ZESZYTY NAUKOWE POLITECHNIKI BIAŁOSTOCKIEJ 2010
Informatyka – Zeszyt 6

Tomasz Rybak[1]

USING GPU TO IMPROVE PERFORMANCE OF CALCULATING RECURRENCE PLOT

Abstract: Simulation and analysis of sophisticated systems require much computations. Moore's law, although still allows for increasing number of transistors on the die, does not lead to increase of performance of single chip — instead it leads to increased parallelism of entire system. This allows for improving performance of those algorithms that can be parallelised; recurrence plot is one of such algorithms. Graphical Processing Units (GPU) show the largest increase of parallel computations capabilities. At the same time they do not behave as traditional CPUs and require different style of programming to fully utilise their capabilities. Article shows techniques that can be used to increase performance of computing of recurrence plot on GPGPU.

Keywords: recurrence plot, non-linear analysis, fractal analysis, optimisation, parallel computations, GPGPU, CUDA

1. Introduction

Increase of performance of computers allows for using more sophisticated methods of analysis. Google's Page Rank algorithm or analysis of social networks done by Yahoo or FaceBook require enormous calculations to give results. Such algorithms perform so many computations and operate on such large amounts of data that single machines have problems with finishing calculations. Non-linear methods of analysis of systems, like recurrence plot described in this article, perform vast amounts of computations to provide user with characteristic of the system. Multi-core machines and distributed clusters allow for providing results faster as long as it is possible to parallelise computations.

While embedding many cores inside CPU is recent invention, producers of graphical cards have solved many problems with parallel systems that CPU manufacturers face. For many years even the cheapest GPUs were equipped with many units responsible for computing positions of points on the screen, fetching colour values from textures, drawing pixels after calculating their colours after analysing light

[1] Faculty of Computer Science, Bialystok University of Technology, Bialystok

present in 3D scene. Many scientists use GPUs for performing computations, using their enormous parallel capabilities. This requires changing description of scientific problem in the language of 3D graphics though. Article shows which libraries provided by GPU manufacturers can help with increasing performance of computations while requiring the least changes to algorithms.

This paper is organized as follows. The next section describes theory behind recurrence plots and its usage. Section 3. describes advances in hardware that allow for optimisations of calculations by using parallel capabilities of GPU. Section 4. describes details of implemented solution and achieved performance improvements using different methods. The last section presents summary and possibilities of improving program in the future.

2. Non-linear methods of analysis

Many systems show dynamic non-linear behaviour. Such systems are influenced by many parameters; values of those parameter change system in non-linear fashion. We assume that every system emits one dimensional signal (vector of samples); changes of values of those samples are caused by all variables describing the system.

Every analysis of non-linear system starts with determining dimensionality (size of phase space — the number of variables) of such a system. Finding dimensionality is done by using the loop. We begin with very small initial dimension d (1 or 2). Then, in the loop, we calculate dimension d_A of attractor reconstructed from such a system (described by our temporary dimension d). In the next iteration we increase dimension d and again calculate dimension d_A of reconstructed attractor. Loop is finished and final dimension d is known when reconstructed attractor's dimension d_A does not change after increasing d in the subsequent iterations.

Another variable influencing behaviour of non-linear system is the time delay τ. It is used to limit number of samples that are used to reconstruct attractor. Value of τ determines how many samples are used in analysis. Limiting number of points allows for avoiding clutter that would arise if all samples would be used to reconstruct attractor. Choosing proper value of τ influences validity of analysis of the system. If τ is to small, chosen points will be close to each other and system will appear stationary. When τ is too large, chosen points are far away and we lose information about trajectory of attractor in the phase space. To calculate proper value of τ one can use autocorrelation or mutual information (equation 1). The first minimum reached by the mutual information function becomes the value for delay τ.

$$I(X,Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{1}$$

Recurrence plot can be defined as Haeviside function over difference of distance of points is space (over chosen metrics) and some threshold. Recurrence plot treats values from one-dimensional signal as properties of points in multi-dimensional phase space. It takes d samples separated by τ values and calculates distance between points created from such samples. Points are close in the phase space when distance between them is less than ε (threshold distance). Distances between all pairs of points are computed. If points P_i and P_j are closer then threshold distance ε, value in matrix on point Rp_{ij} is 1, otherwise it is 0. Matrix showing distances between all points is called recurrence plot. Matrix contains either ones or zeros for all pairs of points — depending on whether pair of points is close or far away according to chosen metrics (equation 2). Because distance calculation is symmetrical operation, recurrence plot matrix is also symmetrical. Details of calculating recurrence plot can be found in the previous papers, [14] and [15]. Many implementations of recurrence plot can be found on the web page http://recurrence-plot.tk/.

$$R_{ij} = \Theta(\|x_i - x_j\| - \varepsilon) \tag{2}$$

Single recurrence plot is 2D matrix of values from set of $\{0,1\}$. It can be plotted on the screen of paper. Black dot (value of 1) at coordinates (i,j) means that on system at time i and j was in similar state, because its attractor was represented as points that were close together (their distance was less than chosen threshold). This means that dot is plotted if two sequences coming from input data are similar (their inner product is larger than threshold). This allows for visually analysing similarity of signal at different scales. However this technique requires large amounts of memory and long processing. Similar techniques are used in analysis of gene sequences (FASTA, BLAST) to find similar gene sequences.

After computing recurrence plot we can perform visual analysis looking for sets of black dots (meaning points that are close together). Repetition of results requires however automation of analysis of data. According to theory horizontal (and vertical, as this matrix is symmetrical) lines mean that system was stationary or was changing in laminar way, very slowly. This is because each non-zero value in recurrence plot mean that two points are close together. Line means that sequence contains points that are close together. Diagonal lines mean that system was recurrent — points were returning to the same space after some time. Repetition of system is described by divergence — the instability of the system.

3. Modern Graphics Hardware

Graphical hardware performs many calculations which are necessary to generate photo-realistic image: 3D coordinates of all objects in the scene need to be transformed, non-visible points need to be eliminated, lightning must be computed, all 3D coordinates need to be transformed into 2D screen coordinates, textured need to be mapped to generated pixels, and finally the finishing effects (like bump-mapping) need to be computed for all the pixels. Every object in the displayed 3D world must come through such a path dozens of times per second to achieve sufficient fluency of animation ([8]). For this reason for many years companies producing graphical cards were making them faster and faster thanks to Moore's law. At the same time they were putting more functionality to graphical cards, to allow for game's programmers to perform more visual effects on graphical cards, without occupying already busy CPU.

It is impossible to increase performance of graphical cards indefinitely by increasing frequency of hardware clocks — similarly to limitations known from the CPUs. But from the early days producers of graphical cards had easier problem to solve. In the case of images, the same calculations are performed on large sets of points. This means that graphical problems are extremely easy to parallelise. Graphical cards, instead of using faster clocks, were equipped with more and more processing units; each of them had limited performance, but overall GPU was very efficient because it was able to compute value for hundreds of pixels at the same time. The big milestone in development of graphical hardware was invention of shaders, introduced in GeForce 3 in 2001. Shaders were small pieces of code that described transformations done on vertices (points in 3D space) and pixels (point in 2D space — on the screen). Introduction of shaders allowed for increasing performance of GPU without increasing its clock speed: tiny fragment of code was executed by dozens of processors, each of them operating on different input data. Such approach to calculations is called SIMD — Single Instruction Multiple Data principle.

This inherent parallelism present in graphical domain is the main reason why GPUs differ from CPUs. While the latter use large amount of cache to hide inevitable latency when accessing memory or peripherals, GPUs are using vast amounts of threads that are ready to be executed. When one thread is waiting for data to be transmitted, GPU switches to another thread that is ready to be executed. This is possible because of extremely fast thread context switching, provided by hardware architecture. CPU in the same situation must predict which part of memory will be needed for the next few instructions and fetch it while current instructions are still being executed. If this prediction is incorrect, entire CPU must wait for the missing data to be

read from the memory. For the same problem GPU just switches for another thread to be executed ([8]).

Shader is a function intended to operate on graphical primitives. This means that programmer is forced to describe problem in domain of 3D graphics: matrix has to become texture, calculations are changed into vertex transformation, etc. Shader languages do not have advanced control statements that would allow for easy implementation of scientific code. GPUs are built to focus on the raw calculations, not transistors needed for implementing jumps and loops. All those features of GPUs and shaders mean that using full potential present on graphical card was not easy.

3.1 CUDA

In 2008 NVIDIA introduced CUDA (Compute Unified Device Architecture), which is intended to ease writing general-purpose programs that use graphical hardware. CUDA allows to write programs for GPU as easy as for CPU by using C language. Programs written in CUDA consist of two parts: one is executed on CPU and one on GPU. Part executed on CPU (function "main") is responsible for managing data, calling code on GPU, transferring data to and from GPU, etc. This part runs as long as program is running — its end causes execution of other parts to finish. Parts that are executed on the GPU are grouped in functions called "kernels". CPU loads data and points GPU (by using CUDA library and driver functions) which kernel to call with which parameters. Each kernel is run in parallel, by as many processors as there is on the chip. Each kernel is executed by many threads and each thread is responsible for operating on fragment of input data. CUDA is responsible for mapping threads to different fragments of input. This mapping is done automatically, without need to manage it by programmer. This way hardware is almost always (depending on optimisations) fully used: on weaker GPU there will be 1000 threads running, on more powerful 16000 threads, so problem can be solved 16 times faster, but in both cases the same program will be run, and the same set of calculations will be performed. The only difference will be that in one case there will be more threads executing chosen kernel. To experience performance gains it is advised to run more threads than there is physical processors. Additional threads will be used to hide memory access latency; when GPU needs to access slow memory it switches to another thread and executes its code while waiting for data to arrive ([8]).

CUDA is not the only possible solution for using graphical hardware for calculations. ATI has its own solution called "Stream" that is using Radeon GPGPU for calculations ([2]). Intel is promising Larrabee chip which is supposed to be multicore processor similar to the ones used by NVIDIA and ATI, but it is promised to be

compatible with x86 instruction set and offer strong cache coherency, meaning that it should be able to run existing programs without changes or recompilation. Another example of hardware offering parallel execution is Cell processor, made by Sony, Toshiba and IBM, used in PS3 and high-end computational hardware sold by IBM. The latest proposal in the world of parallel execution engines in OpenCL (Computing Language, [6]). As OpenGL allows for using the same code to run on different graphical cards, OpenCL allows for writing programs that can run computations on CPUs and GPUs without any source code changes. OpenCL programs use shader units on GPUs and cores on CPUs to provide programmer with access to parallel execution units in the same way, regardless of used hardware.

It is easy to see graphical roots of CUDA; threads are grouped in blocks, and blocks in grids. Blocks and grids have 3 dimensions and are placed in 3D space. This reflects hardware architecture — which first purpose is to operate on 3D and 2D graphical primitives. But this architecture is not limiting CUDA — one thread can perform execution on one element of matrix; this thread-block-grid placing make it rather easy to manage large amounts of threads and to know which thread is performing calculations on particular part of input data. Such an architecture is the compromise between abstraction and being close to the hardware. It allows to achieving as much performance as possible by managing threads and their groups..

To help with scientific problem solving CUDA is offering many built-in functions, like basic arithmetic functions, trigonometric functions, logarithms, etc. Full list of available functions and details of their implementation can be found in [5]. Because of hardware limitations many of currently available graphical cards do not offer fully hardware optimised and at the same time IEEE-754 compliant implementations of some functions. Programmer must often trade accuracy for speed. Situation changes quite rapidly though, as new hardware adds new mathematical functionality.

There are many levels of CUDA capabilities. New graphical cards add new features, like better thread management, support for double precision float point numbers, more mathematical functions implemented in the hardware, etc. Each graphical card presents programmer with version of offered functionalities, in the form of number; consumer-oriented GPUs have computing capabilities 1.1, scientific-oriented Tesla cards offer 1.3 capabilities. It is impossible to run program intended for CUDA 1.3 on card with capabilities 1.1, so it is important to know what is possible and which options of compiler are used. Details of all capabilities and matrix describing which chip and graphics card offers which capabilities can be found in the CUDA Programming Guide ([4]).

CUDA is not hiding different types of memory present in the GPU. Programmer can access all memory types that are available in the hardware. Each of those memory

types has own advantages and disadvantages. Some of them are cached and some are not, some are private to threads or block of threads, other are shared among all executed kernels. While writing CUDA kernels and CPU code maintaining GPU code execution programmer must be aware of hardware limitations of memory access: cache coherency, cache conflicts, cost of accessing memory belonging to different banks. It is important to use different memory types to hide inefficiency of every one of them. For program to be executed with maximum possible performance, programmer must use combination of many of those memory classes. There is need to choose own trade-off between size, performance, and flexibility of used memory.

There are many research groups trying to increase performance of programs using GPUs. Gu et. al. [10] use different types of memory and different states of programs to increase performance of FFT. They change code to best use available hardware which allowed them to achieves better performance than libraries provided by NVIDIA. Another team trying to improve performance of FFT is lead by Nukada [13]. Bakkum et. al. use CUDA to speed up performance of database operations [1]. They were able to achieve performance gains from 20 to 70 times. Malony et. al. created sophisticated system that runs programs, analyses its trace, and then uses this information to try to increase performance [12].

3.2 PyCUDA

CUDA offers its functionalities to programs written in the C language. This choice means that it is available for many platforms, and it can be used from any language that can call C libraries.Calling CUDA kernels from C is not easy; there is much program state to manage, as one needs to manage memory on CPU and GPU, transfer data between them, create context in which kernels are called, manage number of threads, etc. That's the reason why many libraries were written to call CUDA from different languages. There are binding for CUDA in Java, .net, Python. Calling CUDA in high-level language makes it easier to use GPU to increase performance of computations. We avoid tedious resource management, and also does not get much performance hit — the most performance-dependent code is executed on GPU. Avoidance of resource management can also allow for focusing on better optimisation, and with some effort can even lead to CPU-GPU parallelism, as code on CPU and GPU are independent and can be called asynchronously.

PyCUDA is the CUDA library for Python — high-level interpreted language. Python offers many libraries for performing much of the most common programming task, and provides many build-in structures, like lists, dictionaries. It is able to execute code in two modes: as full programs, which are passed to the interpreter, and

in interactive mode, in which programmer is giving orders in the real time. The latter mode is great for testing new ideas; interactive mode of Python connected with PyCUDA is wonderful for experimenting and looking for possible optimisations ([11]).

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

a = numpy.random.randn(4000).astype(numpy.float32)
b = numpy.random.randn(4000).astype(numpy.float32)
c = numpy.empty(4000).astype(numpy.float32)

m = SourceModule("""
__global__ void add(float *a, float *b, float *c, int N) {
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx < N)
c[idx] = a[idx] + b[idx];
}
""")
f = m.get_function("add")
f(cuda.In(a), cuda.In(b), cuda.Out(c), 4000, blocks = (256, 1, 1), grid=(16, 1))
```

Fig. 1. Program calculating sum of vectors using PyCUDA

PyCUDA's intention is to provide access for any functionality available in CUDA. Sample Python code calling kernel adding two vectors is shown on Figure 1; it it Python version of program shown in Section 1.3.4 of [3]. PyCUDA provides programmer with all advantages of Python such as Garbage Collector for all objects, including those stored on the GPU. This makes writing programs much easier. For managing resources that are scarce (like memory on the GPU) programmer may manually free any object, but this is not needed, as it will be freed during the next run of garbage collector. PyCUDA presents CUDA errors as exceptions to better integrate CPU and GPU code. PyCUDA accepts CUDA code as strings and calls compiler; NVIDIA compiler accepts the CUDA code and returns binary machine code. PyCUDA sends this binary to GPU; to avoid constant compilation PyCUDA caches compiled binaries and calls compiler only if code to be executed changed. This means that it is possible to change code during program's execution, and PyCUDA will be able to deal with it. This CUDA code cache is hold on the hard drive so it can be used during subsequent program runs. This also means that for the first execution call will

be slower as compiler will need to be run, but for all the next ones everything will be as fast as for original CUDA.

PyCUDA treats and manages CUDA programs as strings. This is the new implementation of old LISP idea "code is data". For PyCUDA program CUDA code is data (formally string inside the program) so it is possible to operate on it. By using templates it is possible to write skeleton of code and fill it with details during execution, when those values are known. This can be seen as analogous to implementing Strategy or Template Method patterns ([9]) but without subclassing and object-orientation.

At the same time writing multi-threaded programs that would be executed on CPU in Python does not bring much performance benefits. Python interpreter has Global Lock; it means that all instructions that are written in Python (even if they are executed in different threads) must wait for one another to be interpreted. Only one thread may be active in the interpreter at one time. This is not a problem in case of CUDA as all threads are executed on GPU and not on the CPU, but it disallows for improving program's performance by parallelising CPU code. CUDA threads are not interpreted by Python environment and need not wait one for one another under Global Interpreter Lock. This lock causes any CPU-threaded code in Python not to run faster than sequential code; it must be taken into consideration when analysing performance.

Python is not using machine code but internal representation of programs so it is able to provide programmer with more details regarding execution of code. For each kernel in CUDA programmer may access variables that point her how large the function is, how many registers it is using, how much shared memory is needed to execute particular kernel in one thread. This allows insight into hardware occupancy and can lead to optimisation; e.g. if function is using too much registers, it could be beneficial to move some input parameters to the constant memory, and allow compiler to use freed registers for optimising execution of kernel function. Detailed time information regarding execution is available via timers, which offer sub-millisecond accuracy and allow for timing of any events is GPU ([3]).

4. Optimising computations of recurrence plot

I wrote program in Python and program in C that calculate recurrence plot. The C program computes recurrence plot using CPU; the Python program contains two sets of functions, one to perform computations on CPU and one that used GPU. This way program can be run on machines equipped with capable graphics card and on ones that do not contain NVIDIA-based GPU — appropriate functions are called depending on presence of hardware detected at the start of the program. All those

program are used to compare results obtained by using CPU and GPU. Programs in C and Python were also used to test influence of interpreting of Python code on performance.

CPU code is single-threaded and loops over all points in the resulting recurrence plot; it is just implementation of equations shown in Section 2.. For the GPU code I followed advice from NVIDIA Best Practices ([3]) and decided to use one GPU thread to calculate value of one point in the recurrence plot matrix.

Creation of highly parallel code was possible because in recurrence plot all output points depend on the shared input vector, but do not depend on values of other output points. This means that values calculated by each of the threads are independent and no thread need to wait for any other thread. All threads are thus independent and may run at the same time, reducing the time of calculations as many times as there is threads running.

Figure 2 shows comparison of time needed by GPU and GPU to compute recurrence plot. Times are shown using logarithmic scale. GPU requires much less time to compute recurrence plot that CPU. Until there is still space of GPU to run more threads, increase of size of input vector is not visible. It means that it takes the same time to compute recurrence plot as long as not all cores in GPU are used.

In case of PyCUDA the first run of code is always slow because of need to compile code. Later executions are faster as PyCUDA caches already compiled code to save time needed to run it. As long as source code does not change only first run is slow which is visible in Table 1. Table shows time (in seconds) that it took to perform computations. It shows that for CPU performing computations 10 times took 10 times longer. For GPU performing computations 10 times took only slightly longer time — most of the time was taken by compilation of code. This is reinforced by the last column of Table 1 which shows time needed to perform computations on GPU, without compiling code.

As can be seen in Table 1 time needed to perform computations on CPU is proportional to the size of the input data for both C and Python version. The largest size of input data is smaller for GPU (10000 samples) than for CPU (16384 samples) because of lack of space in graphics memory for larger input data. Table 1 shows that for the small size of data using GPU is not beneficial. Any gains in performance achieved when using GPU for performing computations are lost because of cost of compiling GPU code and transfer of data between host and device. But as soon as input data is large enough (512 samples in case of described hardware and recurrence plot for Python code, 2048 for C code) the time of performing computations on GPU is shorter than on CPU, even including management tasks. For the largest size of

input data (8192 samples) time needed for compute recurrence plots on GPU is 210 times smaller than time used by CPU.

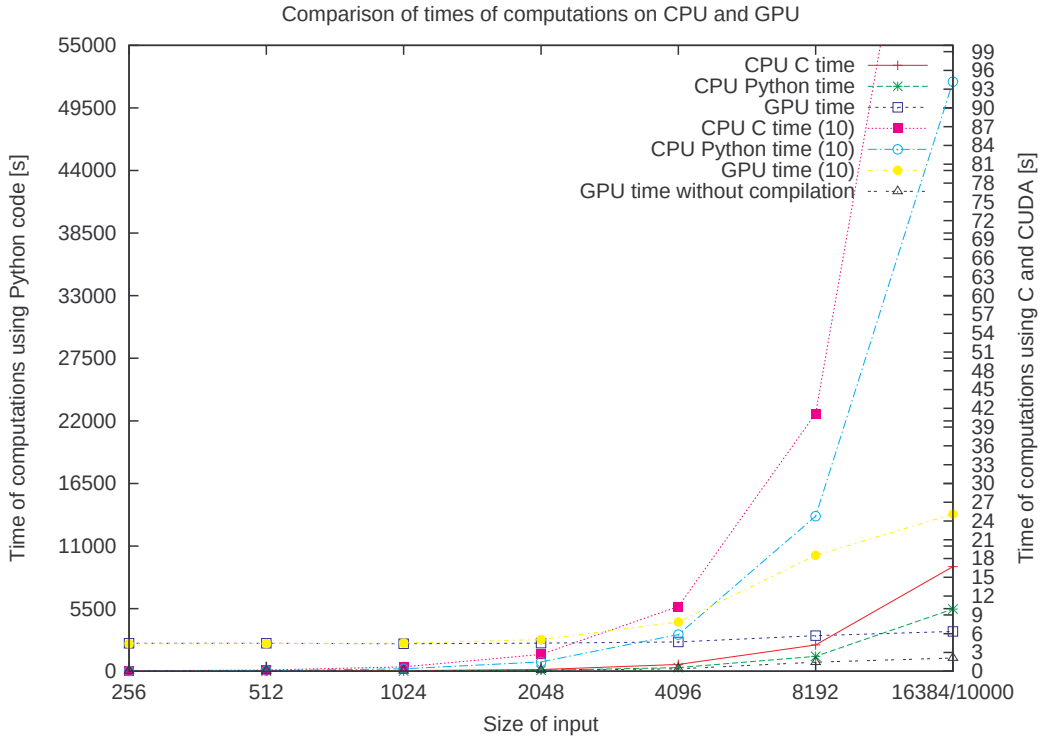

Fig. 2. Time needed for CPU and GPU to calculate recurrence plot.

To detect trends in the signal it is useful to calculate not only one recurrence plot but many of them, each for part of the signal starting at different moment. This procedure will generate not 2D matrix (recurrence plot) but series of them. This can be seen as creation of 3D matrix — which can take advantage of 3D block of threads provided by CUDA GPU. It poses problem with managing third dimension (Z axis) of the threads. In all of the currently available hardware Z dimension offers the smallest range of values. Also block also have the limited number of threads in it ([4]) so one must decide which axis will be given the most threads, and which will need to use CPU loop to manually call all kernels with appropriate parameters. The best choice

Size	CPU-C (1)	CPU-Python (1)	GPU (1)	CPU-C (10)	CPU-Python (10)	GPU (10)	GPU compiled
256	0.007	1.28	4.47	0.06	12.21	4.40	0.05
512	0.032	5.08	4.45	0.19	49.65	4.42	0.06
1024	0.091	20.63	4.40	0.69	210.77	4.49	0.07
2048	0.267	85.32	4.47	2.68	815.60	5.06	0.14
4096	1.072	322.68	4.67	10.33	3219.37	7.86	0.39
8192	4.186	1307.67	5.68	41.17	13622.35	18.50	1.43
16384/10000	16.705	5462.22	6.36	165.63	51804.87	25.11	2.10

Table 1. Time needed for CPU and GPU to calculate recurrence plot.

is to use the largest number of threads in the one plane with the same Z value ([3]). This means that it is impossible to perform all calculations at the same time and it is necessary to call kernel many times for the different values of Z. But the speedup achieved because of caching and avoiding repeated transferring the same input vector over the bus makes it worth it even for the price of more sophisticated code.

The code analysing recurrence plots uses one thread to calculate lengths of lines in one row, column or diagonal line. Because histogram is generated for the entire matrix there is a risk of inter-thread conflicts in histogram. To avoid this clash I use atomic functions that were introduced in CUDA 1.1. This means that this code can be no longer executed on low-level CUDA hardware like GeForce 8xxx but only on GeForce 9400 and better. Currently program does not check capabilities of CUDA hardware and assumes that if there is CUDA-capable GPU it can run any code regardless of required capabilities.

4.1 Increasing the performance by using hardware capabilities

Usage of GPU allows for great improvements of performance. At the same time CUDA does not hide implementation details, but makes it possible to see hardware that runs kernels, for example different types of memory. Proper usage of different types of memory allows for gaining even more performance. At the same time improper usage of memory types might cause code to be executed inefficiently and destroy all performance gains.

GPUArray present in the PyCUDA stores data in global memory of graphics card. This is the largest memory but it is also the slowest one. It is not cached which means that consecutive reads are as slow as the first one [5]. Among different types of memory texture memory is the best suited for computing recurrent plot. This is read-only memory that is not very fast but it is cached in the spatial way. It is used for texturing in graphics programming; spacial caching means that not only read value

but also its neighbours (their number depending on dimensionality of texture being read) are read ([3]). Texture cache assumes that when one texture fragment is used its neighbours will soon be needed to draw neighbour pixels on the screen. In the case of recurrence plot the situation is very similar — close threads read very close values from the input vector (see description of recurrence plot in the Section 2.). Texture memory was then natural choice for speeding up the program.

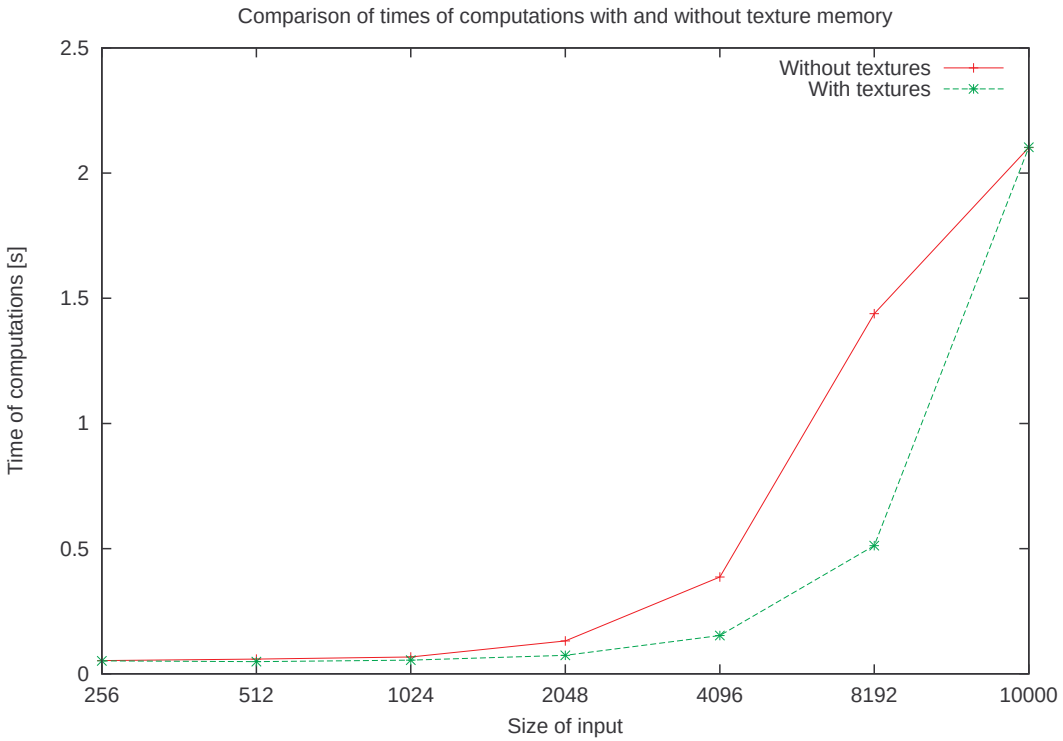

Fig. 3. Time needed to calculate recurrence plot with and without texturing

The problem with texture memory is that it is limited in size. Depending on the type of values stored in the texture and the dimensionality (1D, 2D, 3D), the size of texture in one dimension is limited to $2^{12}(4096)$ or $2^{13}(8192)$ elements. This means that it is not possible to use texture memory to perform calculations on large input vectors.

To allow usage of texture memory when it is possible program was changed that it compares size of input signal with maximum texture size. If it is smaller, texture memory is used; if not, global GPU memory is used to store the input data. Check of size is performed on run-time and is checked for each signal independently. It means that no input is punished because of size of previously computed data.

Figure 3 shows time needed for calculate recurrence plot with and without usage of the texture memory. The same data is shown in Table 2. Usage of texture memory improves performance. For small size of input data difference is insignificant but as data size grows difference in performance gets significant. As can also be seen performance is the same for both cases when data is too large to fit into texture memory. As noted earlier instead of failing program notices that it cannot use texture memory and switches to the main memory. It prevents computations from failing, even though they are slower is such a case.

Size	Not using textures	Using textures
256	0.05	0.05
512	0.06	0.05
1024	0.07	0.05
2048	0.13	0.07
4096	0.39	0.15
8192	1.44	0.51
10000	2.10	2.10

Table 2. Time needed to calculate recurrence plot with and without using texture memory

It should be possible to use shared memory as described in “CUDA Best Practices Guide” [3] in case in which input data is too large to be able to use texture memory. Another possibility of optimisation is different usage of threads. Currently one thread performs very limited computations — only few additions and multiplications, for calculating one point in matrix. This is not the best solution according to “Best practices”. The key for achieving the best performance is to implement code and instrument it; this will lead to the rich library of functions and ability to choose one basing on available hardware and performance needs.

4.2 Metaprogramming

According to Andreas Kloeckner [11] it is possible to achieve great performance by checking all possible choices for better performance (like unrolling loops, chang-

ing values of parameters as constants), generating code, and checking which variant gives the best performance. PyCUDA with its ability to generate code allows for easy experimentation to achieve great performance.

As noted in Section 3.2 PyCUDA stores kernels as text which allows for easy manipulation of those functions. This ability was used to generate programs on runtime, depending on the signal and parameters of execution; it allowed to avoid performance bottlenecks present in GPU hardware. GPU does not gives good performance for code that uses loops and conditional instructions. If there is divergence in code paths executed by different threads all other threads are stopped and only threads that contain particular jump conditions are executed ([3]).

Thanks to textual representation of CUDA code I was able to use template engine Cheetah to generate code in Python to CUDA. The most important change done was removal of the internal loop that depends on the dimension of calculated recurrence plot d (see Section 2.). Instead of looping program generates as many operations as there would be loop resolutions. This means that code executed on each of the threads is the same and there is no risk of divergent execution. This also allows for removing all looping variables. Because all threads in the block share registers, the more variables is used by thread, the less threads can be run simultaneously. Decreasing the number of variables used by threads allows for running more threads at the same time.

Usage of generated code allows for reducing number of parameters passed to the kernel. In the canonical case all variables (τ, dimension d, threshold ε) must be passed as parameters. This limits number of available registers and makes source code more sophisticated. All threads share pool of available registers, so every operation that increases number of available register allows for running more threads at the time. Switching to the generating code that includes values of parameters as constants allows CUDA to store this values in constant memory. This is special memory that is cached and shared among threads. In many cases read of all threads of the same position in constant memory is as fast as reading of one thread (see [4]) and it reduces number of used registers. Although the change of any value of parameters (dimension, tau, or threshold) forces recompilation of kernel code, the same change allows for unrolling loops — increasing overall GPU code performance.

Generating code simplifies situation with texture and global memory. Instead of two variants of each of the functions (one for global one for the texture memory) parameter was added to the Cheetah engine to choose which kind of memory should be used. When code is generated, size of input vector determines which kind of memory is used and appropriate variant of the used function is generated. This means that code calling the kernel function must know which variant was generated, but

this problem can be solved with usage of "with" statement present in Python[2]. The "with" statement allows for writing code surrounding particular block and is executed at the beginning and at the end of the block. This is the variant of the Aspect-Oriented Programming (AOP).

The final method of optimisation is using compiler options. Very often compiler does not have information about our code as is not able to change algorithm to use different type of memory or different hierarchy of threads. At the same time compiler can be used to optimise low-level details — remembering that people are better with high-level image of entire situation. Different options of compiler optimisations can be used after all described optimisation techniques are used. For example when entire loop is unrolled and can be executed without even one jump, there is no need to try to detect and resolve thread divergences and compiler may use other optimisations.

5. Summary

Usage of highly-parallel hardware on GPU via CUDA gave enormous performance improvements. It was done on sub-100EUR NVIDIA GeForce 9500GT. NVIDIA offers full range of hardware, from such cheap cards to Tesla cards intended to be used in high-end computational clusters.

Article used approach successfully applied by other researchers ([10], [12]). I was also able to achieve similar performance gains, from 20 to 100 times ([1]). Although created system is not sophisticated as described by Malony ([12]) there are many possible extensions to program described in this article.

Program does not check capabilities of CUDA hardware and assumes that if there is CUDA-capable GPU it can run any CUDA code. This should change in the future versions of the program.

The biggest chance to improve overall performance is avoiding transferring data between host and device. Currently each kernel call requires transferring data to GPU, performing computations, and transferring results back to CPU memory. Ability to hold working set on the device could decrease time needed to perform analysis. At the same time it would reduce amount of memory available for computations. Another disadvantage of such solution is risk of divergence of data stored in memory belonging to CPU and GPU. This might lead to different results depending on which device executes the code — especially as CUDA does not offer full IEEE-754 compatibility yet.

In the middle of 2010 NVIDIA introduced new GPU chip family called Fermi. It is posed to replace Tesla chips, and offers vast improvements that can lead to better

[2] Described in PEP 343, http://www.python.org/dev/peps/pep-0343/

performance: integrated address space, better cache hierarchy, different management of threads, and so on ([7]). This means that even more decisions regarding code that might influence performance of computations are needed. Because of financial limitations author does not have access to Fermi chip and is not able to compare results between Tesla and Fermi, At the same time problems described in this article (and solutions to them) are even more important as hardware offers more features.

References

[1] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM.

[2] ATI Corporation. *ATI Stream Computing OpenCL Programming Guide*, 08 2010. http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guid Accessed at 2010-10-04.

[3] NVIDIA Corporation. *NVIDIA CUDA C Best Practices Guide Version 3.2*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf Accessed at 2010-10-04.

[4] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide Version 3.2*, 09 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf Accessed at 2010-10-04.

[5] NVIDIA Corporation. *NVIDIA CUDA Reference Manual Version 3.2*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_Toolkit_Reference_Manu Accessed at 2010-10-04.

[6] NVIDIA Corporation. *NVIDIA OpenCL Programming Guide for the CUDA Architecture Version 3.2*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Programming_Guide.p Accessed at 2010-10-04.

[7] NVIDIA Corporation. *Tuning CUDA Applications for Fermi Version 1.3*, 08 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/Fermi_Tuning_Guide.pdf Accessed at 2010-10-04.

[8] Kayvon Fatahalian and Mike Houston. A closer look at gpus. *Communications of ACM*, 51(10):50–57, 2008.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1995.

[10] Liang Gu, Xiaoming Li, and Jakob Siegel. An empirically tuned 2d and 3d fft library on cuda gpu. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 305–314, New York, NY, USA, 2010. ACM.

[11] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda: Gpu run-time code generation for high-performance computing.

[12] Allen D. Malony, Scott Biersdorff, Wyatt Spear, and Shangkar Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 127–136, New York, NY, USA, 2010. ACM.

[13] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[14] Tomasz Rybak and Romuald Mosdorf. Computer users activity analysis using recurrence plot. In *International Conference on Biometrics and Kansei Engineering*, Cieszyn, Poland, 2009. AGH.

[15] Tomasz Rybak and Romuald Mosdorf. User activity detection in computer systems by means of recurrence plot analysis. *Zeszyty Naukowe Politechniki Białostockiej. Informatyka Zeszyt 5*, pages 67–86, 2010.

UZYCIE GPU W CELU ZWIęKSZENIA WYDAJNOśCI OBLICZANIA RECURRENCE PLOT

Strzeszczenie: Analiza skomplikowanych systemów wymaga przeprowadzenia wielu obliczeń. Prawo Moore'a, choć wciąż pozostaje w mocy, nie pozwala na zwiększanie wydajności pojedynczego procesora, ale pomaga w tworzeniu wydajnych równoległych systemów. Pozwala to na zwiększanie wydajności dla algorytmów które można zrównoleglić; recurrence plot należy do takich algorytmów. Procesory graficzne (GPU) oferują największą ilość równoległych jednostek obliczeniowych, jednocześnie jednak ich wydajne wykorzystanie wymaga innego podejścia programistycznego. Artykuł opisuje w jaki sposób wykorzystano technologię CUDA do przyspieszania obliczania recurrence plot.

Słowa kluczowe: recurrence plot, analiza fraktalna, optymalizacja, obliczenia równoległe, GPGPU, CUDA

Artykuł zrealizowano w ramach pracy badawczej S/WI/2/09.