The Art of Parallel Processing using General Purpose Graphical Processing units

–TiViPE software development

Technical Report

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

September 14, 2010

Contents

Abstract

The aim of this report to elaborate TiViPE modules that make use of NVIDIA's compute unified device architecture (CUDA) programming. The focus will be on the construction of these programs making the best use of the GPU hardware using CUDA. These results are sometimes counter intuitive compared to traditional hardware, for instance 32 bit floating point arithmetic is faster than 8 or 16 bit integer arithmetic, and often faster than 32 bit integer arithmetic. In the past avoiding arithmetic by using look-up tables was faster than computing a function, on the GPU processing a polynomial function is considerably faster.

1 Introduction

The aim of TiViPE is to integrate different technologies in a seamless way using graphical icons [2]. Due to these icons the user does not need to have in depth knowledge of the underlying hardware architecture. For a better understanding of the software architecture and hardware, see [3].

This report is outlined as follows: Section 2 elaborates on the basic components. In Section 3 the concept for programming TiViPE modules that make use of the GPU will be described. Section 4 gives an overview of the available GPU modules together with their timings on a 285GTX GPU card.

2 Basic components

The basic set of components involves data allocation, freeing, and transportation.

2.1 Device information and selection

Obtaining information from all GPU devices is obtained by using `cudaGetDeviceProperties` call, it is implemented as module `GPUinfo`, for a graphical representation see the left side of Figure 1. This module gives the following output:

```
Number of devices 2
Device 0=GeForce GTX 285
* Global memory 1073414144
* Shared memory per block 16384
* Registers per block 16384
* Warp size 32
* Memory pitch 262144
* Maximum threads per block 512
* Maximum threads dimensions 512 512 64
* Maximum grid size 65535 65535 1
* Total constant memory 65536
* Major 1
* Minor 3
* Clock rate 1476000
* Texture alignment 256
* Number of multi processors 30
--------------------------------
Performance in GFLOPS 1062.72
--------------------------------
Device 1=GeForce GTX 285
* Global memory 1073479680
  ...
* Number of multi processors 30
--------------------------------
Performance in GFLOPS 1062.72
--------------------------------
```

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`. For the example given above, it implies that useful settings are 0 or 1.

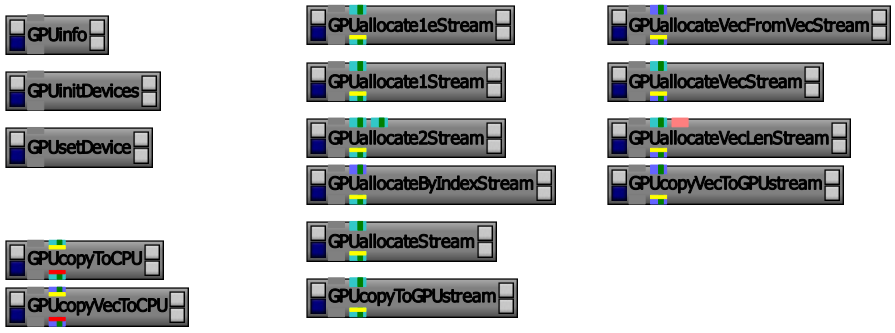

Figure 1: *Basic GPU modules.*

2.2 Data transportation

The basic components for data handling on GPU are data allocation and freeing, data transfer from GPU to both GPU and CPU and vice versa. The complete set is illustrated in Figure 1.

The speed of data transportation is an important aspect in real time systems, and has been discussed in [3]. An implementation of data transportation is given in Figure 2a and its timing results are given in Figure 2b-e.

In Figure 2a two green modules are merged to a single module called GPUmemorySpeedTestDev. Note that GPUsetDevice has only one output connection, this gray connection is the so-called control connection. Merging these modules is mandatory since the device setting would be lost if these modules would be executed sequentially by separate processes. Merging these modules to a single one preserves the setting of the device and by providing a device index a test can be run with one device or with two devices in parallel.

The timing results are obtained by using "ReadTable" twice together with the "Plot" module. The results of the plot module are given in Figure 2 and will be used throughout this report.

2.2.1 Data transport evaluation

Three different PCs have been used to evaluate the memory transfer

- A PC with PCI-express 2.0 (8 GB/s) and two 285GTX GPUs (159 GB/s; 1063 GFLOPS).
- A PC with PCI-express 1.0 (4 GB/s) and one 8800GTX GPU (86.4 GB/s; 518 GFLOPS).
- A notebook with PCI-express (4 GB/s) and one 8400M-GS GPU (6.4 GB/s; 38.4 GFLOPS).

Note that memory has an upper bound as well. For the Intel this was bound to 10.4 GB/s and 12.8 GB/s for high-end processors. Recently these numbers have doubled due to Intel's nehalem technology. Since multiple processors make use of the available bandwidth it might become a bottleneck, see [3].

The results given in Figure 2b-e have been obtained by taking the average over 10,000 iterations, and for large data blocks ($\geq$ 25 MB) 1,000 iterations.

Figure 2b illustrates the results for loading from CPU memory and storing it in GPU memory. The latest PC technology demonstrates that a 1GB/s data transfer is reached for even for small data sets. A throughput of 3 to 4 GB/s is reached at data sets of around 1 MByte. The maximum throughput

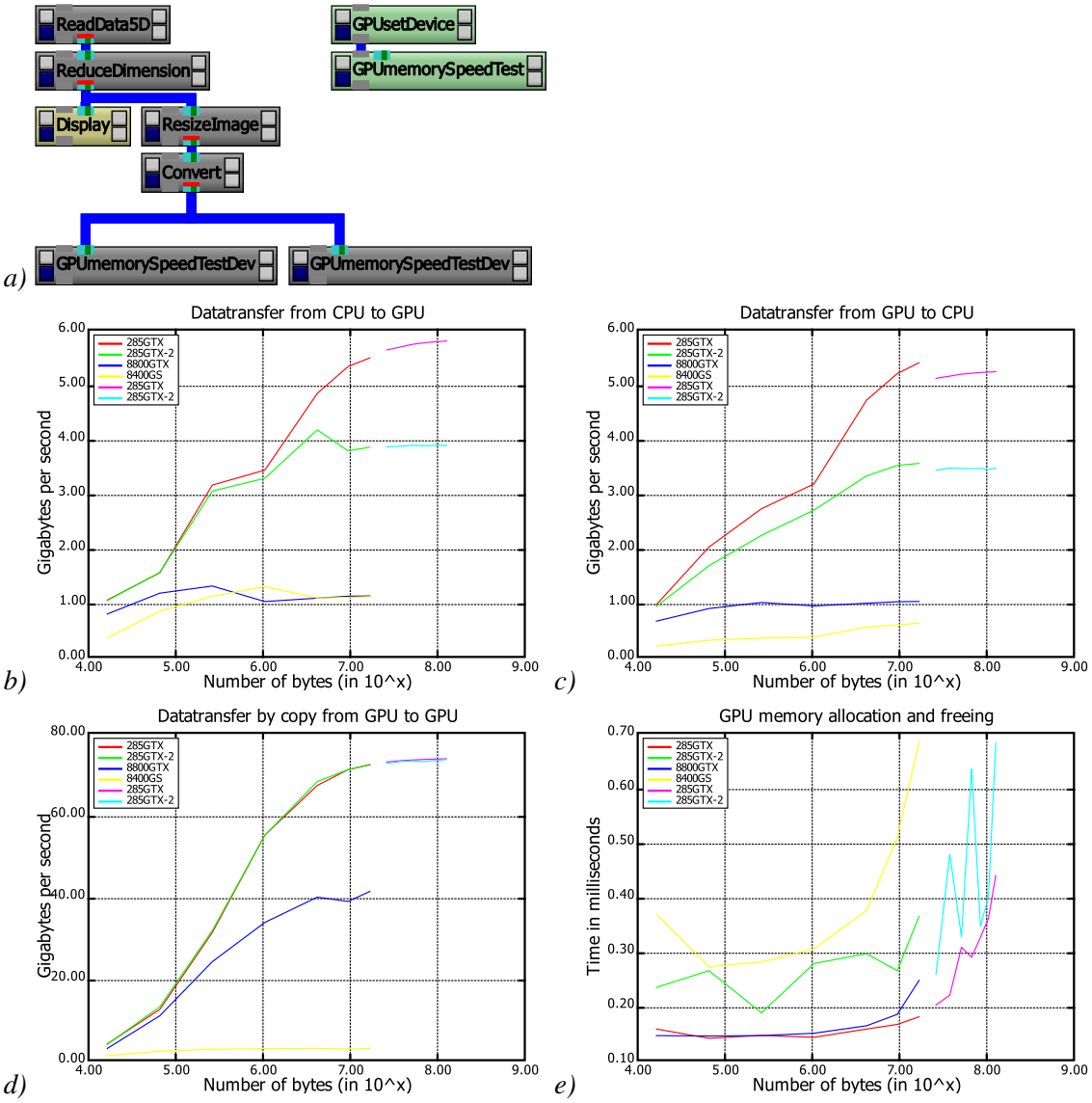

Figure 2: *Data transportation. a) TiViPE implementation. b-e) Timing results for data transfer from CPU to GPU, from GPU to CPU, by copy on the GPU, and for allocating and freeing data on the GPU.*

measured 5.8 GB/s and that implies 72.5% which is close to the maximum bandwidth, since there is 20 percent overhead on PCI-express.

Figure 2c illustrates data trafficking the other way around. Also here a bandwidth of 1GB/s is reached for small data sets. The performance "dip" for large datasets is due to the number of iterations instead of 10,000, 1,000 iterations have been performed. For the transportation of data from single 285GTX it resulted in a throughput between 5.1 to 5.2 GB/s, while repeating the same with 10,000 iteration resulted in a performance between 5.6 and 5.8 GB/s. Indicating that storing data in CPU memory is dependent on the amount of data rather than the data size alone.

Figure 2d demonstrates smooth curves reaching a throughput of up to 95% of the bandwidth. The 285GTX is capable of copying an 128x128 pixel 8 bit monochrome image at a speed of 1 GB/s it is less than 16 (μs). Note that in this case the used bandwidth is 2 GB/s, since on load and one store is used for a copy. Its clear that better performance is reached when 1 MB of data or more are copied.

Figure 2e memory allocation and freeing demonstrates that memory allocation and freeing takes constant time for a 285GTX GPU for data blocks up to 10 MB. Both are combined for the simple reason that allocated memory will be freed, unless memory leaks are desired. It takes 150 μs to allocate and free a single block, it seems very little, but it might have substantial impact impact in a real time environment.

Suppose that 8 computational steps are performed subsequently, and that every step requires allocation and freeing of one block, it would require 1 ms for data management only. In practise for every step multiple blocks are allocated quickly resulting in substantial memory management overhead.

It appears that the CPU is capable of allocating and freeing memory considerably faster, in object oriented programming these steps are performed many times, its commonly performed by a the constructor and destructor of a class.

TiViPE blocks have adopted the CPU concept of allocation and freeing. Unfortunately as mentioned before this might have strong impact on the processing speed of the GPU. For the GPU memory needs to be allocated first, next processing should take place where no memory management takes place, afterwards the memory will be freed. It explains why there are a substantial number modules for data transportation, see also Figure 1.

2.2.2 Parallel but not too parallel

A simple test has been performed by running two GPUmemorySpeedTestDev modules at the same time, both transferring 4 megabytes of data. The results of a single process are given in the first row of Table 1. It illustrates that a single process transports data of a rate of almost 5GB/s over a 16 lane PCI-Express v2.0 bus. A copy of data is performed on the GPU that requires one load and one store. For the bandwidth occupancy we have to double the rate since a load and a store is used. Taking into consideration that the maximum bandwidth of a 285GTX GPU is bound by 159 GB/s, the measured performance is around 90% of this maximum.

In case of receiving the maximum bandwidth the conclusion is that at most 2 processes should be used when using transferring data from CPU to GPU and vice versa, when two GPU cards are used.

2.2.3 Transfer by a single step

The data rate of 10,000 iterations was higher than 1,000 iterations. What happens if a single block is processed?

Figure 3 provides minimum, average, and maximum of 1,000 measurements for the PC with two 285GTX GPUs.

processes	device	toGPU(GB/s)	onGPU(GB/s)	toCPU(GB/s)	A+F(ms)
1	1x0	4.83163	67.1675	4.70536	0.1571
2	**1x0, 1x1**	**4.16156**	**68.1017**	**3.31804**	**0.2954**
2	2x0	1.26783	36.4422	1.23471	0.3227
4	2x0, 2x1	1.23307	35.4598	1.20343	0.5794
6	3x0, 3x1	0.82091	22.2305	0.79770	1.5313

Table 1: *Data transfer rates with one or two GPU cards activated. The number of processes is given in the leftmost column, its distribution on the 2 GPU devices in the next column. A+F denote allocation and deleting/freeing data.*

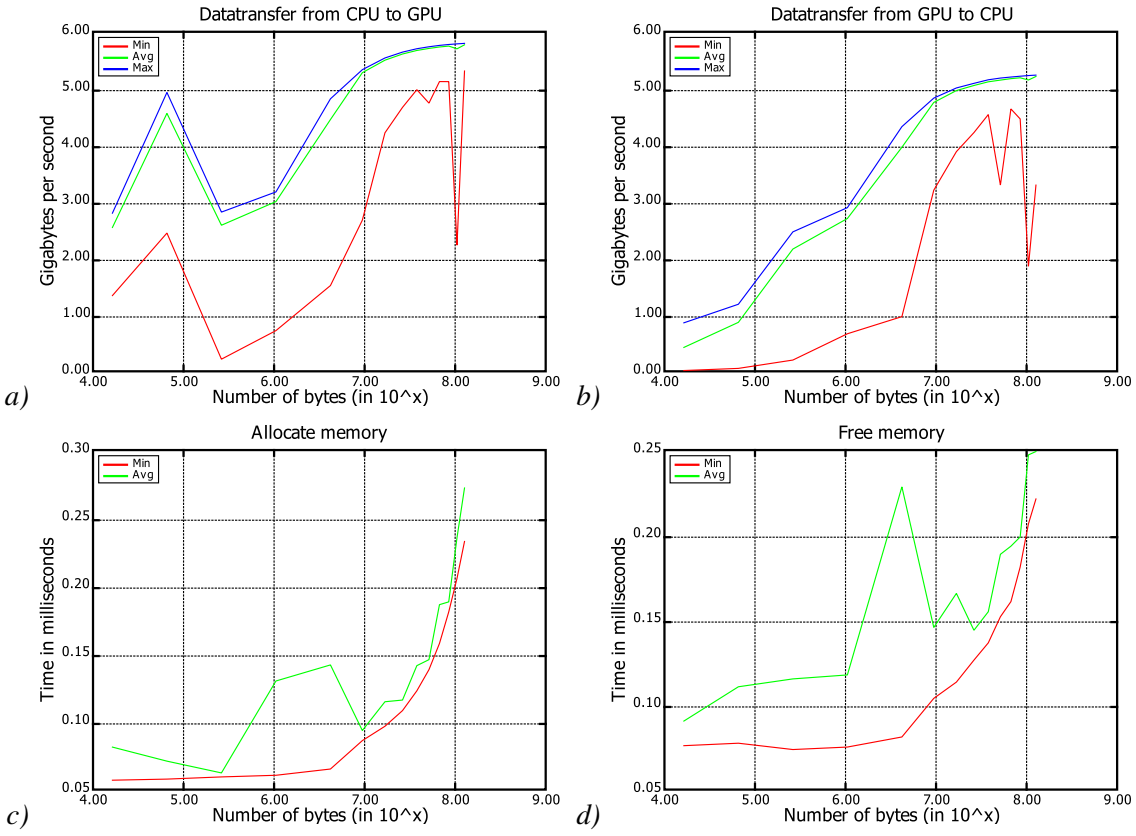

Figure 3: *Data transportation measurements for a single block of data. a-d) Timing results for data transfer from CPU to GPU, from GPU to CPU, GPU allocation of memory, and freeing allocated GPU memory, respectively.*

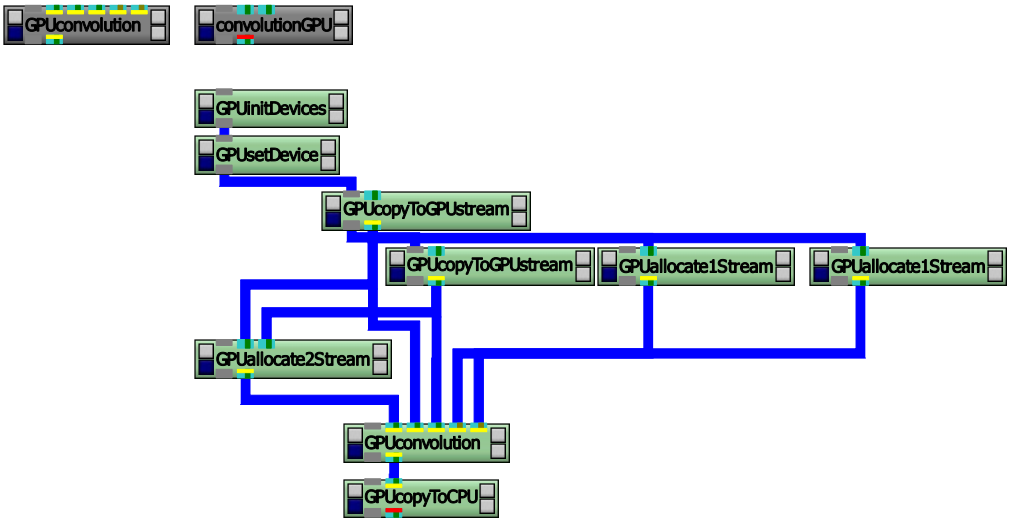

Figure 4: *TiViPE icon for convolution. Icons in green are use to construct CPU icon.*

The drawback of single block measurements is that other processes might be scheduled and take 10 or 20 ms of processing time, these are the worst case situations for allocation and freeing of memory that have been measured, but omitted in Figure 3c-d. These occurrences are few since average and maximum data transfer rates differ marginally.

Surprisingly the transfer rates for a single block from CPU to GPU are higher (Figure 3a) compared to those with 10,000 cycles in a run (Figure 2b). The transfer the other way around confirms that the maximum throughput is around 5.2 GB/s and further the curves given in Figure 3a and Figure 2c appear, as expected, very similar. Figure 3c and d confirm that memory allocation takes at least 60 and 70 μs for allocation and freeing memory. Figure 2e provides the sum allocation and freeing shows a similar curve.

3 TiViPE application by convolution example

The concept for programming TiViPE modules that make use of the GPU is decomposed in memory allocation and transfer to GPU, processing on the GPU, and transfer to CPU, and freeing memory. This concept is illustrated by the green modules in Figure 4.

The computational component is the “GPUconvolution”, for a definition and details on the implementation of a convolution, see [3], it contains 5 inputs rather than the expected 2, one for data and one for kernel(s), as given in “convolutionGPU”. This is because the GPU modules do not contain any data allocation or freeing. The GPUconvolution module contains five inputs from left to right, the pre-allocated output data, the input data, kernel data, image buffer, and kernel buffer, respectively. Latter 2 inputs are used solely if the fast Fourier transform (FFT) is used.

The timings for spatial convolution of the GPUconvolution module are listed in Table 2. For a 1024 squared image or kernel the Fourier transform for both forward and backward transform take around 1.4 ms while the multiplication in Fourier domain takes 0.2 ms. Depending on if the image and kernel need to be transferred to the Fourier domain the overall time needed is 1.4 (forward image) + 1.4 (forward kernel) + 0.2 (complex multiplication) + 1.4 (inverse transform) = 4.4 ms.

kernel size	Float	Char	Short	Integer	Long	Double
3x3	362					
5x5	470	868	674	713	1983	938
25 generic	1544	1424	1221	1789	3169	2330
7x7	647					
9x9	848					
11x11	1122					
13x13	1412					
15x15	1767					
17x17	2352					
19x19	2834					
21x21	3279					
23x23	3800					
25x25	5058					
625 generic	12700					

Table 2: *Spatial convolution timings on 285GTX for data size 1024x1024 for different kernel sizes. Timings in microseconds (μs).*

4 TiViPE modules

This section provides an overview of the available GPU modules their timings are provided for a 285 GTX GPU card at normal clock speeds. The initial set of available TiViPE modules that perform their main computations on the GPU are illustrated in Figure 5.

4.1 Processing time prediction for any GPU card

In case a different GPU is used or different clock speeds one can make an estimation of the time needed for a routine. Its as follows:

$$speedfactor = \frac{1}{2}\left(\frac{GPU_{Gflops}}{1063} + \frac{GPU_{GBsBandwidth}}{159}\right) \tag{1}$$

For instance if the card is an 8800GTX it has a $GPU_{Gflops} = 518$ and $GPU_{GBsBandwidth} = 86.4$. So its computational power is 0.48 and its bandwidth is 0.54 yielding a $speedfactor = 0.52$. Hence it is expected that it takes about 1.92 times the time compared to a 285GTX.

Note that this factor holds for floats only. For instance resizing other data type is considerably slower on a 8800GTX, see also Table 3.

4.2 Processing time

Processing time for 1024 times 1024 elements are given in Table 3.

Resizing data is size dependent, for images of 1024x1024 pixels of type float resizing is illustrated in Table 4 in the top row are the resizing factors given for first and second dimension while at the bottom row the timings are given.

Computing the histogram for all intensities

$$h(I) = \sum_{x,y} I(x,y) \tag{2}$$

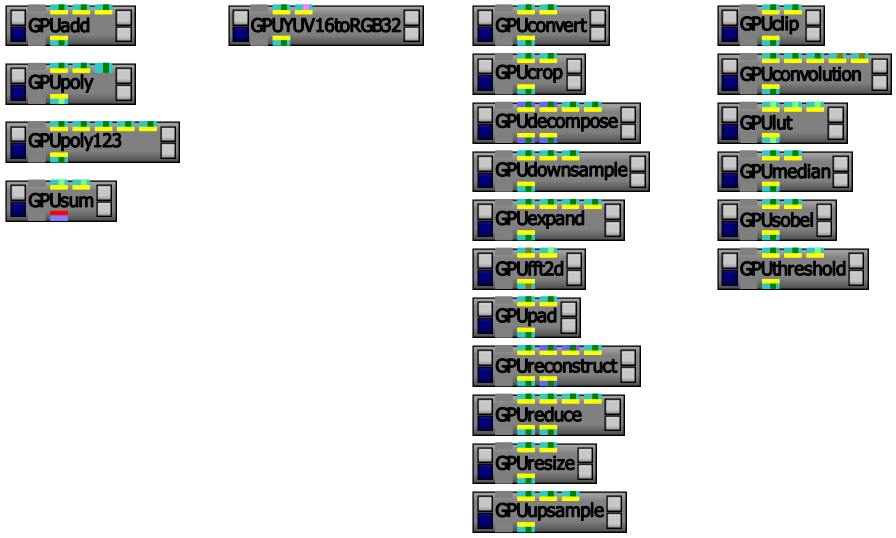

Figure 5: *Set of available GPU modules. Columns from left to right represent arithmetic, color, datamanipulation, and filters, respectively.*

class	module	description	Float	(U)char	(U)short	(U)Integer	(U)long	Double
Arithm.	poly123	float const	141	199	129	152	210	258
Arithm.	poly123	double const	314	288	285	306	388	372
Datam.	convert	8,16, 32 bit	79	69	69	79	128	128
Datam.	convert	64 bit int	118	103	98	99	126	126
Datam.	convert	double	118	167	171	170	170	128
Datam.	convert nc	8,16, 32 bit	101	109	111	111	133	136
Datam.	convert nc	64 bit int	138	144	140	140	153	156
Datam.	convert nc	double	137	328	326	325	320	153
Datam.	crop	from 2048^2	91	85	90	92	128	127
Datam.	fft2d		1367	X	X	X	X	X
Datam.	pad	from 800x600	101	97	100	102	130	130
Datam.	resize	to 2048^2	259	250	260	262	X	X
Datam.	resize	**on 8800gtx**	588	2861	2604	585	X	X
Filters	convolution	5x5 kernel	470	868	674	713	1983	938
Filters	convolution	25 generic	1544	1424	1221	1789	3169	2330
Filters	median	3x3 kernel	342	505	457	343	775	867
Filters	median	5x5 kernel	925	1410	1274	925	2516	3171
Filters	sobel		263	345	309	272	615	400
Filters	threshold	adap binary	177	180	172	179	227	219
Filters	threshold	fixed nobin	119	118	118	117	149	129

Table 3: *Timing in μs for 285GTX gpu card. Timings marked by X denote that they are not supported.*

0.25x0.25	0.5x0.5	0.75x0.75	1x1	2x1	2x2	3x2	3x3	4x3	4x4
40	60	66	80	143	259	388	560	744	984

Table 4: *Timings in (μs) for resizing a 1024x1024 image (float).*

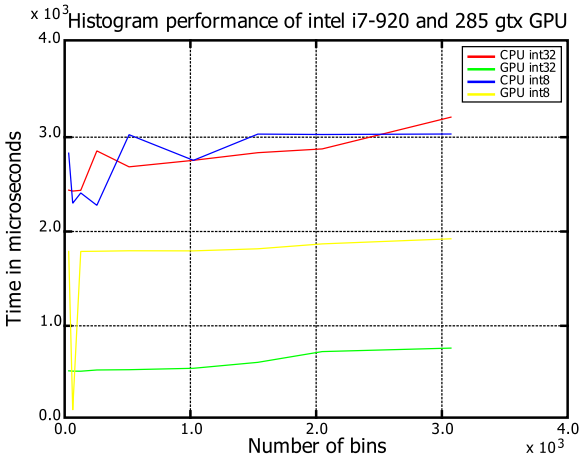

Figure 6: *Histogram performance for both CPU and GPU.*

where I(x, y) denotes the intensity of the image, can be performed in parallel on a GPU, but its performance isn't substantially better compared to a single core of a CPU, as illustrated in Figure 6. The only exception is the strongly optimized 32 bins solution for unsigned characters. In that case the histogram takes 80 micro seconds for a 1024x1024 monochrome image.

4.3 Visualization

TiViPE contains two standard icon called Display and Animate for visualization. The make use of OpenGL and CPU processing. The aim of the construction of the GPUdisplay and GPUanimate modules is to provide a very similar look and feel compared to their CPU equivalents. These routines use OpenGL as well, but process the data on the GPU using the concept of cudaGLMapBufferObject, processing on the GPU, followed by cudaGLUnmapBufferObject.

Table 5 illustrates the speeds of the old version compared to the new GPU version. For a single GPU device/card the performance is around 120 and 360 μs for the 285GTX and 8800GTX, respectively. The performance increase of a GPU is of two orders of magnitude compared to a CPU.

The processing itself takes barely any extra time, hence it does not matter what data-type is being used. Internally we can select to compute either unsigned character data or floating point computation for visualization. In the latter case, it implies that internally 4 times more data is used. Although it is possible technically to process floating point data, there are 2 reasons that it is not extremely useful:

1. Software and/or hardware provide 8 bit or 256 grades per channel (r, g, b, alpha).

2. In case your display is able to process 10, 12, or 16 bits per channel, and your software supports it (like these 2 TiViPE modules) the human eye is not sensitive enough to distinguish between two neighboring gray values or colors. So additional processing is needed to make these differences visible to the human eye.

4.3.1 Multiple GPU cards

The more interesting case is when at least 2 GPU cards are used, since in that case most of the processing time is spend in mapping and unmapping a buffer object. Its performance appears to be much

type	cpu-u	285gtx	8800gtx	2 x 285gtx-u	2 x 285gtx-f
uchar	9.6	0.11	0.33	1.1	3.1
uchar3	13.1	0.11	0.33	2.3	7.5
float	19.9	0.11	0.33	1.1	3.1
float3	36.0	0.11	0.33	2.3	7.5
uchar		0.12	0.36	3.1	9.5
uchar3		0.12	0.36	7.5	29.5
float		0.12	0.36	3.1	9.5
float3		0.12	0.36	7.5	29.5

Table 5: *Timings in milliseconds displaying a 1024x1024 and a 2048x2048 image, at top rows and bottom rows, respectively.*

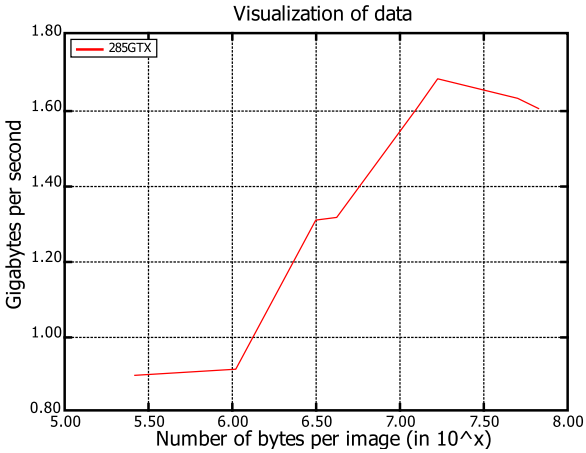

Figure 7: *Speed of visualization measured in GB/s.*

slower as shown in the rightmost two columns of Table 7. The throughput measured in GB/s is given in Figure 7.

In conclusion a one mega pixel monochrome image can be visualized in a little more than one millisecond. Setting up 2 display modules has a negative impact on the performance since it takes every display 4.1 milliseconds to visualize the data while a single module would take 1.1 milliseconds.

It appears that there is a correlation between the CPU-GPU transfer and the visualization timings (see also Figure 2 and Figure 7), its possible that there is just a factor 3 between them, possibly suggesting that 3 data transfers take place. One might suspect that cudaGLMapBufferObject copies the device memory to the host and then re-uploads it into OpenGL after modification. Possibly that copying data back and forth between GPU and CPU is faster using glMapBuffer to get data into an OpenGL buffer object.

On the NVIDIA forum, Mark Harris wrote the following on July 28, 2008:

> *to add a note here, because I got bitten by this recently. Currently on Windows XP OpenGL interop falls back to transferring data between GL and CUDA contexts via the host on multi-GPU systems. EVEN IF the CUDA and GL code are running on THE SAME GPU.*
>
> *Yes, this is a major limitation, and it will be fixed in the future. For now, you'll get better performance for interop-bound applications by running them on a single GPU.*

internal	elements x 1024^2	285gtx device 0	285gtx device 1
uchar	1	0.11	1.0
uchar	3	0.11	2.0
float	1	0.11	2.1
float	3	0.11	5.7
uchar	4	0.12	2.6
uchar	12	0.12	7.2
float	4	0.12	7.1
float	12	0.12	21.5

Table 6: *Timings in milliseconds displaying a 1024x1024 and a 2048x2048 image, at top rows and bottom rows, respectively.*

> *BTW, this means you need to remove the second GPU or disable it in the windows device manager. Simply not extending the desktop onto the second GPU is not enough (in my tests).*

Indeed Mark is right about the desktop extension, but fortunately there is a work-around to get the optimal performance of a single card for visualization, but being able to use multiple cards:

1. disable all cards except the first one. For window XP select `Control Panel->System`. A pop-up window appears Select `hardware->Device Manager`, select `Display adapters->Select` a device and disable it by a right-click and selecting disable.
2. A reboot is required to active these new settings.
3. Repeat the first item, but now enable the devices. The OS does not require a reboot! This setting yields best of both worlds.
4. To ensure that the same setting is obtained the next time, one could disable the device immediately, and reboot later.

These settings have been tested by using two displays that are connected to the first device.

Note that within TiViPE one can validate the number of active cards by running GPUinfo. The good part about the current setting is that multiple GPUdisplay modules can be used, they appear to work much faster for the first device that contains the desktop compared to the second device that is not connected at all to a display. Its obvious that in the latter case the data is transferred from GPU device 1 to CPU, and from CPU to GPU device 0. Considering the case given at the third column of the bottom row of Table 6 it takes 21.5 milliseconds to transfer the data from device 1 to 0, yielding a throughput of 2.23 GB/s. This corresponds to the expected bandwidth of 4.5 GB/s, see also Figure 3.

The fact that it takes 29.5 ms with one display connected to the first and one to the second, see bottom row of Table 5, and 37.4 ms (independently on which GPU device it is executed) when both connected to the first suggests that data is transfered first to CPU and then copied to both devices.

4.4 Multiple scale image processing

This section contains the modules for multiple scale processing with the aim to be able to perform a Gaussian pyramid decomposition to resize an image. Such a construction contain a high-pass and a low-pass image. The image reduction process involves lowpass filtering and downsampling the image pixels. The image expansion process involves upsampling the image pixels and lowpass filtering. This so-called Laplacian Pyramid has been described by Burt and Adelson [1].

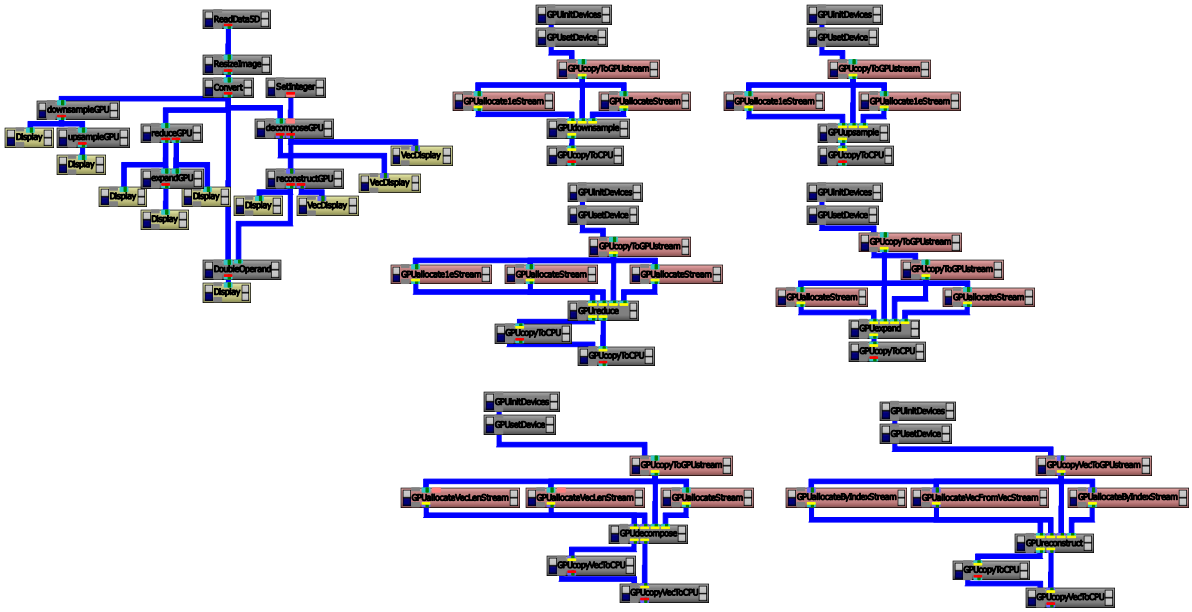

Figure 8: *Modules used for multiple scale processing. On the left an example with all 6 modules that are involved in pyramid processing. The other 6 networks represent each one module, where for every module memory allocation/freeing and transportation has been performed. They have been compounded to a single module with post-fix "GPU" rather than the pre-fixed "GPU" in the network on the left.*

The routines involved in pyramid processing are illustrated in Figure 8.

The routine used for downsampling is as follows

$$g_l(i,j) = \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n) g_{l-1}(2i+m, 2j+n) \tag{3}$$

for levels $0 < l < N$. Kernel w is a 5-by-5 pattern that is separable

$$w(m,n) = \hat{w}(m)\hat{w}(n) \tag{4}$$

and symmetric, and needs to give equal contribution. This is satisfied when $\hat{w}(0) = a$, $\hat{w}(-1) = \hat{w}(1) = 1/4$, and $\hat{w}(-2) = \hat{w}(2) = 1/4 - a/2$.

The routine for upsampling is as follows

$$g_{l,n}(i,j) = 4 \sum_{m=-2}^{2} \sum_{n=-2}^{2} w(m,n) g_{l,n-1}(\frac{i-m}{2}, \frac{j-n}{2}) \tag{5}$$

For the terms for which $(i-m)/2$ and $(j-n)/2$ are integers are included into this sum.

The timings for pyramid routines are provided in Table 7.

The two main routines for pyramid construction are GPUdecompose and GPUreconstruct. GPUdecompose constructs a Gaussian and a Laplacian pyramid from a given input image. GPUreconstruct on the other hand uses the Laplacian pyramid as input and constructs a Gaussian pyramid, level 0 of this pyramid is the same as the input image. In the experiment provided in Figure 8 this is almost the case, there are some rounding errors, likely caused by numerical loss on the 32-bit floating point. The largest error was found to be 0.0015 where the values are between 0 and 16,000.

Routine	4096	3072	2048	1536	1024	768	512	384	256	192	128	...	4
GPUdownsample (float)	2517	1419	638	364	172	104	56	37	25	19	18		18
GPUupsample (float)	2574	1438	643	367	172	103	57	37	24	17	17		17
GPUreduce (float)	6766	3724	1754	987	464	280	145	95	63	45	39		39
GPUexpand (float)	3993	2238	1678	595	453	168	87	56	37	26	21		21
GPUdecompose (float)	10203	5797	2706	1612	848	565	376						
GPUreconstruct (float)	6605	3745	1744	1032	538	354	229						

Routine	(U)char	U(short)	(U)int	(U)long	Float	Double
GPUdownsample (N=2048)	668	689	696	1277	638	1906
GPUupsample (N=2048)	473	557	669	1189	644	1236

Table 7: *Timings on a 285GTX GPU in milliseconds for different pyramid modules for different sizes N and different datatypes. For GPUdownsample holds N^2 to $(N/2)^2$, GPUupsample $(N/2)^2$ to N^2, GPUreduce for N^2 input, and GPUexpand for N^2 output. GPUdecompose and GPUreconstruct have been measured with 7 pyramid levels.*

Routine	Description	Size N	Float
GPUlut		2048	557
GPUpoly	k=3	2048	253
GPUpoly	k=13	2048	415
GPUlut		1024	162
GPUpoly	k=3	1024	67
GPUpoly	k=13	1024	107
GPUlut		512	59
GPUpoly	k=3	512	21
GPUpoly	k=13	512	31

Table 8: *Timings on a 285GTX GPU in micro-seconds for computation of a function. Size N denotes N^2 elements, k implies a polynomial function of $k-1^{th}$ order: $y_i = \sum_{i=0}^{k-1} c_i x_i$.*

4.5 Polynomial function and look-up table

In mathematics a function is a relation between a given set of elements (the domain) and another set of elements (the codomain), which associates each element in the domain with exactly one element in the codomain. These elements are often real numbers.

On a computer these functions are processed by a floating-point-unit. However many embedded processors, especially older designs, do not have hardware support for floating-point operations. Hence emulation or approximation is mandatory.

In this section a comparison is made between a look-up table. On the GPU standard texture hardware is available to perform linear interpolation using such a table. We used a table of 4096 elements between 0 and 1 as illustrated in Figure 9a.

Figure 9a illustrates the implementation of this test. At the left upper part an input images of $N \times N$ elements is read and processed by the lutGPU and polyGPU modules. Both routines consists of a merged set of modules that are given at the bottom left and right for the polynomial and LUT routine, respectively. The top right part simulates a LUT of 4096 elements and approximated by a polynomial least squares fitting algorithm called polyFit. Figure 9a illustrates both functions. The polynomial constants are used as input for the polynomial function.

Its stunning to find out that a polynomial function is faster than a look-up table even polynomials of order 12 are faster than a look-up table of 4096 elements.

How much time it takes extra to compute a polynomial function that is one order higher? We

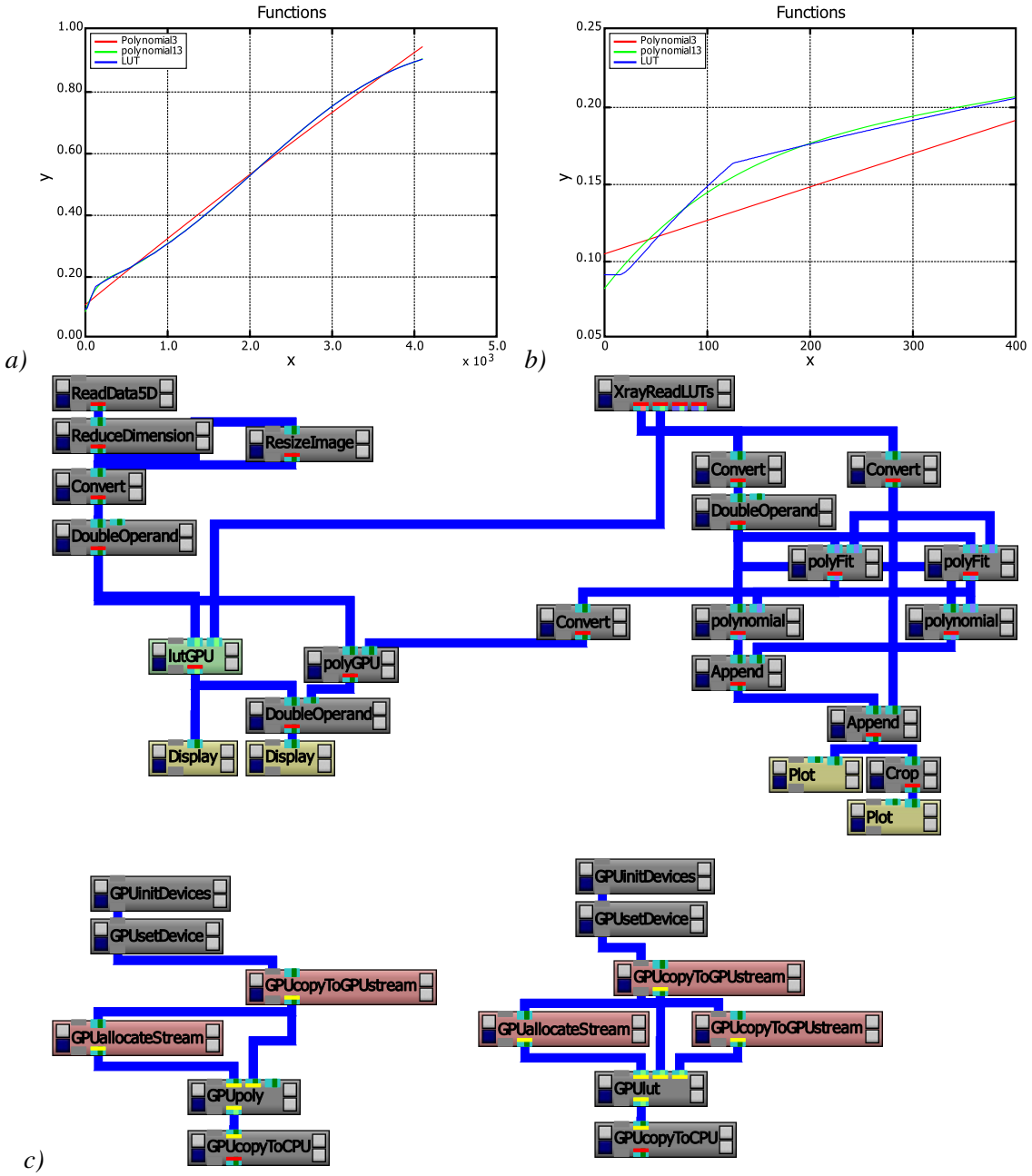

Figure 9: *a) Look-up table and two polynomial function approximations. b) Enlargement of (a). c) TiViPE network used for evaluation.*

measured 430.83 and 415.38 microseconds for k=14 and 13, respectively, see also Table 8. This implies 2048^2 operations in 15.45 microseconds. It implies that 271 Giga-operations are computed per second. Since the operation used to compute one degree higher is

```
val = val * di + cj;
```

Two floating point operations are required yielding 542 G-Flops or 51 percent of the peak performance of a 285GTX GPU.

References

[1] P. J. Burt and E. H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31(4):532–540, April 1983.

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

[3] T. Lourens. The art of parallel processing using general purpose graphical processing units – hardware, cuda introduction, and software architecture. Technical report, TiViPE, 2009.