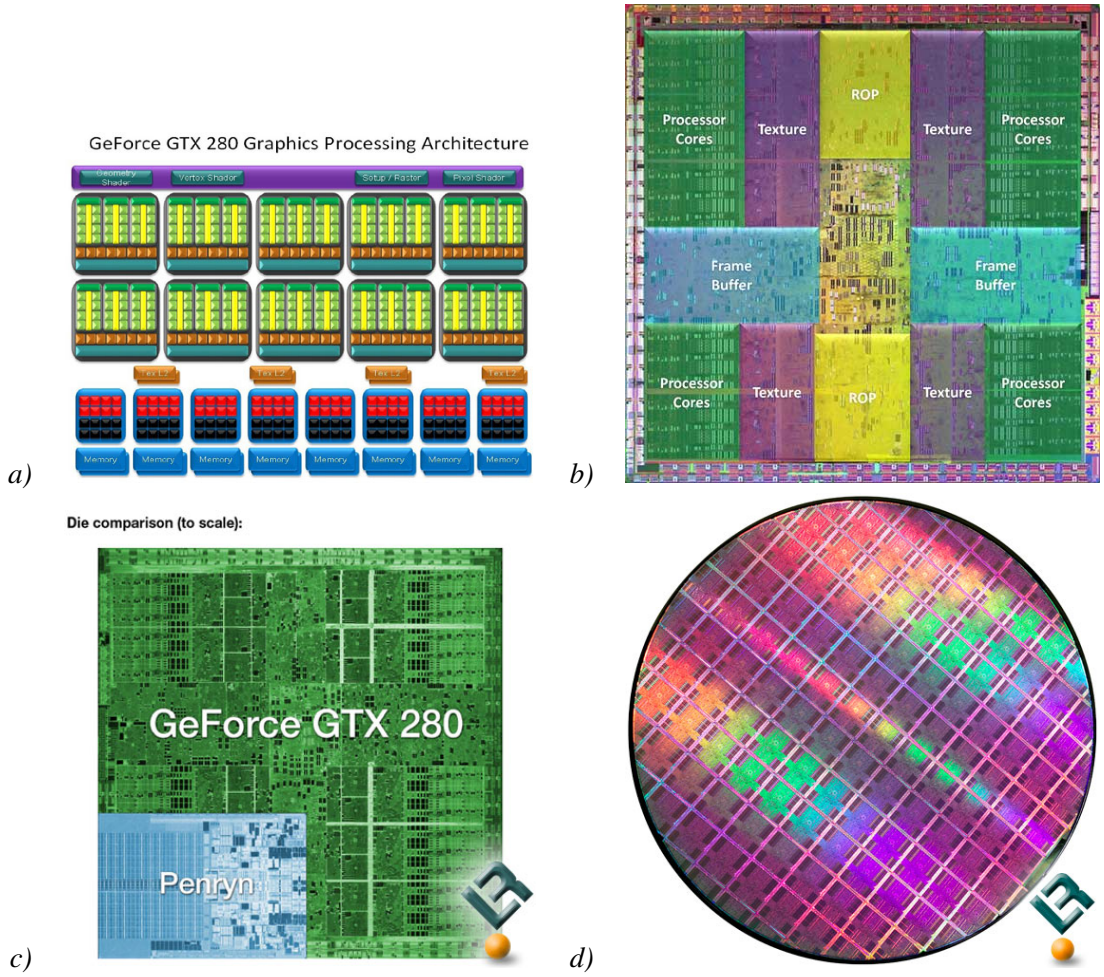

Figure 3: *a) Schematic details of NVIDIA 280GTX. b) Die overlay. c) Size compared to CPU. d) Die. Data obtained from LR web-site (http://forums.legitreviews.com).*

3 GPU Hardware

The first thing to be said about the latest NVIDIA GT200 series is that it's not a major departure from the current stable of G80-derived GPUs. Instead, it's very much a refinement of that architecture, with a multitude of tweaks throughout intended to improve throughput, efficiency, and the like. The GT200 adds a handful of new capabilities at the edges, but its core graphics functionality is very similar to current GeForce 8- and 9-series products.

The GeForce GTX 2xx series is designed specifically for parallel computing, incorporating unique features like shared memory, atomic operations and double precision support. In order to do this it has 240 cores running at speeds of 1.3GHz or higher.

If you take the die of the Intel 45nm Penryn processor that is found in the newest quad-core processors and sit it on top of the GeForce GTX 280 you'll get an idea of just how big this new core really is (Figure 3c). Eighty percent of the transistors of a GPU are used for computation compared to just 4% for a CPU!

As the overlay shows in Figure 3b, the stream processor clusters, texture units, raster operation processors (ROPs), and frame buffer memory partitions are located in a grid layout in the GPU. The wafer shot (Figure 3d) shows just how big the GT200 is.

Figure 3a shows a high-level view of the GeForce GTX 280 GPU parallel computing architecture. A hardware-based thread scheduler at the top manages scheduling threads across the thread processing clusters (TPCs). You'll also notice the compute mode includes texture caches and memory interface units. The texture caches are used to combine memory accesses for more efficient and higher bandwidth memory read/write operations. The elements indicated as 'atomic' refer to the ability to perform atomic read-modify-write operations to memory. Atomic access provides granular access to memory locations and facilitates parallel reductions and parallel data structure management.

The G80 core used to perform ROP frame buffer blending at half speed. The GTX 200 can perform the same tasks at full-speed. The GT200 GPU also sports twice the number of registers for longer, more complex shaders. The chip's output buffer size has been increased by a factor of six and it offers IEEE 754R compliant double precision for improved floating-point accuracy, its the first generation GPUs with 64 bit floating point precision.

Figure 3a provides a lot of information by the schematic rectangles. The 10 large groups across the upper portion of the diagram are what NVIDIA calls thread processing clusters, or TPCs. TPCs are familiar from G80, which has eight of them on board. The little green boxes inside of the TPCs are the chip's basic processing cores, known in NVIDIA's parlance as stream processors or SPs. The SPs are arranged in groups of eight, and these groups are called SMs, or streaming multiprocessors.

Let's combine the power of all three terms. 10 TPCs multiplied by three SMs times eight SPs works out to a total of 240 processing cores on the GT200, and that is nearly twice the G80's 128 SPs.

One of the key changes in the organization of the GT200 is the increase from two to three SMs inside of each thread processing cluster. The TPCs still house the chip's texture addressing and filtering hardware (brown rectangles), but the ratio of SPs to texturing units has increased by half, from 2:1 to 3:1.

The lower part of the diagram reveals a corresponding rise in pixel-pushing power with the increase in ROP (raster operator) partitions from six on the G80 (GeForce 8800 GTX) and four on the G92 (GeForce 9800 GTX) to eight on the GT200. Since each ROP partition can output four pixels at a time, the GT200 can output 32 pixels per clock. And since each ROP partition also hosts a 64-bit memory controller, the GT200's path to memory is an aggregated 512 bits wide.

4 Data transfer

In real time processing and latency issues data transfer plays a key role. In order to understand the performance of data transfer between processor and memory one has to understand the Intel architecture, its north bridge, and PCI-express (2.0).

The upper bound performance of the PCI-express bus is 8 giga bytes per second (GB/s). The memory bus, at the other side in Figure 2, is divided into 2 pipelines, all even cores have access to one while the odd core numbers have access to the other pipeline. A single core of an intel E5430 can write up to 3GB/s to memory and in that respect it does not fill the entire bandwidth of the pipeline. When a single process runs on all cores the maximum throughput per core is 1GB/s per core. In practise this number is around 800MB/s per core. Its obvious that when multiple processes run on a single core its throughput needs to be divided among the processes reducing the I/O bandwidth even more. The core i7 Nehalem technology provides a higher bandwidth its discussed in [2].

Assuming that every process is bound by memory I/O we can derive the maximum throughput per process:

$$T = \frac{B}{2 \max\left(\sum_{i=0}^{\frac{C}{2}-1} P_{2i}, \sum_{i=0}^{\frac{C}{2}-1} P_{2i+1}\right)}, \tag{1}$$

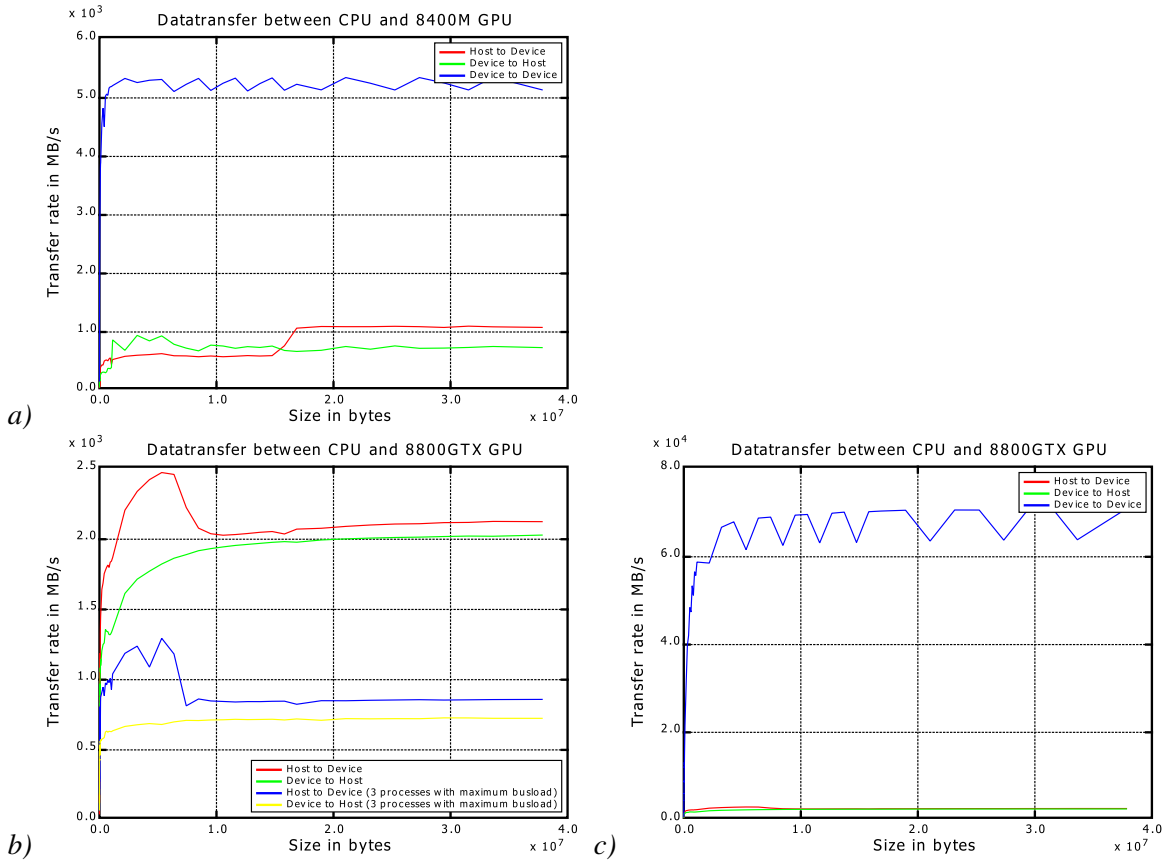

Figure 4: *Data transfer rates for different page sizes, ranging from 1 kilobyte to over 40 megabytes. a) Performance of 8400 mobile GPU device. b) and c) illustrate the performance of the 8800GTX GPU device.*

where B denotes the maximum bandwidth (8GB/s), P_i denotes the number of processes on core i, and C denotes the number of CPU cores.

We performed a test on the I/O bandwidth between CPU and GPU without any other processes and with cores 0, 1, 2, and 3 using maximum I/O bandwidth. These processes also use the full CPU processing power. The process that performs the bandwidth test for the GPU requires 100% CPU load for a single core as well.

Table 1 provides the transfer from host to device for page-able memory for 32MB of data is performed for paged and pinned memory Transfer rates are close to the expected performance T from (1). For instance, when the first 4 cores have a process that is solely writing data, and the process is running that does the check the GPU related bandwidths, we have either 3 processes on the even or odd cores. Hence it is expected that the CPU I/O performance declines with a factor 3. The second row in Table 1 illustrates that there is a decrease in performance of a factor 2.4 to 2.6.

When the same is performed with 8 cores instead of 4, we even notice a decrease in device to device data transfer. This is due to the maximum CPU load required by a single process. Since 9 processes are active computational resources are shared which has effect on the maximum possible GPU device to device transfer.

Figure 4 demonstrates the bandwidth for different memory chunks. The maximum bandwidth is achieved when having chunks that are 4 megabytes or larger. A performance of at least 50 percent is achieved at page sizes larger than 50 kilobytes.

	H to D	D to H	D to D	P	P H to D	P D to H
8800GTX paged 1 process	2122.5	2013.2	70426.7	1	1.0	1.0
8800GTX paged 5 processes	895.5	778.5	70334.9	3	2.4	2.6
8800GTX paged 6 processes	466.1	423.5	70043.3	4	4.6	4.8
8800GTX paged 9 processes	444.8	377.7	48788.4	5	4.8	5.3
8400M paged 1 process	1053.1	712.1	5315.7			
8800GTX pinned 1 process	2920.8	3122.1	70376.5			
8400M pinned 1 process	2130.9	501.1	5312.3			

Table 1: *Data transfer rates from host to device, device to host and device to device in megabytes per second (MB/s). P denotes the number of processes for all even cores, P H to D and P D to H denote the number of expected running processes on on all even cores, for host to device, and device to host data transfer, respectively.*

5 Introduction to CUDA programming

This section elaborates on the setup of the CUDA programming environment, its syntax, and some basic steps in CUDA programming.

5.1 Installation of CUDA

The following environment variables are used by CUDA:

```
CUDA_BIN_PATH ...\CUDA\bin
CUDA_INC_PATH ...\CUDA\include
CUDA_LIB_PATH ...\CUDA\lib
```

5.2 CUDA project

Setting up a CUDA project in Microsoft visual studio is straight forward using template vpcproj and sln files.

The main difference compared to a CPU only project is that the nvcc compiler will be invoked. A command line example looks as follows

```
$(CUDA_BIN_PATH)\nvcc.exe -ccbin $(VCInstallDir)bin; -deviceemu -c
 -DWIN32 -D_CONSOLE -D_MBCS -Xcompiler
  /EHsc,/W3,/nologo,/Wp64,/O2,/Zi,/MT
 -I $(CUDA_INC_PATH) -I someMore/include
 -o $(ConfigurationName)\cuprogram.obj cuprogram.cu
```

In a project cuda files are **required** to have a .cu file extension. When using a qmake project file the project is extended by using the following:

```
#--------------------------------------------------------
# Cuda Files
#--------------------------------------------------------
CUDA_SOURCES  = TVPcuda.cu
CUDA_SOURCES += TVPcudafilters.cu
#--------------------------------------------------------
# Cuda flags
```

```
# use -deviceemu to test the code by emulation by
# uncommenting the line below
#--------------------------------------------------------------
# CUDA_FLAGS   += -deviceemu
#--------------------------------------------------------------
# Cuda compiler
#--------------------------------------------------------------
win32 {
    cuda.output   = $$OBJECTS_DIR/${QMAKE_FILE_BASE}_cuda.obj
    cuda.commands = $(CUDA_BIN_PATH)/nvcc.exe -c $$CUDA_FLAGS \
                    -Xcompiler $$join(QMAKE_CXXFLAGS,",") \
                    $$join(INCLUDEPATH,'" -I "','-I "','"') \
                    ${QMAKE_FILE_NAME} -o ${QMAKE_FILE_OUT}
}
cuda.input   = CUDA_SOURCES
QMAKE_EXTRA_COMPILERS += cuda
```

In essence the project file defines what cuda files (with extension .cu) should be compiled by the nvcc compiler.

5.3 Parallelism

Parallelism in CUDA is obtained by spawning off threads massively, typically in the order of thousands. It is called thread batching. Thread batching is done by thread blocks and a grid of thread blocks. A thread block has fast shared memory, synchronization points in the kernel, and 2D or 3D array support: (x,y) is x + yDx or (x, y, z) which is x + yDx + zDxDy. A grid is 2D (x, y).

5.3.1 General purpose functions

The function is a void return, and parameters are bound to 256 bytes.

Function Quantifiers

- `__device__` callable on the GPU from the GPU
- `__global__` callable on the GPU from the CPU
- `__host__` callable on the CPU from the CPU

Variable Quantifiers

- `__device__` global memory on the GPU
- `__constant__` constant memory on the GPU
- `__shared__` shared per-block memory on the GPU

Execution configuration

```
__global__ void Function<<<Dg, Db, Ns, S>>>(int argument)
```

Dg (dim3) denotes the dimensions of the grid Dg.x * Dg.y denote the number of blocks used. The z-dimension is fixed to Dg.z = 1.

Db (dim3) denote the size of each block, and Db.x * Db.y * Db.z denote the number of threads per block.

Ns (size_t) gives the number of bytes in shared memory that is dynamically allocated per block. By default it is 0, its an optional parameter.

Stream parameter S (cudaStream_t), is by default set to 0, its an optional parameter also.

There are four important built-in variables:

- blockIdx (dim3) that provides the index in a block
- blckDim (dim3) that provides the dimensions of a block
- threadIdx (dim3) that provides the thread index in a block
- gridDim (dim3) that provides the dimensions of the grid

Parallelism in CUDA is performed by a language extension:

```
void Function<<<GridSize,BlockSize>>>(int argument)
```

Both grid an block is represented as a vector with three parameters x, y, and z. The maximum block size is 512, 512, 64, but the multiplication of x, y, and z should be at most 512 for the current generation GPUs. The grid size allowed is 65535, 65535, 1.

Calling a kernel is done as follows:

```
dim3 grid ((N+255)/256, 1, 1);
dim3 block (256, 1, 1);
vectorAdd<<<grid,block>>>(d_A,d_B,d_C,N);
```

The kernel is implemented as follows:

```
__global__ void vectorAdd(float* iA, float* iB, float* oC, int N)
{
  int idx = threadIdx.x + blockDim.x * blockId.x;
  if (idx < N)
    oC[idx] = iA[idx] + iB[idx];
}
```

The above algorithm will be discussed in ore detail in the following section.

5.4 Timers

In parallel programming the outcomes of an algorithm can be timed using the following code snippet:

```
  /*************************************************************
   * Setup and start timer
   *************************************************************/
  unsigned int timer;
  CUT_SAFE_CALL (cutCreateTimer (&timer));
  cutStartTimer (timer);

  //.... the code to be timed
```

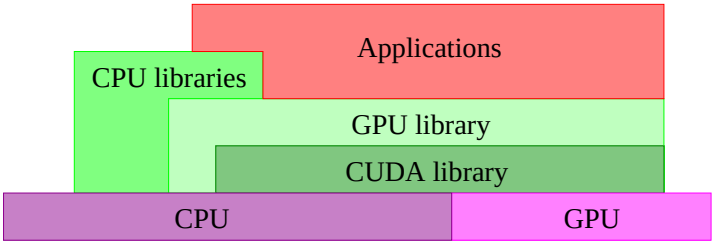

Figure 5: *Representation of architecture. The aim is to fully hide the CUDA architecture from the user and application developer.*

```
/*************************************************************
 * Provide timing results
 *************************************************************/
cutStopTimer(timer);
cout << "GPU time elapsed: " << cutGetTimerValue (timer)
     << " (ms)\n" << flush;
CUT_SAFE_CALL (cutDeleteTimer (timer));
```

In case this code is used in a CUDA file (.cu) the `cout` statements should be replaced by `printf` statements.

5.5 C alike

CUDA is C alike, but contains language extensions as shown earlier in this section. The most obvious difference are the triple smaller and larger signs.

CUDA allows templates making it as powerful as C++ in that aspect.

When programming CUDA use `malloc` and not `new`. Latter does not yield appropriate memory allocation.

For obvious reasons print statements cannot be used in a CUDA kernel. When this is desired for debugging purposes one should enable the emulation mode.

6 Software architecture

The software architecture is setup in such a way that it consists of a low level library, the so-called CUDA library where all interface routines have the “TVPcuda” prefix name, and a high level library called the GPU library all with “TVPgpu” prefix names, latter yields the GPU application programming interface (API).

The GPU library provides an interface to the standard C/C++ libraries and can be interfaced with these libraries without knowing anything about the underlying hardware. In case of replacement of the CUDA GPU concept, one would need to re-implement all modules available in the GPU library, and remove the CUDA library. The GPU library is a C++ library where normal classes and routines are constructed. Any c++ compiler can be used. The CUDA library contains cuda .cu and header .h files. This library exports the C-routines in the header files and uses the nvcc cuda compiler. Details on parameter settings and use of compiler can be found in Section 5.2.

Applications will have access to the GPU code by solely accessing the GPU library. An illustration is given in Figure 5.

6.1 Data structure

The key essence in image processing is an appropriate unified simple data structure `Data5D`:

```
Data5DTypes (int) datatype; // type of data used, e.g., int, float
int               w;        // dimension: width of data
int               h;        // height of data
int               d;        // depth of data
int               t;        // time of data
Data5DExtensions (int) e;   // extensions of data, e.g, MONO, COLOR
void              *data;    // data itself w*h*d*t*e elements
```

This data structure is used in many building blocks in rapid prototyping and programming environment TiViPE [1]. Data points either to a block of data on the CPU or GPU. Data is moved by using core GPU modules, see Section 6.2.

6.2 Basic components

The basic components for data handling on GPU (allocation and freeing) and transfer from GPU to both GPU and CPU and vice versa are provided by the following routines:

```
Data5D TVPgpuCopyToGPU (Data5D d)
Data5D TVPgpuCopyToCPU (Data5D d)
void   TVPgpuMoveToGPU (Data5D &d)
void   TVPgpuMoveToCPU (Data5D &d)
bool   TVPgpuAllocate (Data5D &d);
void   TVPgpuFree (Data5D d);
Data5D TVPgpuCopy (Data5D d);
```

Obtaining information from all GPU devices is obtained by using `cudaGetDeviceProperties` call, it is implemented as module `GPUinfo`, see Section 6.

In case multiple cards are used one can switch between these devices by setting a different id `cudaSetDevice (int deviceId)`, it is implemented as module `GPUsetDevice`, see Section 6.

6.3 Basic example

Lets assume a simple function, for instance, a binary threshold. A basic C implementation might be as follows:

```
Threshold (Data5D out, Data5D in, double threshold)
{
  // case in.datatype == DATA5D_FLOAT
  float *din  = (float *) in.data;
  float *dout = (float *) out.data;
  for (int i = 0; i < in.w * in.h; i++)
    if (din[i] < threshold))
      dout[i] = 0.0;
    else
      dout[i] = 1.0;
}
```

The concept of CUDA is to remove the above for loop by a parallel computational mechanism. Some languages provide a so-called "forall" statement or use a pragma statement before the loop. CUDA has an alternative mechanism that allows a user to strip, or tile the data, by defining a block and grid size. In the kernel standard variables (see Section 5.3.1) can be used to retrieve the index i in the array.

In the GPU library it could be implemented as given below. For the sake of completeness the threshold example contains all data transfers:

```
Data5D TVPgpuThreshold (Data5D in, double threshold)
{
  Data5D out = TVPSData5Dcopy (in);
  TVPgpuMoveToGPU (in);
  TVPgpuMoveToGPU (out);
  TVPcudaThreshold (out, in, threshold);
  TVPgpuMoveToCPU (out);
  return out;
}
```

In the CUDA library the assumption is made that data resides on the GPU, as given explicitly in the example above. The cuda routine is as follows

```
void TVPcudaThreshold (Data5D out, Data5D in, float threshold)
{
  // case in.datatype == DATA5D_FLOAT
  dim3 block (16, 16, 1);          // a tile of 16x16=256 elements
  dim3 grid (in.w/16, in.h/16, 1); // divide the image in tiles
  float *din  = (float *) in.data;
  float *dout = (float *) out.data;
  TVPcudaThresholdKernel<<<grid, block>>>(dout, din, in.w, threshold);
}
```

this routine is executed on the CPU entirely, its the kernel routine that is executed on the GPU, and is as follows:

```
__global__ void TVPcudaThresholdKernel (float *dout,
                                        float *din,
                                        int   width,
                                        float threshold)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x; // standard
  int y = blockIdx.y * blockDim.y + threadIdx.y; // cuda vars
  int i = y * width + x;

  if (din[i] < threshold)
    dout[i] = 0.0;
  else
    dout[i] = 1.0;
}
```

The code inside the loop in the CPU example is exactly the same as that in the kernel code of the GPU, making such code easy to port and understand.

6.4 General framework

Lets consider a desktop PC with a multi core CPU and one or more CUDA supported GPUs. Currently the number of cores in a CPU is 4 or 8. In practise only few processes run at a time on a single core. Hence we define that as coarse grain parallelism. The GPU has at most 240 processors available, where many threads are activated, typically in de order of 1,000 or more, we will define this a fine grained parallelism.

A parallel algorithm is an algorithm where several computations are carried on simultaneously, as opposed to a serial algorithm, where computations takes place sequentially. Considering the standard PC, a parallel algorithm is divided into two classes: coarse grain and fine grain parallel algorithms. We define algorithms that can be carried out simultaneously by at least 1,000 computations as fine grained, from 2 to 1,000 computations as coarse grained, and an algorithm that can be carried out by 1 computation simultaneously as a serial algorithm.

In the current architecture the central processing unit (CPU) is initiating all computations, hence we need to consider when fine grained parallelism can be used to compute an algorithm.

Lets consider a simple example $c_i = a_i + b_i$ for $i = 1024^2$, as given in Section 5.3.1. Its clear that this is a fine grain parallel algorithm, since every element of c can be processed independently. The CPU algorithm is as follows:

```
float *rdata = new float[d1.w * d1.h];
float *data1 = (float *) d1.data;
float *data2 = (float *) d2.data;
for (int i = 0; i < d1.w * d1.h; i++)
  rdata[i] = data1[i] + data2[i];
```

It takes 4,435 microseconds (μs) to execute it for a single step on a E5430 Intel xeon cpu at a speed of 2.66 GHz on a single core. The CPU has 8 cores so under ideal circumstances it can be processed in 554 μs.

The CUDA implementation is as follows

```
gd1 = tg.TVPgpuCopyToGPU (d1);    // 1,872 us (free inclusive)
gd2 = tg.TVPgpuCopyToGPU (d2);    // 3,053 us (free inclusive)
TVPcudaAdd (gd1, gd2);            //   220 us
ret = tg.TVPgpuCopyToCPU (gd1);   // 3,864 us for 1024^2 float
tg.TVPgpuFree (gd1);
tg.TVPgpuFree (gd2);
```

It takes 9,012 μs to execute the above algorithm for a single step. Obviously this algorithm performs transportation between CPU and GPU and vice versa. Uploading the data takes around 1,57 milliseconds (ms), and around 0.3 ms to free it, for 4 mega bytes (MB) of data, that corresponds to approximately 2.5 giga bytes per second (GB/s). Doubling the upload to 8 MB of data results in a slight decrease in data transfer. Solely, down loading 4MB of data after adding is done in 3.2 ms, yielding a down load speed of around 1.25 GB/s.

The timing issues for GPU and CPU up- and down load are considered important, but can be estimated easily by considering the maximum memory bandwidth of the PCI-express bus. Once the data is on the GPU data transfer is less an issue for the GeForce 8800GTX the bandwidth is 86.4 GB/s and thus far less than a millisecond. Even on the 8400GS it is still 6.4GB/s, hence a 12Mb image takes a little over 2 ms to be copied, on a low-end GeForce 8 series graphics card.

The aim, however is to measure the speed of the algorithm, not the transfer rate of data that is bound by PCI express and the front side bus (Intel) or hyper transport bus (AMD), since data transfer

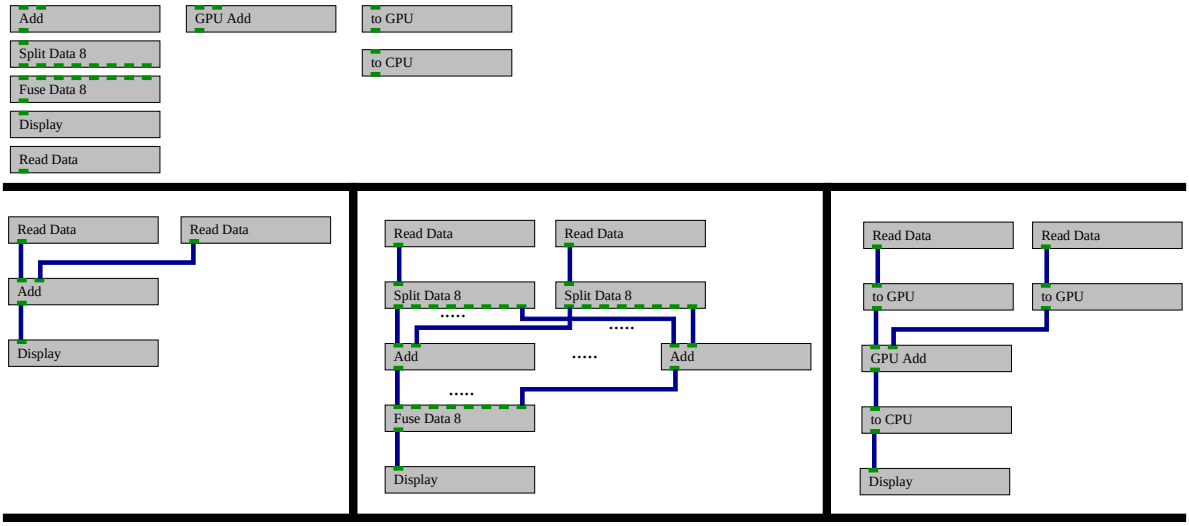

Figure 6: *Software framework. Computation of $c_i = a_i + b_i$ as a serial CPU algorithm (left), coarse grain parallel CPU algorithm (middle), and a fine grain parallel algorithm (right). Eight building blocks, given at the top, are required to construct these algorithms.*

is usually performed at the beginning and end of the algorithm. In Section 2 will be elaborated on the CPU and GPU architecture, where all bottlenecks for data transfer are discussed. Hence, we solely measure the computations needed for the algorithm The CUDA implementation is as follows

```
TVPcudaAdd (gd1, gd2);              //   220 us
```

This algorithm takes 220 μs an 8800GTX GPU, and thus the GPU is 20 times faster than a single core and at least 2.5 times faster for this simple algorithm when all 8 cores had been used. The current algorithm that is demonstrated is a worst-case[1] example on the GPU, from computational perspective already 2.5 times faster. *From algorithmic point of view its therefore appropriate to say that a fine grained parallel algorithm should be computed on the GPU.* Section 6.5 elaborates on the computational model of the GPU and will demonstrate in a very similar example the strength of the GPU, and why the current example is a worst case example.

The framework for dividing modules into a serial, coarse grain parallel, or fine grain parallel algorithms is graphically illustrated in Figure 6. A single block denotes a serial algorithm or a fine grained parallel algorithm. By representing an algorithm as a block in a graphical way its easy to understand which blocks can be executed in parallel. In the left part of Figure 6 the two ReadData blocks can be executed in parallel. This concept has been implemented in graphical programming environment TiViPE [1]. In case OpenMP is used for parallel CPU programming, a similar concept as given for the GPU blocks can be used.

6.5 Software Model

Under the assumption that data resides in GPU memory, can we make a prediction how much time it takes to compute an algorithm. The answer is yes, and it is done in the same way as for a serial computation. Compute the number of cycles that are required for the algorithm divide it by the number of processors and one could compute the time needed for the algorithm. It also holds for data transfer.

[1]This is a worst-case example because there is one floating point operation per data element. The rest is data transfer.

In this section we will demonstrate that the model is bound by the bandwidth and the number of instructions that can processed. Latter can be expressed in instructions per second or FLOPS.

The number of cycles used for pure computation is between 4 and 36. Part of the time is required transfer data from global GPU memory to the processing units, these instructions can be expressed in number of cycles. A stunning 400 to 600 cycles is required to get data from global memory!

The software model is therefore the sum of loads, stores, and instructions used to compute. In the part below we will give an example to demonstrate prediction and measured values for this model.

6.5.1 Example: polynomial function

Lets test this by computing a polynomial function:$y_i = \sum_{j=0}^{N} c_j * x_i^j$. In order to keep it simple we assume that $c_j = 1.0$. The cuda kernel code for this function is as follows:

```
__global__ void TVPcudaPolyKernel (float *ret,
                                   float *d,
                                   int   width,
                                   int   n)
{
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  int i = y * width + x;
  float xi = d[i];
  float yi = 1.0f;
  for (int p = n - 1; p >= 0; p--)
    yi += (1.0f + yi) * xi;   // 2 add and 1 mul
  ret[i] = yi;
}
```

The time needed to compute this polynomial function for n = 0, 1, and 50 is 135, 178, and 1518 μs, respectively for a 1024^2 32 bit floating point data set. In this algorithm 7 integer operations, 7 assignments, and 1 comparison are necessary for n = 0, and it requires 135 μs for the whole data set. It implies that 0 FLOPS have been computed up to now.

6.5.2 Memory bandwidth: Load and Store

The global memory bandwidth of the 8800GTX is 1.8GHz x 6 x 64 bits = 1.8GHz x 384 bits = 80.5 GB/s.[2] In the example given above there are one load (`float xi = d[i]`) and one store (`ret[i] = yi`) of 1024^2 32 bit floating point elements. The data transfer for 8 MB therefore requires at least $(8 \times 1024^2 \times 1000^2)/(1.810^9 \times 384/8) = 97\ \mu s$.

6.5.3 Computational performance: Instructions

The seven 32 bit integer operations take the same time as a 32 bit floating point operation. It has been validated by using the double add and multiplication statement `yi += 1 + 3 * 2` where `yi` has been made an integer.

Approximately 27 μs are required to compute statement `yi += (1.0f + yi) * xi` for the whole data set yielding $3 \times 1024^2 \times 1000^2/27$ = 116 GFLOPS. Modification of this statement to `yi`

[2] A giga byte is 1024^3 bytes. Sometimes the number 86.4 GB/s is given this holds when giga = 1000^3.

`+= 2.0f * xi + 1.0f` provides a timing of 1178 μs for n = 50 resulting in a performance of 154 GFLOPS.

The overhead of the loop requires 2 extra registers, an integer operation, and a comparison, if we would know a-priori the value of n, we could optimize the code a bit by unrolling the loop. By removing the loop it takes 149 and 776 μs to compute statement `yi += (1.0f + yi) * xi` for n = 1 and 50, respectively, and 160 and 465 μs for statement `yi += 2.0f * xi + 1.0f` for n = 1 and 50. Giving a performance of 246 and 505 GFLOPS, respectively. Latter is close to the theoretical peak performance of the 8800GTX that is obtained by (madd + mul = 2 + 1 flops) x 1350 MHz x 128 processors = 518.4 GFLOPS. In this example 3 operations are in a single instruction.

It would be appropriate to state that the 8800GTX is able to process 1.728×10^{11} instructions (of 4 cycles) per second. Based upon this number we can make a prediction how much time the first three statements take each: $1024^2 \times 1000^2 / 1.728 \times 10^{11} = 6.07\ \mu s$.

6.5.4 Model prediction

Making a computation of the kernel we need:

- 18 μs for first three statements
- 49 μs to load
- 6 μs for assignment of yi (4 cycles)
- 0 μs for the register assignment (o cycles)
- 6 μs for the comparison in the loop (4 cycles)
- 49 μs to store

making a total of 128 μs for a polynomial function of order 0. The total of 135 given earlier indicates that there are 7 μs overhead. The plot where the loop has been removed, see also Figure 7a, illustrates that 129 μs are necessary. Also here there are 7 μs overhead. The 7 missing μs could be due to the overhead of the two function calls, one from the GPU and one from the CUDA library. The same figure illustrates that statement `yi += 2.0f * xi + 1.0f` requires two instructions, based upon the timing of the the 50th and 0th order we know that the 50th order requires 100 instructions more per data element than the 0th order. The time required for a statement can be derived from the 0th and 50th order polynomial: (776 - 129) / 100 = 6.53 μs for 1024^2 elements, and that is close to the predicted number of 6.07 given in Section 6.5.3.

Figure 7a shows a beautiful linear behavior in computational time, i.e., for every increment in polynomial order, an additional 13 μs are necessary.

6.5.5 CPU implementation

The polynomial function on the CPU is implemented as follows:

```
for (int i = 0; i < Iw * Ih; i++)
  {
     float xi = Idata[i];
     float yi = 1.0f;

     yi += (1.0f + yi) * xi;    // 2 add 1 mul
```

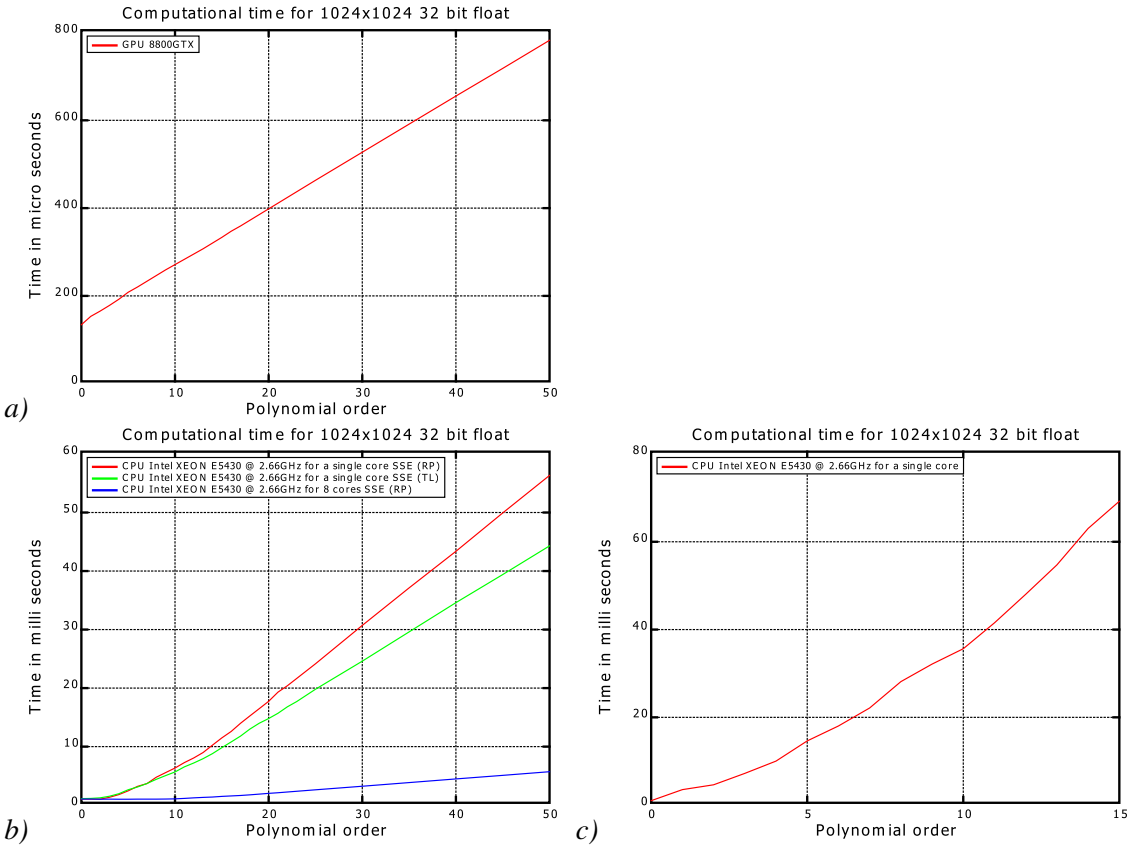

Figure 7: *CPU and GPU performance for computing a polynomial function of the n-th order. a) Performance on the GPU. The time given is in microseconds (μs). b) Performance on a CPU. Time is given in milli seconds (ms). c) Performance on a single core without SSE instructions. Both (b) and (c) demonstrate non-linear behavior that is caused due to in-lining.*

```
    yi += (1.0f + yi) * xi;    // 2 add 1 mul
    // ... for every increment in n add one line
    yi += (1.0f + yi) * xi;    // 2 add 1 mul

    Odata[i] = yi;
}
```

For a 0-th order polynomial function it takes 480 μs for a single core. In this case there is no input data and no floating point operations are performed. As soon as a floating point computation is required the time increases rapidly. Adding an instruction results in a non-linear behavior of the CPU due to code in-lining. Having an order above 10 yields a linear increment of of 7.15 ms in computational time for 1024^2 elements. An illustration is given in Figure 7b and c. In a model such non-linear behavior is harder to predict compared to the GPU.

Comparing performance between the CPU for a single core and GPU is given in Figure 7a and b. Higher order polynomial functions demonstrate a magnitude of an order 2. For instance, the GPU is more than 70 times faster for the polynomial function of order 50 than a single core CPU.

6.5.6 Best case GPU performance

Computation of statement `yi += (1.0f + yi) * xi` takes 13 μs and 7.15 ms on GPU and CPU respectively for 1024^2 data elements. Making the GPU around 550 times faster than a single core CPU. For the statement `yi += 1.0f + 2.0f * xi` it is approximately 1,100 times faster. This is the the best performance difference that can be obtained using normal non optimized C source code on both CPU and GPU.

When the code is optimized on the CPU using parallelism and SSE instructions:

```
const __m128 one = _mm_set_ps (1.0f, 1.0f, 1.0f, 1.0f);
static __m128 Odatasse[1024 * 1024 / 4]; // Iw = Ih = 1024
__m128 *Idatasse = (__m128 *) Idata;
#pragma omp parallel for // num_threads (1)
for (int i = 0; i < Iw * Ih / 4; i++)
  {
    __m128 xi = _mm_load_ps ((float*) (Idatasse + i));
    __m128 yi = one;
    yi = _mm_add_ps (yi, _mm_mul_ps (xi, _mm_add_ps (one, yi)));
    _mm_stream_ps ((float *) (Odatasse + i), yi);
  }
Odata = (float *) Odatasse;
```

a performance of 30 GFLOPS has been achieved, as illustrated in Figure 7c. The GPU yields a performance of over 200 GFLOPS, and that is almost 7 times faster compared to the CPU, see also Figure 7b and c.

6.5.7 Refining the model

Up to now we have touched the memory bandwidth and computation limits. These limits are no theoretical upper bounds since they can be reached as explained earlier in this section.

There are restrictions to the performance of this model:

- minimum number of elements
- number of registers
- memory alignment and memory banks

The data transfer depends on the size of the data as illustrated in Figure 4c and should be more than 50 kilobytes to achieve a good performance and 4 MB or above to get the maximum performance.

The same holds for the number of threads. Typically a setting for the number of threads in a routine is 64 (the minimum), 128, 192, 256, 384, or 512. Latter is the maximum in the current (8 and 9 series) CUDA architecture.

A grid of thread blocks is executed on the device by scheduling blocks for execution on the multiprocessors. Each multiprocessor processes batches of blocks one batch after the other. A block is processed by one multiprocessor only, so that the shared memory space resides in the on-chip shared memory leading to very fast memory accesses.

How many blocks each multiprocessor can process in one batch depends on how many registers per thread and how much shared memory per block are required for a given kernel since the multiprocessor's registers and shared memory are split among all the threads of the batch of blocks. If there are

Registers	≤ 10	11	12	13	14	15	16	17	18	19
Threads	256	128	128	256	256	256	256	64	64	128
Occupancy	100	83	83	67	67	67	67	58	58	50
Registers	20	21	22	23	24	25	26	27	28	29
Threads	128	128	64	64	64	64	256	256	256	256
Occupancy	50	50	42	42	42	42	33	33	33	33
Registers	30	31	32							
Threads	256	256	256							
Occupancy	33	33	33							

Table 2: *Comparison of number of registers, thread settings, and occupancy. Given the number of registers per kernel, the number of threads in a block need to be set to the number of threads given above to obtain the highest occupancy rate in percentage.*

not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

The 8800GTX is able to process 768 threads in parallel. For most of the routines the images are divided into tiles of 16x16 pixels making blocks of 256 threads. Hence, three of these tiles are computed in parallel on a multiprocessor. The 8800GTX has 16 multiprocessors, 4 in a 8600GT and 2 in a 8400M. All 8 series NVIDIA GPUs contain 8 stream processors per multiprocessor.[3]

The number of registers on the GPU are 8192 (8K), they are divided among these 768 threads. Hence, the number of registers per kernel should not exceed 10 (or 16 for the 200 series cards), otherwise the load drops, and part of the GPU will be idle. Lets consider a tile size of 16x16 pixels. In case of 11 to 16 registers only 2 tiles can be computed in parallel, and up to 32 registers only one at a time. Since there are 8192 registers and the minimum block size is 64 there should be at most 128 registers in a kernel.

Setting the appropriate tile size is needed when the number of registers exceeds 10. Table 2 gives the most appropriate tile size by number of threads for kernels with 11 or more registers. The number of registers that are used per routine are visualized using compiler flag `--ptxas-options=-v`.

A drop in load likely has little influence on the load and store of data, but has impact on the computational speed. Likely latter drops by the given occupancy rate. Note that the exact profile needs to be validated by an example, but will not be considered in this study.

6.5.8 Memory alignment

Since in most kernel computations loading and storing data is performed it is essential to consider what impact data loading might have on the performance. We will elaborate on the 2D convolution

$$f(x,y) = g * h = \int_{x'} \int_{y'} g(x',y')h(x-x',y-y')dx'dy' \tag{2}$$

and demonstrate the impact of misaligned data, and the steps needed to get data properly aligned. We will use different kernel sizes 1x1, 3x1, 1x3, 3x3, 5x5, 7x7, and 9x9. For all kernel elements we will take value 1, and assume that the input data has 1024^2 32 bit floating point elements.

The kernel is as follows

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
```

[3]The 280GTX and 260GTX from the 10 series have 10 and 8 multiprocessors, respectively, each with 24 stream processors.

```
int y = blockIdx.y * blockDim.y + threadIdx.y;
int i = y * w + x;
// assume up to 9x9 kernel boundaries
if ((x > 3) && (x < w - 4) && (y > 3) && (y < h - 4))
  {
    // O[i] = I[i];                    // 1x1 kernel
       O[i] = I[i-w] + I[i] + I[i+w]; // 1x3 kernel
    // O[i] = I[i-1] + I[i] + I[i+1]; // 3x1 kernel
  }
else
  O[i] = I[i];
```

The predicted time needed for this routine is

- 3x6 = 18 μs
- if evaluation 4x6 compare + 2x6 compute + 2x2 load + 3x6 binary = 58 μs
- load 49 μs
- store 49 μs
- if-then-else ?

The total time estimated is 174 μs, its measured time is 196 μs. An if-then-else statement can result in threads of the same warp to diverge, sometimes having impact on the speed.

The 1x3 kernel takes 268 μs to compute. It only took 72 μs and 24 are used to add and obtain the indices, that leaves 48 for the load and not the double amount of time. Obviously what the neighbor has been loading is not loaded again.

Computing the 3x1 kernel takes 1060 μs, 4 times the time of a 1x3 kernel. What has happened? We can analyze the situation by starting the CUDA visual profiler. Its detailed analysis tells us that there are many incoherent loads causing banking conflicts.

An alternative method is to bind texture memory to the input data The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance.

Its code is as follows

```
cudaBindTexture (0, texvec, I, w * h * sizeof (float));
TVPcudaSumKernelTF<<<grid, block>>>(outdata, w, h);
CUDA_SAFE_CALL (cudaUnbindTexture (texvec));
```

Variable `texvec` is a global variable that is used in the kernel, hence it is not passed to the routine as a parameter. The kernel calls for a 1x3 and a 3x1 kernel are as follows:

```
O[i] = tex1Dfetch (texvec, i-w) +
       tex1Dfetch (texvec, i) +
       tex1Dfetch (texvec, i+w); // 1x3
O[i] = tex1Dfetch (texvec, i-1) +
       tex1Dfetch (texvec, i) +
       tex1Dfetch (texvec, i+1); // 3x1
```

Approach from global to shared memory	Time	Registers per thread
Texture and threading	895	14
Threading	851	14
Texture and folding	513	6
Folding	473	8

Table 3: *Timing results in μs on a 8800GTX GPU for loading 20x20 floats from global memory into shared memory using 16x16 blocks, using threading and folding. Timings are performed for a 1024x1024 32 bit floating point image.*

Both kernels take 288 μs. In comparison to the 268 μs for the 1x3 kernel without texture binding, the overhead is small.

The time needed for loading a data element from global memory using texture is around 40 μs, and for a few elements this is the best solution, but in case of kernels with 81 elements it requires 5 ms, see also Figure 8, to compute an image.

If we consider that the neighboring picture contains 81 - 9 = 72 data elements that are the same, it might be a good solution to load all data to shared memory first.

6.5.9 Shared memory

Loading data into shared memory, requires some understanding of how data is divided. Lets consider that we have divided an image into tiles of 16x16 pixels, and assume that these pixels are processed in parallel using 16x16 blocks yielding 256 threads. For the convolution of a 9x9 kernel we need to acquire a tile of 24x24 = 576 pixels. In case we want to load all these elements into shared memory we would need 576 / 256 = 2.25 loads per thread. We will call this process *threading* since the data elements are loaded using the minimum number of loads per thread. Another approach would be to consider the shape of the 16 x 16 tile and to load the elements tile by tile. In this case 9 tiles need to be visited, but with the 9x9 kernel at most 4 loads are performed sequentially per thread, this can be verified by folding the 24x24 tile on a 16x16 block. We call this process *tiling* or *folding*.

In table 3 the results for a 1024x1024 floating point image are given, where 20x20 data elements are obtained from global memory and stored into shared memory using 16x16 blocks. Timing results in this table illustrate that the best way of loading data into shared memory is by using the folding technique. Also the number of registers used is less compared to the threading technique.

Figure 8 demonstrates the time required to load an nxm tile from global memory into shared memory. When folding of an nxn tile on a 16x16 block is considered, we expect time jumps at just above 16, 32, and 48. Since folding will increase from 1 reading (at 16x16), to 5 reading (at 18x18 to 32x32), and finally to 9 (at 34x34 to 48x48) readings per thread. The jumps occur at 18, 34, and surprisingly at 46.

For filtering by convolution one will at most compute kernels of the size 11x11 pixels which means a 26 x 26 sized tile. Beyond this size it is expected to be faster to compute the convolution by fast Fourier transform (FFT).

Figure 8 demonstrates the time required to load an nxm tile from global memory into shared memory. Figure 9 illustrates that the break even point for loading into shared memory first is at kernels with 8 elements. A snippet of the relevant CPU code

```
for (int y = 4; y < h - 4; y++)
  {
    int i = y * w + 4;
    for (int x = 4; x < w - 4; x++)
```

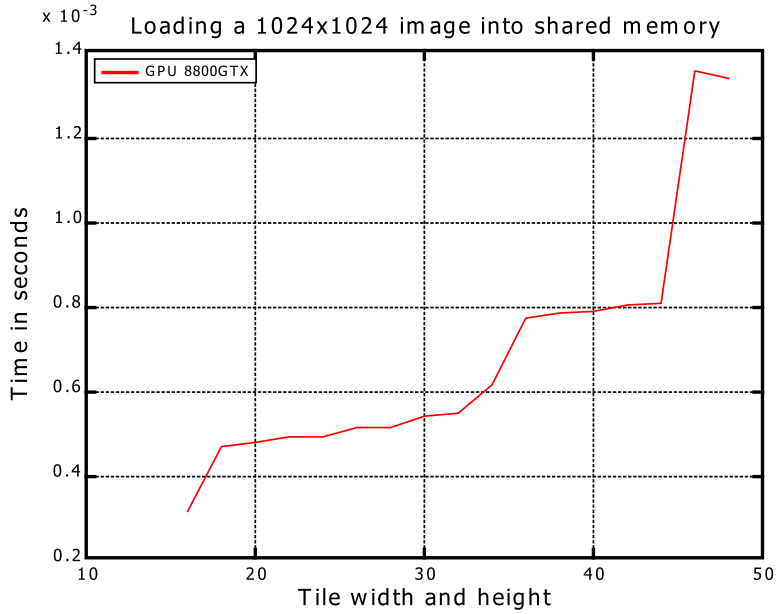

Figure 8: *Overall timing of a 1024x1024 float image with 16x16 blocks of reading an nxn tile from global memory into shared memory.*

```
    {
      // float di = 0.1f * data1[i]; // 1x1
      int j = i-1; // 3x3
      float di = 0.1f * data1[j-w];
      di += 0.1f * data1[j];
      di += 0.1f * data1[j+w];
      j++;
      di += 0.1f * data1[j-w];
      di += 0.1f * data1[j];
      di += 0.1f * data1[j+w];
      j++;
      di += 0.1f * data1[j-w];
      di += 0.1f * data1[j];
      di += 0.1f * data1[j+w];
      rdata[i] = di;
      i++;
    }
}
```

The GPU code after loading data into shared memory is very similar to the CPU given above. For the SSE instructions the Intel IPP library `ippiFilter32f_8u_C1R` has been used.

Figure 9b illustrates that the performance of the CPU is considerably slower 12 times for a 3x3 kernel and already 25 times for a 9x9 kernel. This comparison holds for Intel SSE optimized code, for normal CPU code as illustrated above there is a difference of 60 and 250 times for a 3x3 and 9x9 kernel, respectively.

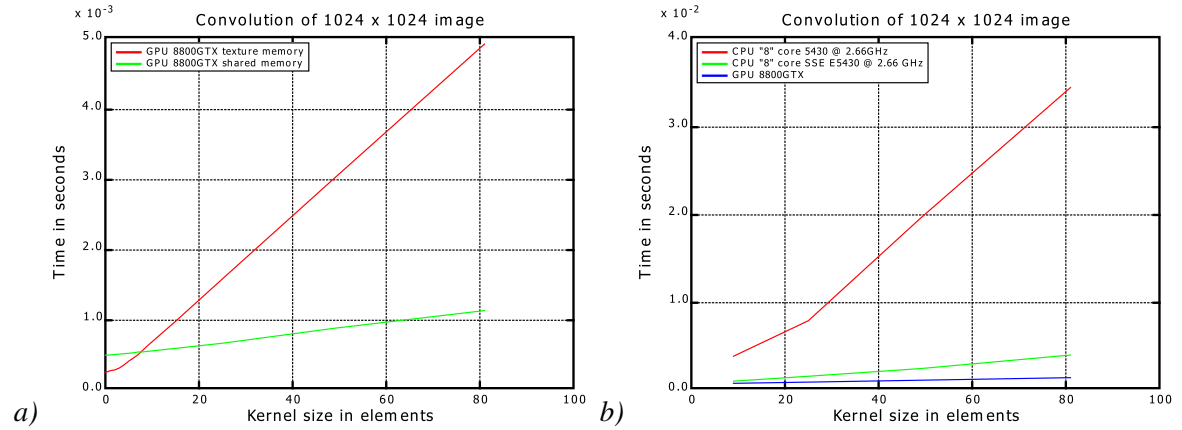

Figure 9: *Performance for convolution. a) GPU performance using different methods (reading solely form texture memory or first storing data into shared memory followed by computing the convolution using shared memory. b) CPU performance for standard implementation and SSE instructions using Intel's IPP library, both performed on a single core. Timings are performed on a single core, but divided by 8 to simulate a ideal octo (or dual quad) core CPU.*

Process	Time	Total time
Upload from CPU to GPU	2420	2420
Load to shared memory	310	2730
Computation of convolution	242	2972
Down load from GPU to CPU	2970	**5942**
Free GPU memory	910	6852

Table 4: *Timing results in microseconds (μs) on a 8800GTX GPU for a 1024x1024 32 bit floating point image.*

6.5.10 GPU Performance by decomposition

Since it takes 6.4 ms to compute a 3x3 kernel on a single CPU kernel, it is already advantageous to transfer data to GPU compute it there and move its result back on the GPU, it was completed in 6.28 ms. It is beneficial to compute 18 floating 32 bit floating point operations per pixel on a 1024x1024 sized image on a GPU! The decomposition of the GPU timings is illustrated in Table 4.

References

[1] T. Lourens. Tivipe –tino's visual programming environment. In *The 28th Annual International Computer Software & Applications Conference, IEEE COMPSAC 2004*, pages 10–15, 2004.

[2] T. Lourens. The art of parallel processing using general purpose graphical processing units –tivipe software developemnt. Technical report, TiViPE, 2009.