ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ, ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Διπλωματική Εργασία

Μεταφορά και μελέτη της εφαρμογής εκτίμησης φυλογενετικών δένδρων “PBPI” σε many-core σύστημα

Συγγραφέας: Καλογήρου Χρήστος

Επιβλέποντες καθηγητές:
Αντωνόπουλος Χρήστος
Επίκουρος Καθηγητής

12 Μαρτίου 2012

UNIVERSITY OF THESSALY

DEPARTMENT OF COMPUTER AND COMMUNICATION ENGINEERING

Undergraduate Diploma Dissertation

Porting and Study of Phylogenetic Tree Inference Application "PBPI" on a Many-Core system

Author: Kalogirou Christos

Supervisor: Antonopoulos Christos
Assistant Professor

12 March 2012

Ευχαριστίες:

Να ευχαριστήσω τους καθηγητές μου τον κύριο Αντωνόπουλο Χρήστο και τον κύριο Μπέλλα Νικόλαο για τη συνεχή καθοδήγησή τους καθ' όλη τη διάρκεια της υλοποίησης της διπλωματικής μου εργασίας. Επίσης να ευχαριστήσω την οικογένειά μου και τους φίλους μου για τη συνεχή τους στήριξη.

Περίληψη

Το αντικείμενο μελέτης μας είναι η φυλογενετική εφαρμογή PBPI. Η PBPI είναι μια εφαρμογή που χρησιμοποιείται για την εκτίμηση φυλογενετικών δένδρων. Η PBPI χρησιμοποιεί το phylogenetic Bayesian inference, έναν αλγόριθμο για εκτίμηση φυλογενετικών δένδρων, σε συνδυασμό αλυσίδες Markov. Επίσης η PBPI ενσωματώνει κάποιες αλγοριθμικές βελτιστοποιήσεις σε σχέση με αντίστοιχες εφαρμογές και χρησιμοποιεί MPI interface για να επιτύχει παραλληλία στον κώδικα προκειμένου να μειώσει σημαντικά το χρόνο εκτέλεσής της.

Σκοπός της μελέτης μας είναι να καταφέρουμε να μεταφέρουμε τη PBPI σε πολυεπεξεργαστικό σύστημα και ειδικότερα πάνω στην κάρτα γραφικών. Στο κείμενό μας αναλύουμε την αρχιτεκτονική της κάρτας γραφικών και γιατί είναι ιδανική για υπολογισμούς σε σχέση με τον επεξεργαστή. Μεταφέροντας τη PBPI πάνω στην κάρτα γραφικών θα μελετήσουμε τη συμπεριφορά της και αν πράγματι η κάρτα μπορεί να βοηθήσει στη βελτιστοποίηση της απόδοσής της.

Επίσης σκοπός της μελέτης μας είναι να δείξουμε και τεχνικές που μπορούν να εφαρμοστούν στην κάρτα γραφικών. Η κάρτα είναι ιδανική για υπολογισμούς, αλλά έχει και αυτή αδυναμίες και χρειάζεται ιδιαίτερος χειρισμός μερικές φορές. Θα δείξουμε τρόπους που μπορούμε να ξεπεράσουμε αυτές τις αδυναμίες κάτι που μπορεί να είναι επαναχρησιμοποιήσιμο και από άλλες εφαρμογές.

Ακόμα θα δείξουμε όλα τα βήματα και στάδια που περάσαμε για να φτάσουμε στην τελική μας απόδοση. Μεταφέροντας μια εφαρμογή πάνω στην κάρτα δεν πετυχαίνουμε κατευθείαν τη μέγιστη απόδοση, όμως τις περισσότερες φορές είμαστε μακριά από αυτή. Θα αναλύσουμε όλες τις βελτιστοποιήσεις που κάναμε και πως τελικά φτάσαμε στην τελικά μας απόδοση.

Επόμενο βήμα μας είναι να συγκρίνουμε την τελική απόδοση της PBPI πάνω στην κάρτα γραφικών σε σχέση με αυτή της αρχικής υλοποίησης για τον επεξεργαστή. Με τη σύγκριση θέλουμε να συμπεραίνουμε αν πράγματι η PBPI είχε κάποιο κέρδος από όλη αυτή τα διαδικασία όπως υπόσχεται το μοντέλο προγραμματισμού για τις κάρτες γραφικών.

Στο τέλος της διπλωματικής υπάρχει και ένα παράρτημα με περαιτέρω μετρήσεις. Οι μετρήσεις αυτές είναι με overclocked ρυθμίσεις. Σε αυτό το κομμάτι φαίνεται ξεκάθαρα ποιοι παράμετροι της κάρτα γραφικών επηρεάζουν άμεσα την απόδοσή της στην PBPI.

Abstract

This undergraduate Diploma Dissertation studies the phylogenetic application called PBPI which estimates phylogenetic trees. PBPI combines Bayesian phylogenetic inference with Markov chains. It uses the MPI interface for exploiting parallelism and uses algorithm improvements in contrast with others similar applications in order to achieve great performance. The purpose of this undergraduate Diploma Dissertation is to make a new implementation for PBPI on a GPU. We analyze the architecture of a GPU and why a GPU is suitable for tough computations. We will study the performance of PBPI on the GPU to see if we can get any performance improvement. Also we will show some techniques we may use for GPU programming. A GPU may be powerful for computations but it has some weaknesses. We will demonstrate some ways for overcoming these weaknesses and how we can use these ways for other applications. We will analyze all the optimizations that were applied on our implementation of PBPI in order to achieve a better performance. These optimizations can be used in other applications too. We compare the performance of the initial PBPI implemented on CPU with that implemented by us on GPU. We also have some graphs, representing the total execution times for CPU and GPU and see if the GPU can really improve the performance.

Περιεχόμενα

Chapter 1 - Introduction

Exploring the mysteries of life is one of the most demanding sectors of biology. Scientists try to explain how life appeared on earth and how all the organisms were created and the relatedness between them using phylogenetic trees. This demands months of research especially if they have to analyze large DNA sequences. There have been created many algorithms for that purpose.

One of the most useful algorithms is the Bayesian phylogenetic inference. It is incorporated in many application because offers a very good quality of the phylogenetic tree. PBPI also incorporates Bayesian phylogenetic inference. PBPI is one of the most fast phylogenetic applications for estimating phylogenetic trees because has algorithm improvements and uses MPI interface for exploiting parallelism. Nevertheless analyzing DNA sequences of different species with PBPI may take several hours of computation.

Applications like PBPI usually have intensive computations. These computations delay the execution of PBPI too much. A CPU cannot execute them effectively. This is why nowadays the GPUs are used for computations. A GPU has more cores than a CPU and it can be used for making calculations in parallel giving a great performance.

Our aim is to port PBPI on a GPU. We will make all the computations on the graphics card to exploit parallelism and improving the performance. We will compare the time needed for the computations on a GPU with that needed for a CPU and we will analyze the results to see if PBPI can actually gain any improvement from the GPU.

Programming on a GPU is little more complex than programming on a CPU. PBPI uses recursion for accessing a tree and a GPU does not support recursion. A technique how a GPU and a CPU can be combined to overcome this problem will be represented. Also there is analyzed how the performance of PBPI can be improved when is ported on the GPU. The GPU programming model has a weakness called memory transfers that may delay the execution of PBPI too much. The optimizations that are done are analyzed and these optimizations can be reused in other applications.

In chapter 2 we analyze the Bayesian phylogenetic inference and the phylogenetic application PBPI. In chapter 3 we represent the architecture of the graphics card that was used and the differences between a GPU and a CPU. In chapter 4 we discuss how to port PBPI on the GPU the optimizations that were made on the GPU and the performance comparison between the GPU and the CPU. We discuss our final thoughts in chapter 5 while in the annex we have some measurements with overclocked settings for the GPU.

Chapter 2 – Studying Phylogenies and the application PBPI

2.1 - An Introduction in Phylogenies

Through centuries many species have disappeared and other have changed so much that new species were created. All these species may have a common ancestor. For many years the scientists are trying to figure out the relatedness between species, even if the species may have lived on earth on different periods. The method they use to study the evolution of species is called phylogeny.

Phylogeny studies the evolutionary relatedness among groups of organisms, which is discovered through molecular sequencing data and morphological data matrices. In a few words phylogeny tries to find the similarities through species taking information from their genes. There are two ways that genes can be inherited: vertical gene transfer and horizontal gene transfer. Vertical gene transfer is the passage of genes from parent to offspring and horizontal gene transfer occurs when genes jump between unrelated organisms. The history of organismal lineage is inextricably connected to these gene transfers.

One serious problem that biologists have to face is that evolution takes place over long periods of time. They have to reconstruct phylogenies inferring the evolutionary relationships among present-day organisms. To accomplish this, they use fossils, but unfortunately the fossils often have a lack of information.

To manage to do all this research on evolution scientists need to include two major steps: a) to construct a phylogenetic tree that maps the evolutionary relationship among a group of taxa, and b) to access the confidence on the estimated tree given the observed data.

A plylogenetic tree helps biologists to visualize evolution. The evolution of species consists of branches of a common ancestor. This information can be seen in Figure 2.1. These trees can describe the descending pattern or evolutionary relationship among species or organisms. The poorness of fossils affects these trees as well, so the genetic data are only available for living taxa because the fossils are often too poor.

The reason of this research is that phylogenies are very useful because all biological phenomena are the result of evolution. Phylogenies help us in several areas such as pharmaceutical research, drug discovery, agricultural plant improvement and disease control studies.

There are several methods to study phylogeny and infer phylogenetic trees. Some of them are parsimony[1] where several trees are estimated and the one that best fit our criteria is chosen, maximum likelihood[2] where the tree with the maximum probability is chosen and MCMC-based Bayesian Inference[3] which will be analyzed on next chapter.

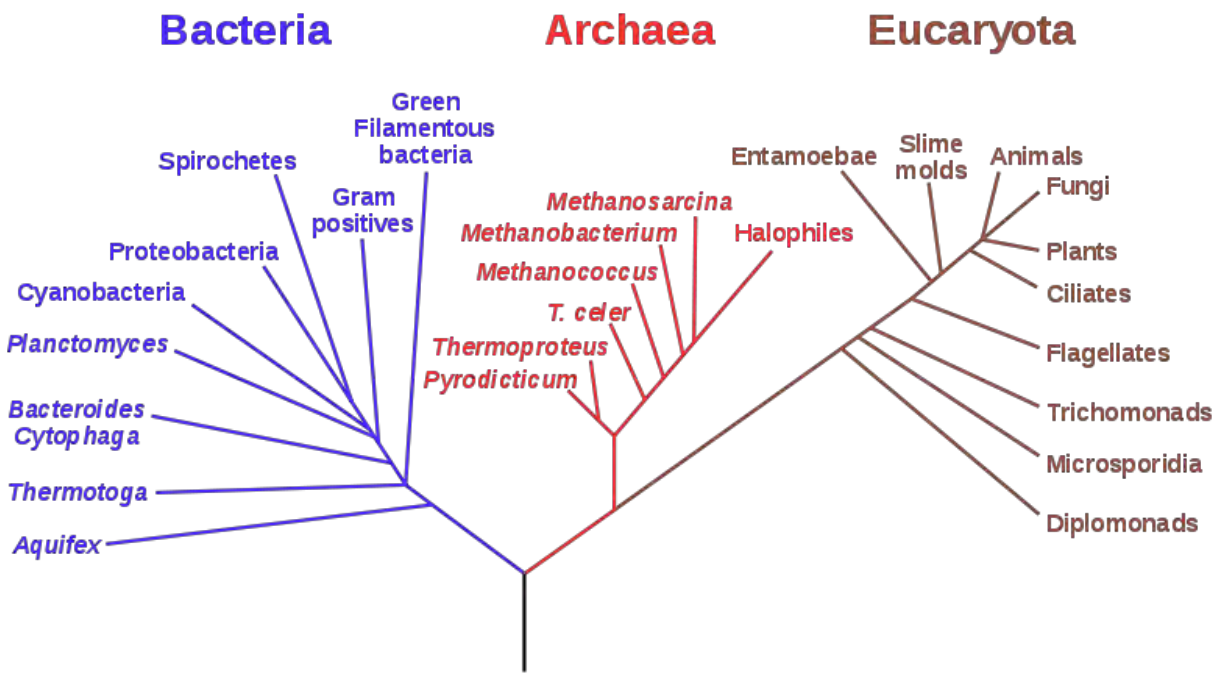


Figure 2.1 – Phylogenetic Tree[4]

2.2 – Bayesian phylogenetic inference

The Bayesian phylogenetic inference estimates a phylogenetic model Ψ= *(T,τ,ϑ)* of a given molecular sequence alignment D (D is the data). This phylogenetic model consists of three components:

- *T*: is a tree structure that represents the evolutionary patterns for the organism under study.
- τ: maps the divergence time along different lineages.
- ϑ: is a model of the molecular evolution that approximates how the chatacters at each site evolve over time along the tree.

In this model, the observed data D and the phylogenetic model Ψ are treated as random variables. Next, we introduce the formulas that Bayesian phylogenetic inference uses to calculate the conditional probabilities:

- $P(D|\Psi) = P(D|\Psi)P(\Psi)$: This is the joint distribution of the data.
- $P(\Psi_i|D) = \frac{P(D|\Psi_i)P(\Psi_i)}{\sum_j (P(D|\Psi_i)P(\Psi_i))}$: This is the posterior probability for a specific phylogenetic model Ψ_i, where $P(D|\Psi_i)$ is the likelihood, $P(\Psi_i)$ is the prior probability of the model and $\sum_j (P(D|\Psi_i)P(\Psi_i))$ is the unconditional probability of the data.
- $P(T_i|D) = \iint P(T_i, \tau, \theta|D) d\tau d\theta$: This is the marginal distribution which helps us to estimate the posterior probability of a specific phylogenetic tree.

2.3 – PBPI

Scientists that work on phylogenies always need an efficient way to analyze their data, because these analyses are often very expensive. One of the best ways to test their DNA sequence data is the programs that incorporate the MCMC-based Bayesian inference. Bayesian-based phylogenetic inference programs are a very good choice for this scope because these programs are usually faster and also have a better tree quality than other algorithms. However, analyzing a DNA sequence using Bayesian analyses is very computationally expensive, especially if the DNA sequence data requires several of memory space and therefore several months of computing time. A high performance application is needed in order to exploit the advantages of Bayesian plylogenetic inference.

PBPI[5] is a parallel implementation of Bayesian phylogenetic inference method for DNA sequence data and it combines Markov Chain Monte Carlo method with likelihood-based assessment of phylogenies. In order to achieve the desired efficiency, PBPI incorporates parallel processing and algorithmic improvements. The aim of PBPI is to estimate phylogenetic trees in order to quantify and visualize the relatedness between different species. PBPI takes DNA sequences of the examined species as input, constructs the tree where each node of the tree represents an organism and calculates the probabilities of the nodes of the tree according to the formulas of Bayesian phylogenetic inference.

There is an array of doubles in each node where the probabilities that show us the relatedness of the species of the tree are calculated. The length of this array depends on the input file. These arrays show the relatedness between the different species of the tree. The arrays of the leaves of the tree are initially filled with data but the arrays of the internal nodes needs computation. We also need the data of the arrays of the children of each node to compute the array of the corresponding node. For example in Figure 2.2 if we want to calculate the array of node 10 we must know the array of node 5 and have calculated the array of node 6.

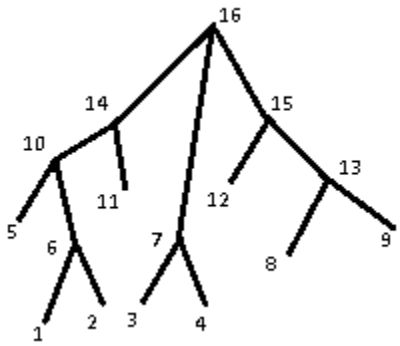


Figure 2.2 – Structure of a phylogenetic tree in PBPI

There were three challenges to overcome, when the PBPI was being designed. First, PBPI is founded on Markov Chain Monte Carlo method[6], which is a sequential method and current steps

depend on previous time steps. Second, it requires frequent I/O operations to store data drawn by the MCMC method. Third, even when a multiple-chain MCMC method is used, there is a limit on the degree of parallelism.

The computation time of PBPI depends on two factors:

- The length of the Markov Chains for approximating the posterior probability.
- The computation time needed for evaluating the likelihood values at each generation.

The algorithm proposed by Felsenstein[7] is used to calculate the corresponding likelihood. This algorithm traverses the phylogenetic tree in post order and computes the conditional probabilities for each internal node from the conditional probabilities of its children nodes. All this computation requires $o(N \cdot M \cdot S^2)$ multiplications, where M is the length of the alignment, N the number of taxa and S the number of possible states at each site.

A property of this algorithm that helps a lot in saving computation time is the likelihood local update. First the conditional probabilities for all the nodes of a phylogenetic tree (T) are calculated. When the calculations for the (T) are completed, a new phylogenetic tree (T') is created. The advantage is that the new tree (T') is constructed from (T) with partial changes, as we can see in Figure 2.3, so the algorithm has only to recompute the conditional probabilities for those nodes appearing in the back tracing path to the root nodes. In our example, we evaluate the likelihood of the tree (T) and then a new tree (T') is constructed. The right child of the node 13 has changed, so we have only to compute the conditional possibilities for the nodes 13, 15 and 16. This likelihood local update only requires $o(logN \cdot M \cdot S^2)$ multiplications improving significantly the performance.

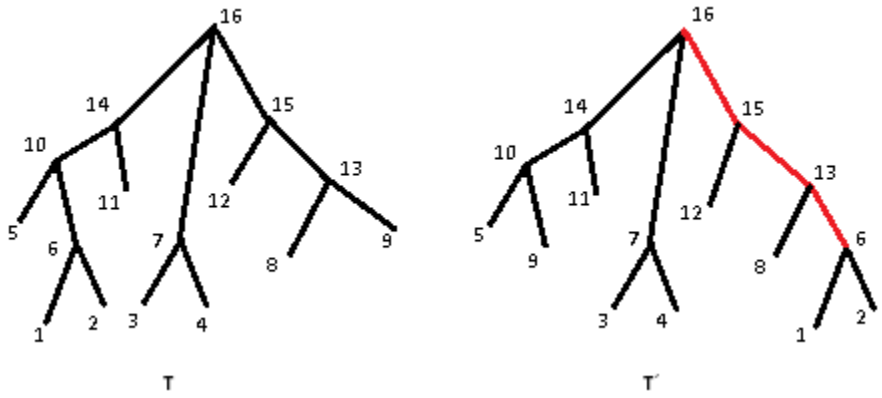


Figure 2.3 – The phylogenetic tree (T) and the proposal phylogenetic tree (T')

Nevertheless, it is not known if the proposed tree T' will be accepted. A method should be used to avoid frequent memory copies to save and restore the conditional probabilities. The method that is used is vector re-linking. The technique of vector re-linking is described as follows:

- A conditional probability vector is allocated for each internal node for both the current and the proposal tree.
- The proposal tree is obtained from the current tree and the vector links between the corresponding unaffected nodes in the current tree and proposal tree are swapped, keeping the remaining links untouched.
- If the proposal tree is accepted then the current tree and the proposal tree are swapped with all links unchanged, else all links for unaffected nodes are swapped back.

PBPI uses these optimizations to reduce computation time. But the biggest advantage of this application is the parallelism levels that it incorporates. PBPI uses the MPI interface to express and implement parallelism. MPI allows us to create many threads on non-common memory that may run concurrently reducing the computation time.

When a DNA sequence data (in our example we have 5 sequences) is given it is broken in smaller segments. The program analyzes the small segments and merges the result of them into a larger tree. Each segment can be analyzed concurrently and moreover several independent analyses can be performed in the same dataset to check the convergence of the chains. The partitioning of the data is shown in Figure 2.4.

Segment 1	
1	GATT
2	CAGG
3	TTAC
4	GGCG
5	TTAG

Segment 2	
1	GAAT
2	TTGT
3	CCAG
4	TGGT
5	GAAA

Segment 3	
1	TAAA
2	TGAA
3	GTCC
4	AACA
5	TGTC

Segment 4	
1	TGGC
2	AGAC
3	TGCG
4	ACGC
5	CTTA

Figure 2.4 – Partitioning DNA sequence data in segments

Except from these granularities of parallelism, there is another degree of parallelism at the Markov chain level as shown in Figure 2.5. Each sequence runs in a number of chains that are executed in parallel. The number of the chains is defined by the user. If there are enough processing elements in our system we may break the sequences in segments, run them in parallel, perform many independent analyses on the same dataset and execute the chains in parallel.

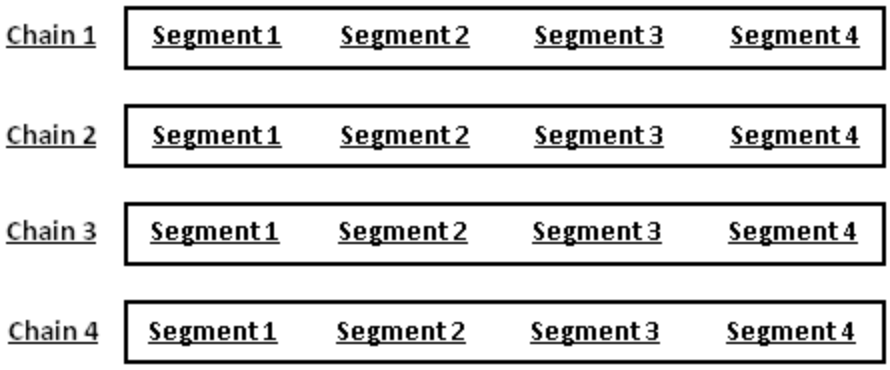


Figure 2.5 – Chain level parallelism

Our aim is to run this application on GPU. We will see the way it runs on a GPU and study if there is any performance improvement on the graphics card.

Chapter 3 – Parallel programming on graphics cards

3.1 – GPU computing

Graphics cards were initially designed as devices to process and visualize computer graphics. When in the late 1990's the hardware became programmable, NVIDIA took this chance and applied this new tendency to graphics cards. It also coined a new term called GPU in 1999 and the General Purpose (GPGPU) era was about to start.

At first it was too difficult to program on a GPU even for those who were programming on languages like OpenGL. They had to use triangles and polygons in order to represent their scientific calculations. The help of some researchers from Stanford, who saw the GPU as a stream processor, made programming on a GPU very easy. The term stream means a set of records that require similar computation and stream processing allows exploiting parallelism on applications.

Moreover, in 2003 a team led by Ian Buck unveiled Brook[8]. At last programmers were able to write programs on GPUs using C language with data-parallel constructs using concepts as kernels and streams. But the most important was that not only Brook programs were easier to write, they were about seven times faster than similar existing code. NVIDIA invited Ian Buck to join the company and they started to work on how to run C seamlessly on GPU. In 2006, NVIDIA revealed CUDA (Compute Unified Device Architecture), the first official solution of the company for supporting code execution on GPUs.

CUDA is a parallel computing platform and programming model that increases dramatically the computing performance, using the power of GPUs. The computations can be done on a GPU, while a few years back they should be only done on a CPU. A new scheme was created: the co-operation of a CPU and a GPU. Exploiting the GPUs' parallel throughput architecture the computation time decreased significantly. It is better to have many threads that run slowly in parallel than a single thread that run quickly. This is called GPGPU.

This concept was considered very useful and the computations can be finally done very quickly and nowadays CUDA is used extensively for accelerating computations in many application domains, such as:

- Bio-informatics where scientists accelerate their analyses.
- Pharmaceuticals where pharmaceuticals companies worldwide accelerate new drug discovery.
- Financial market.

The first NVIDIA GPU series that were designed for CUDA and helped in the sections said above were G8x series. Since then all the newest series support CUDA including GeForce, Tesla and Quadro lines. Figure 3.1 depicts the typical pattern that characterizes a CUDA application:

1. The processing data are copied on the GPU‘s global memory.
2. The CPU initiates the processing on the GPU.
3. The GPU executes the code in parallel.
4. The result is copied back to RAM.

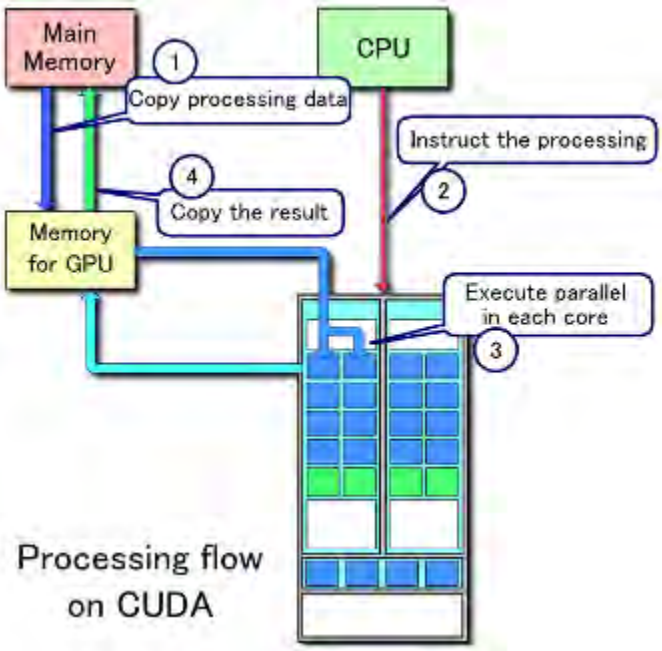


Figure 3.1 – The CUDA model[9]

3.2 - NVIDIA's Fermi architecture

It is very important to realize how a GPU works at a low level, before starting programming it. We will discuss the Fermi architecture[10] and specifically this discussion is based on the GeForce GTX480.

The main building block of the Fermi architecture is shown in Figure 3.2. This level is called CUDA core. CUDA cores are very simple. They just have a pipelined floating-point unit (FPU), a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. They do not have their own register file or L1 caches like CPUs. They do not even have a load or store unit to access memory. These cores are not designed to execute efficiently single-threaded codes, but rather to work in parallel and provide high computational throughput.

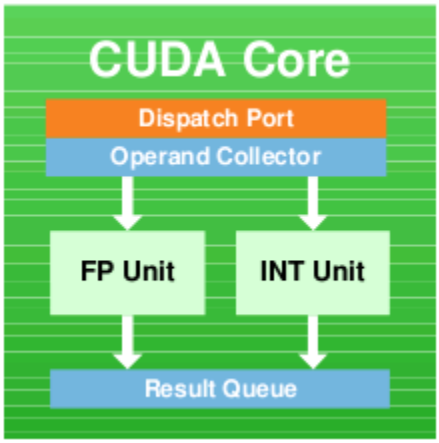


Figure 3.2 – A CUDA Core

Multiple CUDA cores are aggregated at a higher level as streaming multiprocessors. A streaming multiprocessor is depicted in Figure 3.3. Each multiprocessor has 32 CUDA cores and all these cores share the registers, the caches, the local memory, and the load/store units. The special function units (SFUs) handle complex math operations, such as square roots, reciprocals, sines, and cosines. Now it is understandable why a CUDA core does not have caches and load/store units. These 32 cores work as a group, share these resources and are designed to work simultaneously on 32 instructions from a bundle of 32 threads, which NVIDIA calls a warp. Moreover two different warps may run on the same multiprocessor concurrently and this is called dual-issue.

Before continuing to the next level, we have to say how the streaming-multiprocessor works. The scheduler can now issue two instructions per clock, unlike the previous architectures. However, these two instructions cannot be issued by the same thread but by different warps. A multiprocessor can manage up to 48 warps and each warp, as we have previously said, has 32 threads. So, a

multiprocessor can manage 1536 threads and in the case of GTX480, which has 15 streaming multiprocessors, we can have 23.040 parallel threads. We cannot execute all these threads concurrently, but we need to have as many as possible activated threads. When a warp of threads is accessing the memory, the thread scheduler simultaneously switches to another ready warp for executions, hiding this access. At any time, there are executed concurrently the maximum number of threads for the current device. For example, in GTX480 this number is 480 threads (15 streaming multiprocessors * 32 CUDA cores each streaming multiprocessor) and when they have finished their execution, 480 different instructions can be executed in the next clock cycle.

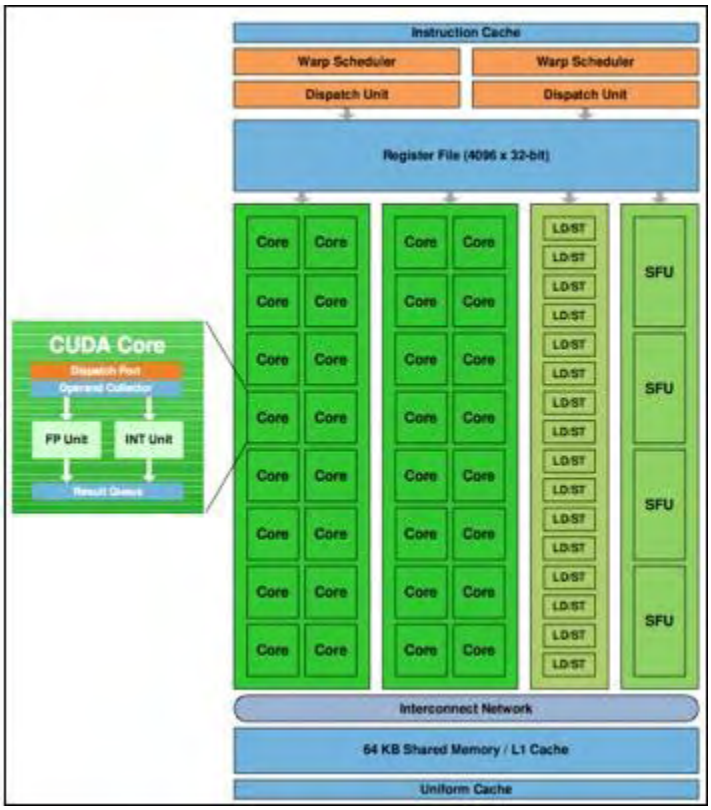


Figure 3.3 – A streaming multiprocessor

The last and highest level is shown in Figure 3.4. We can clearly see the streaming multiprocessors, and the unified L2 Cache of 768KB, which is shared by them. We can also see the six 64-bit DRAM chipsets of 256MB each, also known as global memory, the host interface (PCI Express) and the GigaThread hardware thread scheduler. This scheduler is responsible for thousands of simultaneous threads and very quick context switches between graphics and compute applications.

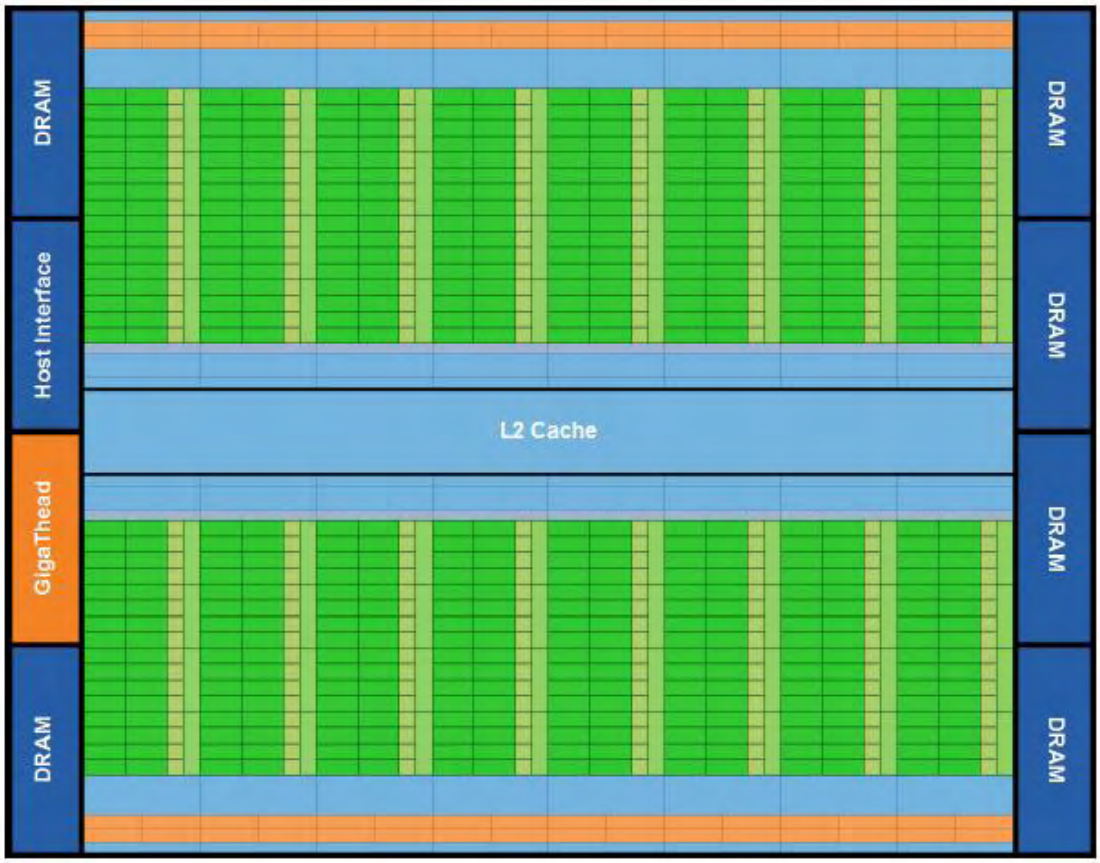


Figure 3.4 – Fermi Block Architecture Diagram

Understanding the way a GPU works it is obvious that is totally different form a CPU. But, there is also another big difference between them. A CPU can communicate straight to the system‘s main memory (RAM) through an integrated memory controller (IMC). A GPU can also communicate with the system‘s main memory, but there is a big problem. A GPU reaches the main memory over PCI-express (PCIe), which has a high latency. A more efficient way is needed, in order not to spend too much time transferring data over PCIe.

In Figure 3.5 we illustrate the memory hierarchy, which saves us a lot of time when accessing memory. The highest level is the DRAM (global memory), which is more than 1GB. We transfer the data that are needed to this memory from RAM and we just use DRAM and not the main memory of our system. We have also mentioned that global memory consists of six 64-bit interfaces, which are installed on the PCB of the graphics card and not in the chipset. This memory is fast enough using DDR5 technology with low latencies achieving high performance in today’s top-level graphics cards. Also, this memory is shared by all the streaming multiprocessors. Although global memory is fast enough, the fact that is installed on the PCB and not in the chipset, makes its accessing a little slow.

For this reason, designers have installed some memory in the chipset as well. This is the next level, it is called unified L2 cache memory and it is also shared by all the streaming multiprocessors. The position that is installed is very important. So, it is put among all the multiprocessors and it is easily accessible by all them. L2 cache memory's capacity is only 768KB, significantly smaller than the global memory, but it is very faster than the global memory. L2 is usually used to cache data of a small capacity that are frequently used by the multiprocessors, allowing us to avoid the "trip" to the global memory.

Even though we can say that L2 is fast enough for the needs of a GPU, there is yet another level of memory. This level called L1 cache is installed in each streaming multiprocessor and its capacity is 64KB. It is lowest and the fastest level reducing latency and increasing bandwidth. The 32 cores of each multiprocessor share this level, but cores from one multiprocessor cannot access L1 cache memory from another multiprocessor. It is local to each 32-core streaming multiprocessor and shared by all those cores.

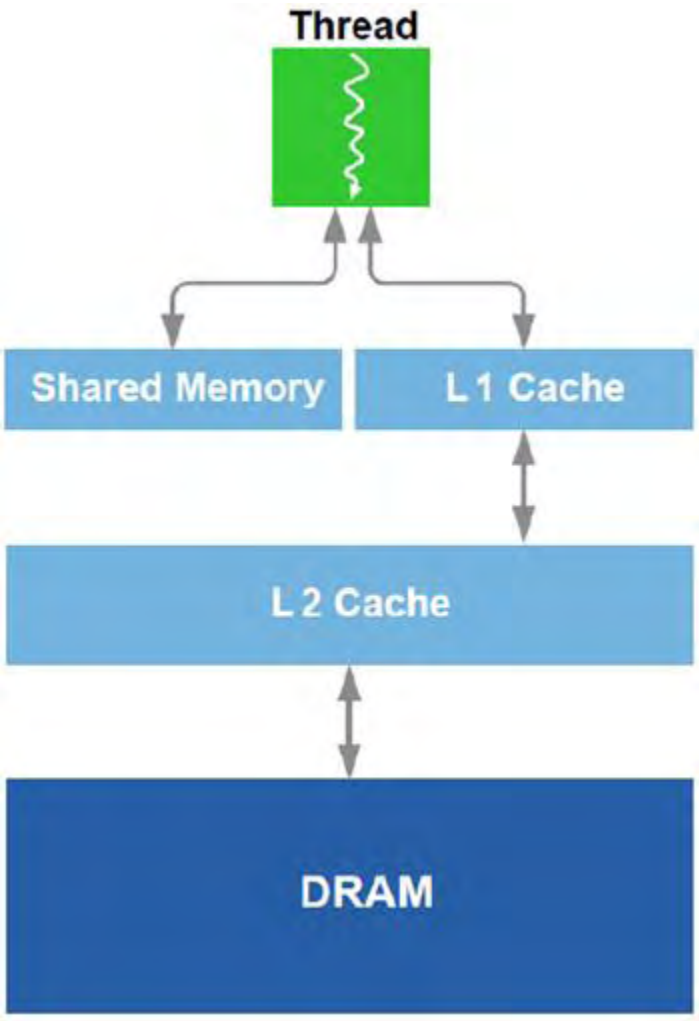


Figure 3.5 – Fermi Memory Hierarchy

3.3 – Differences between a CPU and a GPU

A CPU‘s architecture and specifically the intel i7 860‘s die[11], is shown in Figure 3.6.

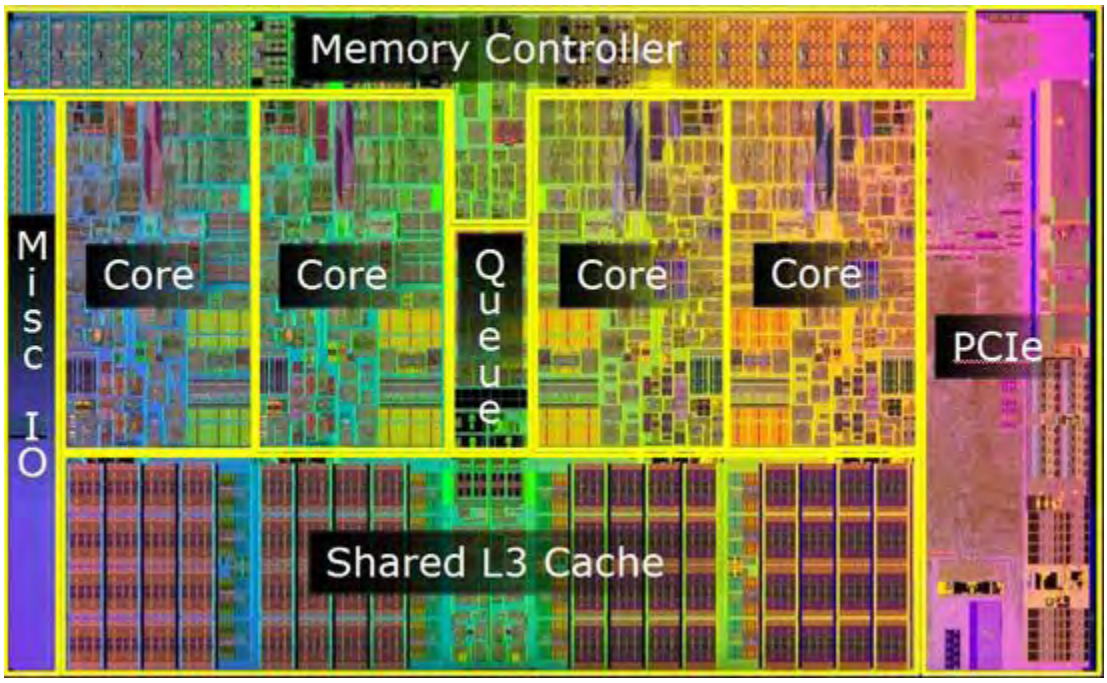


Figure 3.6 – The die of an intel core i7 860

First, there are 4 cores with HyperThreading technology. We can execute 4 threads concurrently and due to HyperThreading we can sometimes execute even 8 threads concurrently if the 2 threads of a core have not to share the same resources. So the maximum number of threads that can be executed concurrently on this CPU is 8. Unlike a GTX480 can have 23,040 active threads because a GPU is good for throughput via a high degree of parallelism. The 480 of these threads are executed concurrently.

Second, there is a difference in the control units of these two architectures. The control unit orders which instructions will be executed on a core. The control unit is very important on a CPU, because each thread may execute different things in each time. Unlike a GPU does not have a complex control unit. The threads are executed in warps in parallel and they do the same thing. We do not have to control each core but we control a group of 32 threads.

Third, there is also a difference in caches. The intel core i7 860 has 3 levels of caches. The lowest level is the 4*64KB L1 Cache, the second is the 4*256KB L2 Cache and the third is the 8MB shared by all cores L3 Cache. CPU uses large caches to hide memory access latency. In comparison with the caches of GTX480, we see that the GTX480 has one less level of cache than the i7 860 and that the higher level in GTX480 is too small compared to the i7 860 ‘s. GPUs may have small caches, but they use fast context-

switching to overlap memory access latency with useful computations. We just use a GPU for computations in parallel, and it is more important to have many cores than big capacities of cache memory.

Chapter 4 – Using CUDA on PBPI

4.1 – Introduction and methodology

We used the back218_L10000np256.nex input file to count the execution time of PBPI. This file contains the DNA sequences of several organisms. These DNA sequences are 10,000 characters length. We executed PBPI with the following options:

- 4 Markov Chains
- 50,000 repeats for each chain
- 1,000 step for each repeat
- 1 run of the PBPI

Our benchmark rig consists of:

- An intel core i7 860
- 8GB RAM
- GeForce GTX480

The core i7 860 is clocked at 2.8GHz (133MHz (bclk) * 21 (multiplier)). The core i7 860 incorporates the turbo boost technology[12]. When an application is executed the core i7 860 automatically changes its clocks to 3.46GHz changing its multiplier to 26 (133MHz (bclk) * 26 (multiplier)).

The GTX480's specifications are:

- Core clock: 700MHz
- Shader clock: 1400MHz
- Memory Clock: 3700MHz DDR5
- Memory capacity: 1536MB

The operating system was used was Kubuntu 11.10. The compiler that was used for the PBPI implemented on CPU is icc and the flags of the compiler were -03 -fomit-frame-pointer. The compiler for the GPU was the nvcc which is based on gcc and not on icc and the flags were the –arch=sm_21. We used the nvvc to complie the .cu files but we made the link with the icc.

The computation time and the total time of this application executed on the specific CPU is shown in the following graph:

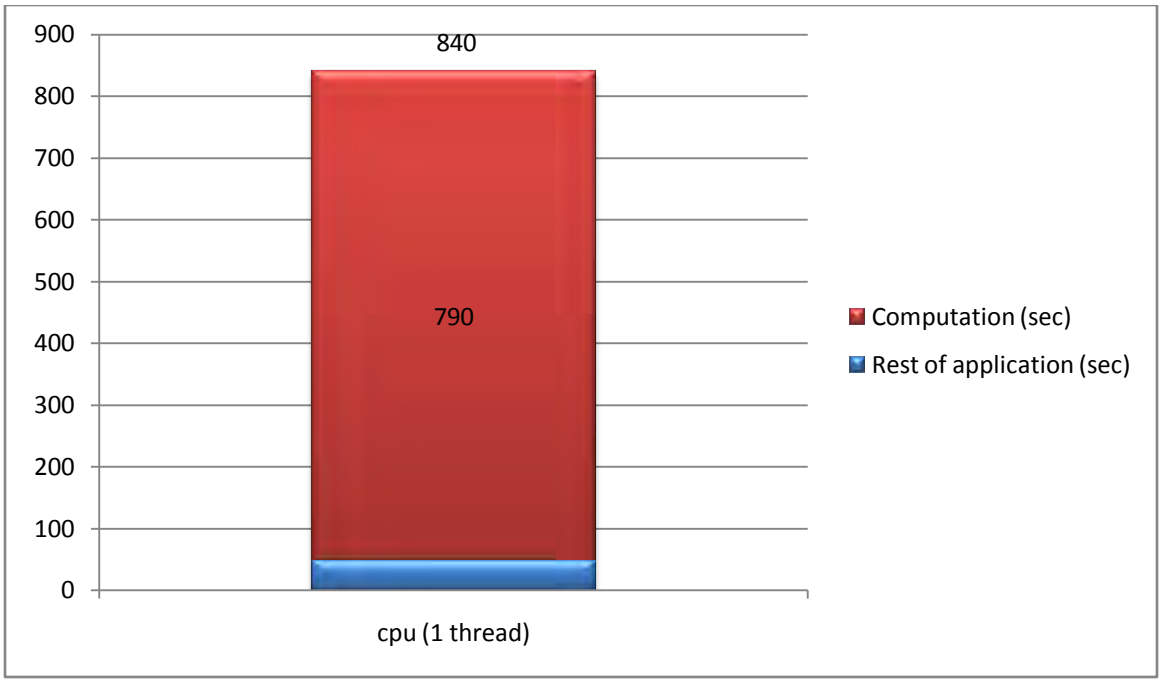


Figure 4.1 – PBPI on single threaded CPU

4.2 – Profiling PBPI

Profiling PBPI shows us which function is the most computationally intensive. We used Intel's Vtune[13] to do so. Profiling PBPI with Vtune we saw that the most time consuming function is the ComputeNodeLikeData. This function estimates the arrays in each node of the tree. The length of these arrays called siteLike is depended on the input file, but they are usually quite lengthy. The length of the arrays for the specific input set was in the order of 40000 items

The computation of different elements of each array is independent and therefore the value of different elements can be computed in parallel. It is obvious that we cannot create 40,000 threads on CPU to calculate this array in parallel, but this a job that can be easily done in GPU. Using a GPU for these calculations we can exploit parallelism and reduce significantly the computation of these arrays.

4.3 – Working-around recursive function calls

A GPU does not support recursion. The main reason is that each thread cannot have its individual stack. Millions of threads are created when someone uses a GPU for computations and there is not enough memory to have a stack for all those threads. There are two ways to solve this problem:

- We can execute the recursion on CPU and invoke the kernel on the GPU in each node.

- We can execute the recursion on CPU, record the sequence of the nodes and invoke the kernel on the GPU one time only.

We opted for the second solution, because multiple kernel invocations would penalize performance. This technique is generally applicable if the recursion is only used to traverse a linked data structure and the traversal order is not dependent on computations.

4.4 – The PBPI implementation on GPU

The most time consuming function is the ComputeNodeLikeData. This is a recursive function. In each step of this recursive is computed the siteLike array of a node of the tree. For this computation we need to know the corresponding siteLike array of the children of that node.

We run this function on CPU because GPU does not support recursion. When we reach a node that needs computation we call five times the cudaMalloc function. CudaMalloc is a function that allocates memory on the GPU's memory. We need to call two times the cudaMalloc function to allocate memory on the GPU 's DRAM for the siteLike arrays of the left and the right child, two times for the trprob arrays of the left and the right child and one time for the siteLike array for the node we have reached in order to save our results. Then we call the cudaMemcpy functions 4 times to transfers the arrays of the two children on the GPU's memory.

When our data are copied on GPU's memory the kernel for the GPU is invocated. In our example 40,000 threads are running exploiting parallelism on GPU. Each thread on GPU is running much slower than a single thread on the CPU but the fact that they are all parallel to each other gives us a greater performance for this computation than a single thread on a CPU.

When our computations are done we call the cudaMemcpy one time and transfer the siteLike array of that node to the RAM. Then we call the cudaFree function five times to free memory from all the above arrays and we continue the recursive access of the tree. In this way are computed all the siteLike arrays of the tree and we have computed all the siteLike arrays of a tree we go to the next tree. All this procedure is show in Figure 4.2 for the node 13 of a tree.

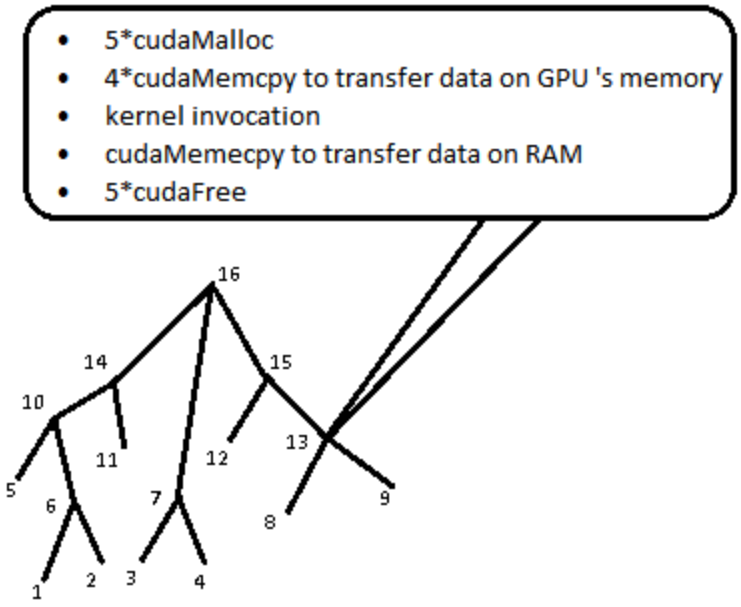


Figure 4.2 – Example of node 13 executed on GPU

Our implementation consists of 3 basic components: the memory allocations, the memory transfers and the computation time on GPU. It is crucial to see analyze these 3 components when programming on a GPU in order to optimize our performance. These 3 components are shown in the graph of Figure 4.3.

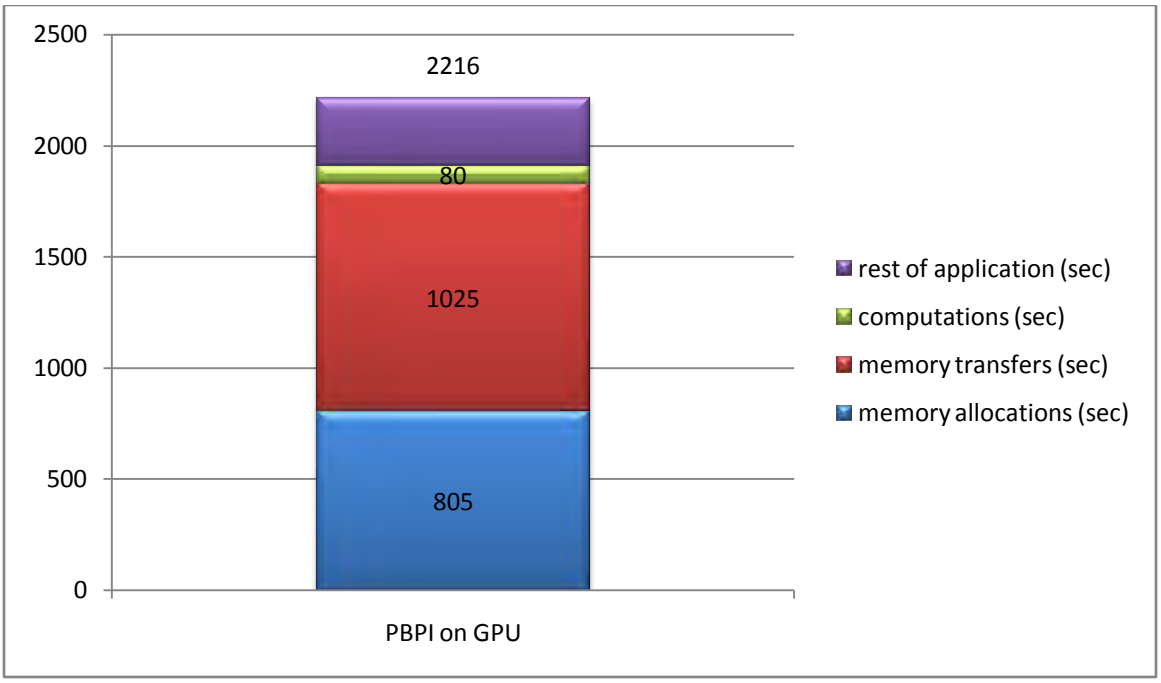


Figure 4.3 – PBPI executed on GPU

Using a GPU for exploiting parallelism on computations gives us a great performance in contrast with a single threaded CPU. On GPU took 80 seconds for the computations while on CPU needs 790 seconds. The total time of our execution is more than the CPU's because of the memory allocations and transfers. The memory allocations need 805 seconds and the memory transfers 1025 seconds.

4.5 – Parallel execution on GPU

Our goal is to run all the items of the array of each node in parallel on GPU. The big advantage of PBPI is that all items of each array are independent to each others. So, we create as many threads as the items of these arrays. When we invocate the kernel these threads are executed in parallel reducing the computation time.

The arrays of the back218_L10000np256.nex input file have 40,000 items. We can simply create 40,000 threads to compute our arrays in this way:

```
dim3 dimBlock(4,100);
dim3 dimGrid(1,100);
```

We now have 40,000 threads running to execute our computations. But this is not very efficient. The threads in a multiprocessor are organized in warps. Each warp has 32 threads. The 40,000 threads cannot be divided with the number 32 and multiprocessors do not always have group of threads of 32

threads each. The solution is that the number of threads that is declared in each block should be a pow of 2.

For example we may use 256 (4*64) threads per block and 157 blocks. This configuration seems to be fine because the number of the threads per block is a pow of 2. However this configuration is not very efficient because we have only a few blocks. We want to have many blocks because when a warp is accessing the memory we want to swap to another warp that waits for computation. On the other hand we should not have excessively too many blocks. Having a few threads per block kills the performance too. We should find the suitable configuration and this can be done by testing different configurations. The best configuration for PBPI is:

```
dim3 dimBlock(4,32);
dim3 dimGrid(1,313);
```

We now have 128 threads in each block and the threads are organized in warps. Also we divide the 40,000 with 128 and the result is 312.5. We round the 312.5 to the next integer and we have 313 blocks. Now we have more than 40,000 threads. We use the 40,000 threads for computation and the remaining threads do nothing. It may seem that these useless threads may give us a bad performance, but this is not actually true. These threads run in parallel and they cost us almost nothing. The performance efficiency is show in the graph of Figure 4.4.

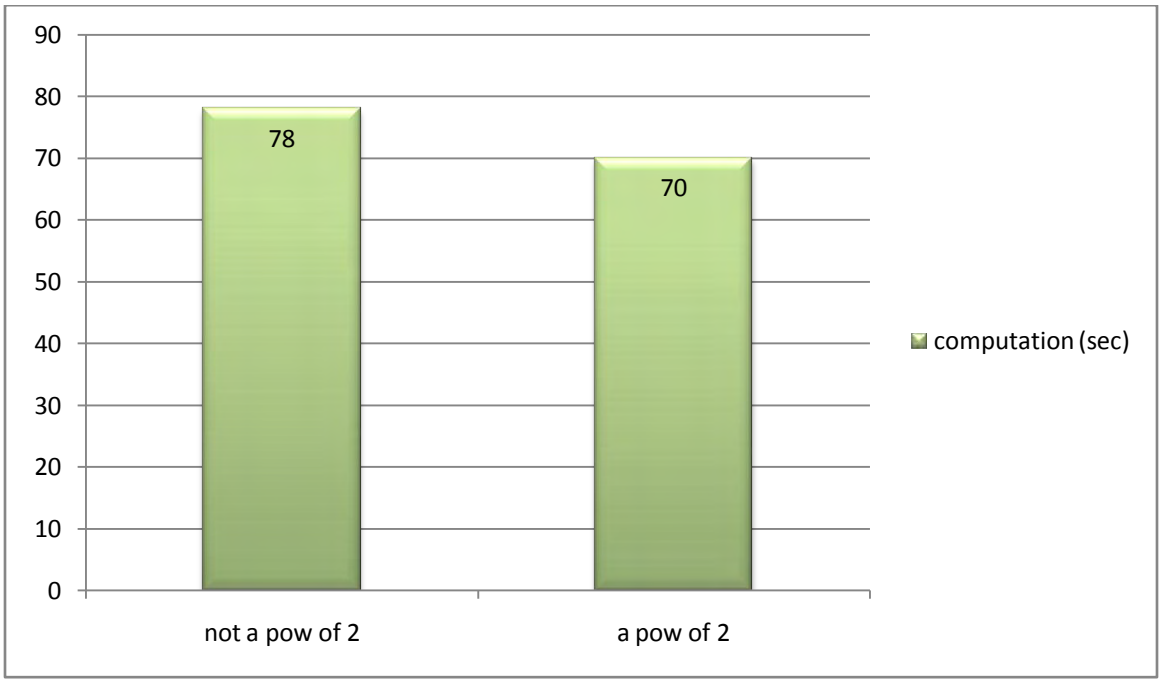


Figure 4.4 – Organizing threads

4.6 – Memory allocations optimization

Our first optimization concerns the memory allocations on GPU. Our total execution time of time was 2216 seconds. The memory allocations time was 805 seconds which is a big part of our total time.

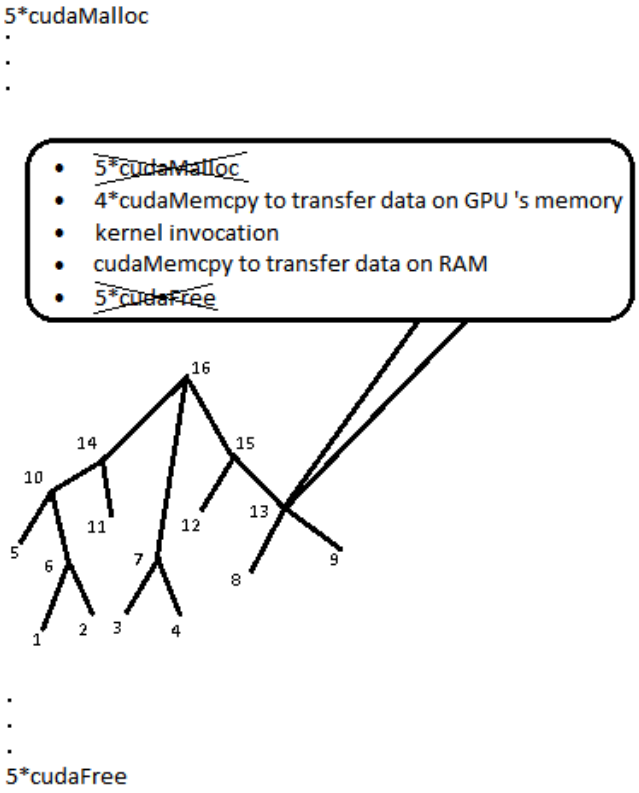


Figure 4.5 – Elimination of memory allocations

All the internal nodes of the trees need the same quantity of memory for the computations. It is not wise to allocate and free that memory for every node. We can allocate these areas in the beginning of the application and reuse them for all the internal nodes. For example we may transfer to GPU the data of the node 8 and 9. These data are needed for the node 13. We can compute the array of node 13, save the result and return it to the RAM. Next we can compute the node 15 and use the same areas in our memory to transfer our data. The node 13 is done and we do not need the data of node 8 and 9. So we can overwrite the areas with these data, saving the new required data for the new computation. We do that for the entire tree and it is shown in Figure 4.5.

Our new total execution time is 1346 seconds is shown in graph 4.6. The memory allocations time is almost eliminated (0.1 seconds instead of 805) but the memory transfers are still a big problem.

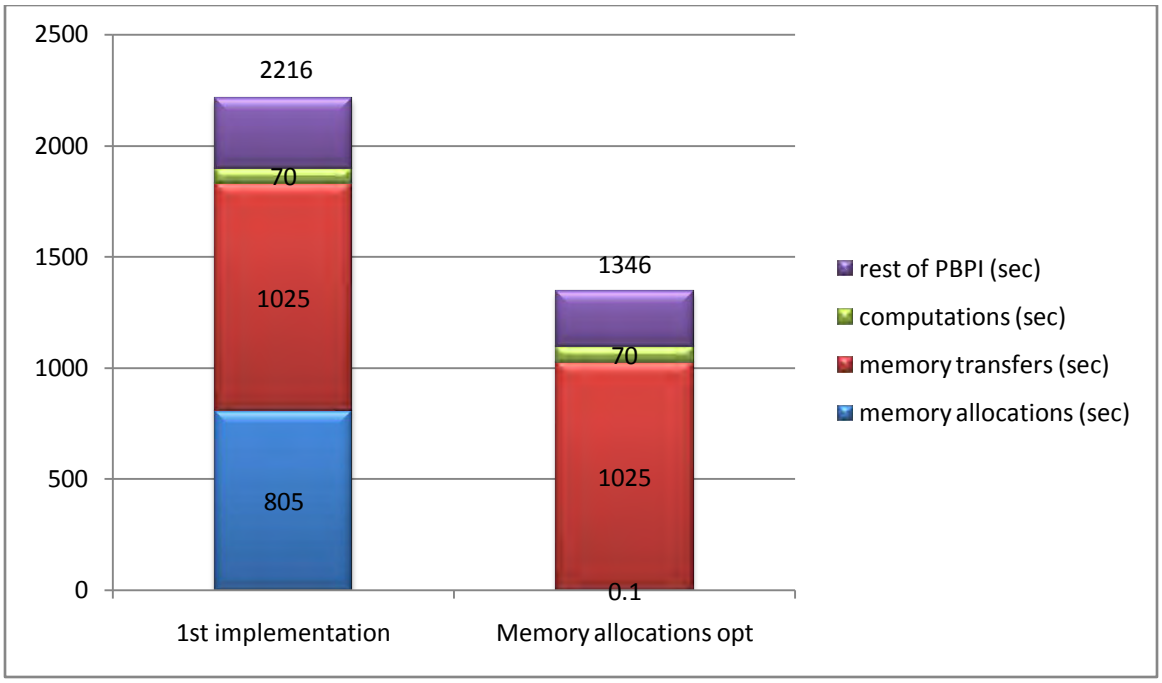


Figure 4.6 – PBPI on GPU

4.7 – Asynchronous memory transfers

The total time for all the memory transfers is 1025 seconds. We want to either eliminate them or overlap them with useful computation. First we used asynchronous copies from and to GPU in order to overlap them with computations. Asynchronous memory copies are a feature that was added in the latest NVIDIA's graphics cards and allows us to execute the memory copies in parallel with other instructions of our code. We do not wait the function cudaMemcpy to return saving a lot of time for our application.

Asynchronous memory copies are used in streams. The concept of streams is very simple. First we put in the same stream the memory copies and the kernel invocation that are depended on each other. Instructions within each stream will execute in-order and each instruction in a stream waits for a previous instruction in the same stream to be executed. On the other hand instructions from different streams may execute in arbitrary order. For example in Figure 4.7 we copy the data and execute the kernel in stream0 for operation A and we can copy the data for operation B in stream1. The kernel of stream0 is executed in the same time with the memory transfers of stream1.

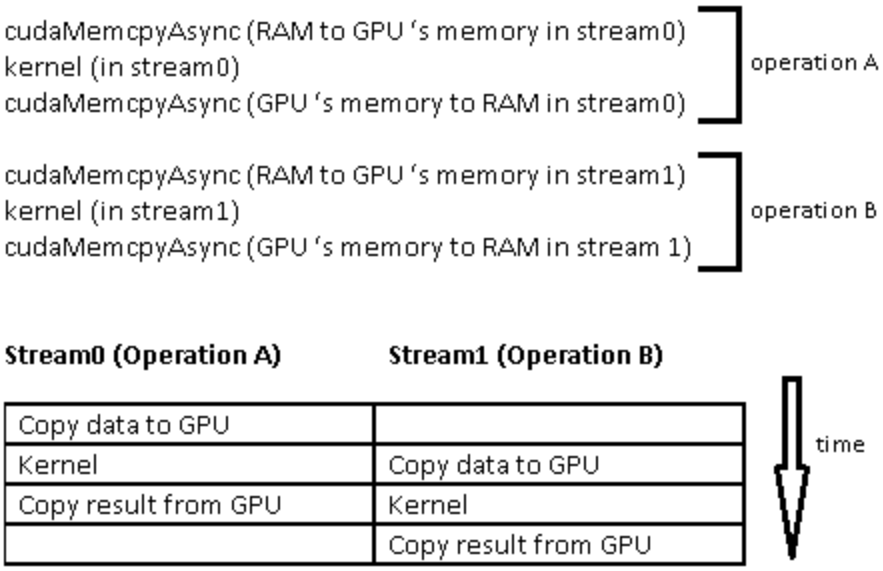


Figure 4.7 – Usage of streams

In our application we copy the data of a node's children asynchronously, execute the kernel for the computation of that node in the same stream with the memory transfers of the children's data and finally we copy the result in the RAM in the same stream with the previous operations. The problem in calculating the arrays for the nodes is that the parent node is depended on its children's nodes. We cannot put them in different streams because we must wait for the children to calculate their arrays. For example as shown in Figure 4.8 we cannot put node 15 in a different stream from node 13. When we want to calculate the node 13 we transfer the data from node 8 and 9, we start the calculations for node 13 but we cannot do anything for node 15 because node 13 must finish its calculations first. So, we put the entire tree in the same stream because the parent nodes are depended on their children nodes. We may not have many streams to overlap the instructions of different streams but we can overlap the instructions of the stream with other instructions that are not related to this stream.

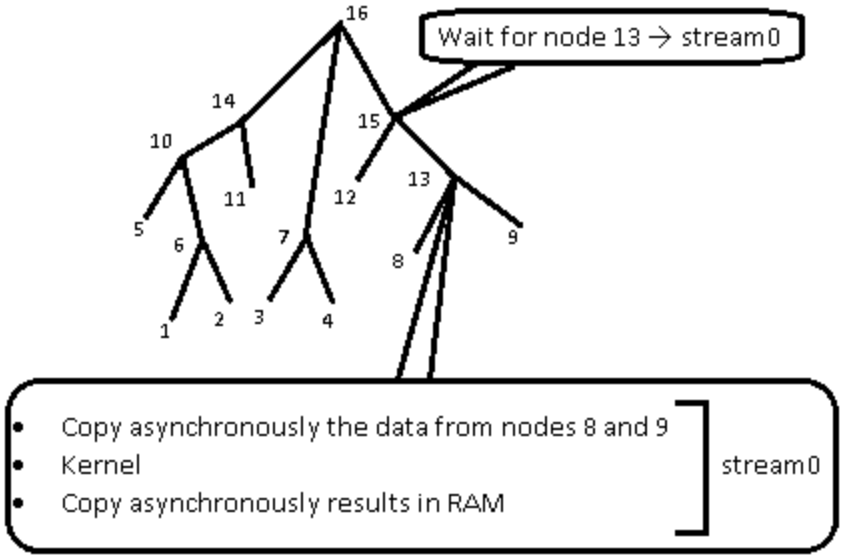


Figure 4.8 – Asynchronous copies in a single stream

Using asynchronous memory transfers helped to decrease the total execution time of memory transfers to 825 seconds from 1025 as we in graph 4.9. We had an improvement in memory transfers but they still have a great share of our total time.

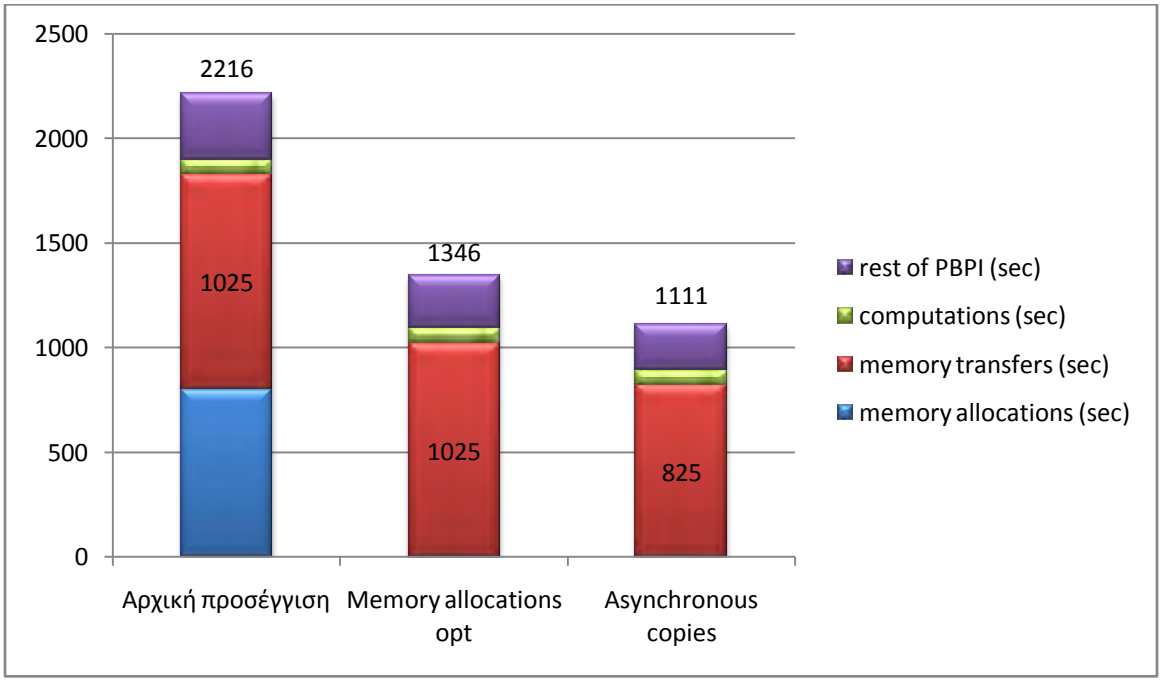


Figure 4.9 – Execution on GPU

4.8 – One kernel invocation for each tree

In this optimization we will implement the technique discussed in chapter 4.3, invoking the kernel on GPU once for each tree. To do so we need to create a struct on GPU that will keep the right order for accessing the tree. Each item of the struct contains the data of a node and the nodes are saved in the order they are accessed on the tree. Then we will access the struct repeatedly on GPU invoking the kernel once as shown in Figure 4.10. One more benefit of this implementation is that we keep on GPU the computed data of a node that are needed by its parent node. We do not transfer them back to RAM and then to the GPU again. It is better to have GPU to GPU copies than RAM to GPU copies via PCIe. The NVIDIA's bandwidth test shows us the reason:

- Copying data via PCIe from RAM to GPU's DRAM: 5500MB/s.
- GPU to GPU copies: 150000MB/s.

The implementation of this is very simple. We access the tree recursively on CPU. When we reach a leaf of the tree we transfer the 40,000 item array of the node on the corresponding position of our struct. For the internal nodes we do not have to transfer these arrays, as they are initially filled with zeros. We just record the id of their children of the internal nodes. So when we reach an internal node for computations, we just use the data of its children to compute the array of this node.

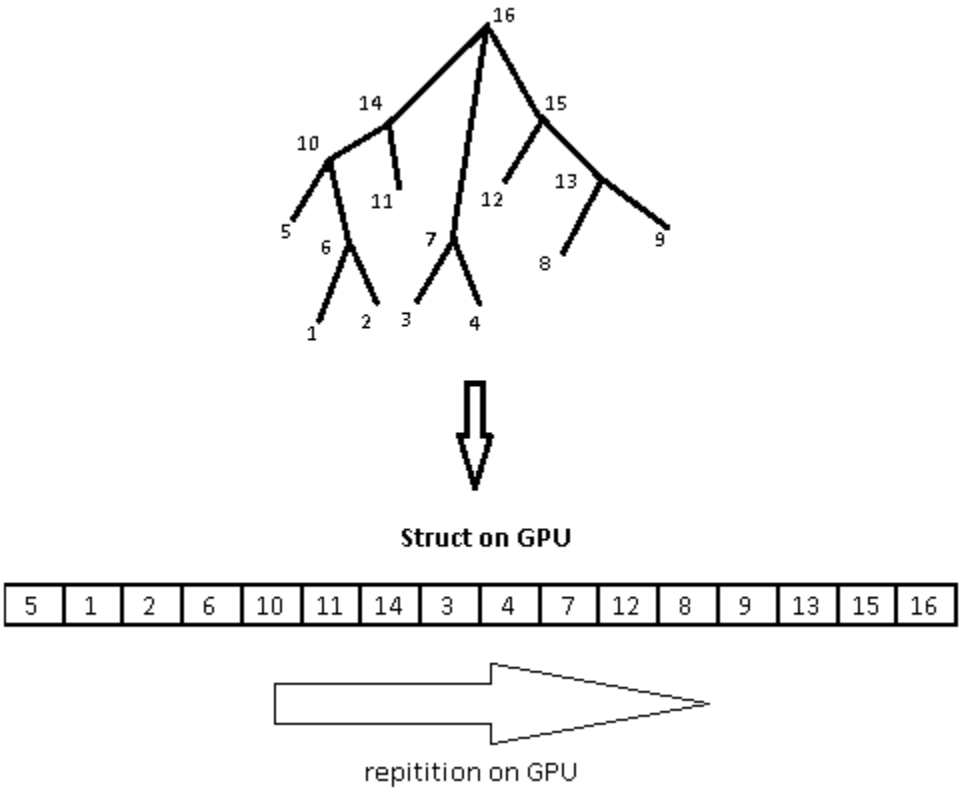


Figure 4.10 – Accessing the tree repeatedly on GPU

The new total execution time is 764 seconds as shown in graph 4.11. The memory copies time decreased at 633 seconds from 825 because we have all the data of a tree on GPU's DRAM.

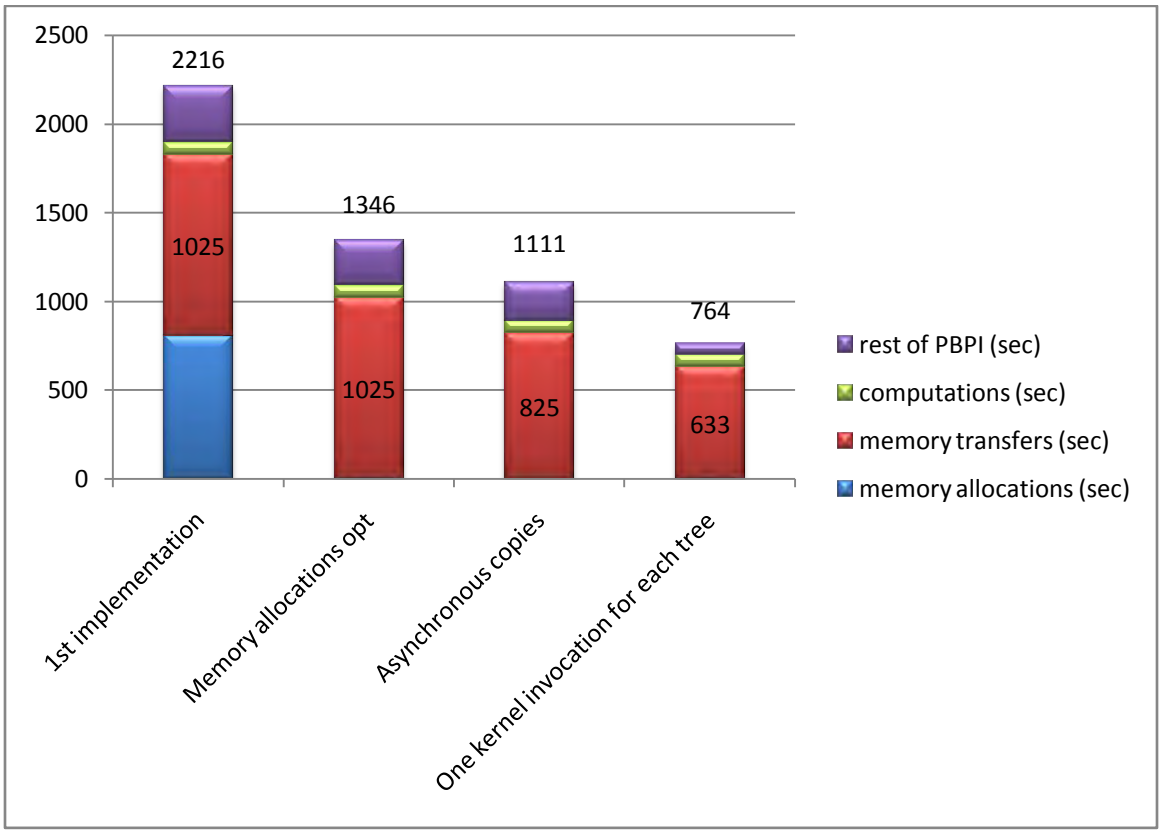


Figure 4.11 – Execution on GPU

4.9 – Memory transfers aggregation

Even though the total execution time of PBPI and the memory transfers time have decreased, the data copies still affect negatively the performance. The reason that our data copies last so much is that we copy the data of every node of every tree. Not only we transfer too many megabytes of data on GPU's memory, but we also transfer them in segments.

The aim of GPU programming is to manage to transfer the data we need in one copy at the beginning of the application. It is preferred one large transfer than many smaller because we pay the transfer latency fewer times and exploit the bandwidth better. This is not always easy, especially if our data are scattered in the memory. Although that does not really affect a CPU, it is always a big problem for a GPU and we usually need to do some extra work on that creating our own memory manager.

This technique is shown in Figure 4.12. If our data are scattered in memory as RAM-1 we save them in a continuous area in the memory as RAM-2. First we allocate on RAM the needed continuous area. Then we save the first data segment at the beginning of that area. When the first data segment is saved we put the pointer to the end of that segment. So, when we save the second segment, it will be saved next to the first one. We do that until all the data are saved and we finally have them in a continuous area in RAM.

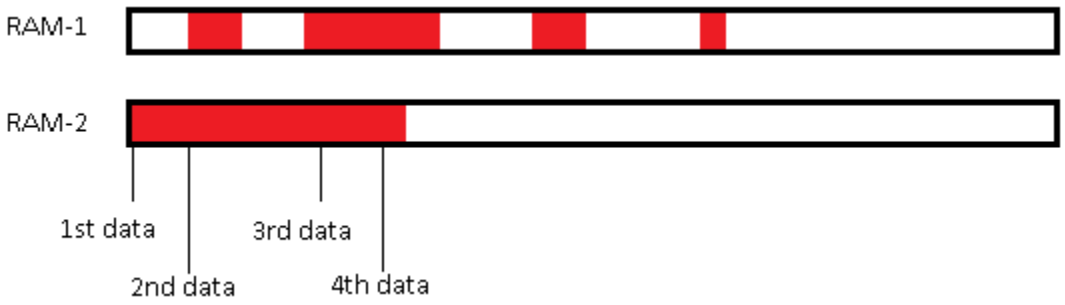


Figure 4.12 –Organizing data in RAM

Having all the 40,000 item arrays in one piece in system's memory, we allocate the total amount of memory needed on GPU and we transfer the data on GPU's memory at the beginning of the application once. Then we access the trees recursively on CPU but we do not any more transfer the 40,000 item arrays on GPU's memory for each node. We only pass an argument of the position of the data we need on GPU for each node, do the computations and return the results back to RAM. We no longer have to transfer any 40,000 item array of the next tree, because were initially copied on GPU's memory and we make all the changes there.

We can see the new total execution time in graph 4.13. PBPI now needs 398 seconds for the data copies instead of 633. The remarkable of this technique is that we do not need to use asynchronous copy for that one memory transfer. We do not copy that area asynchronously and our memory transfers are faster than our previous implementations with asynchronous copies. Also we now pay the latency time once at the beginning of the code without spending time paying for every node for each 40,000 item array.

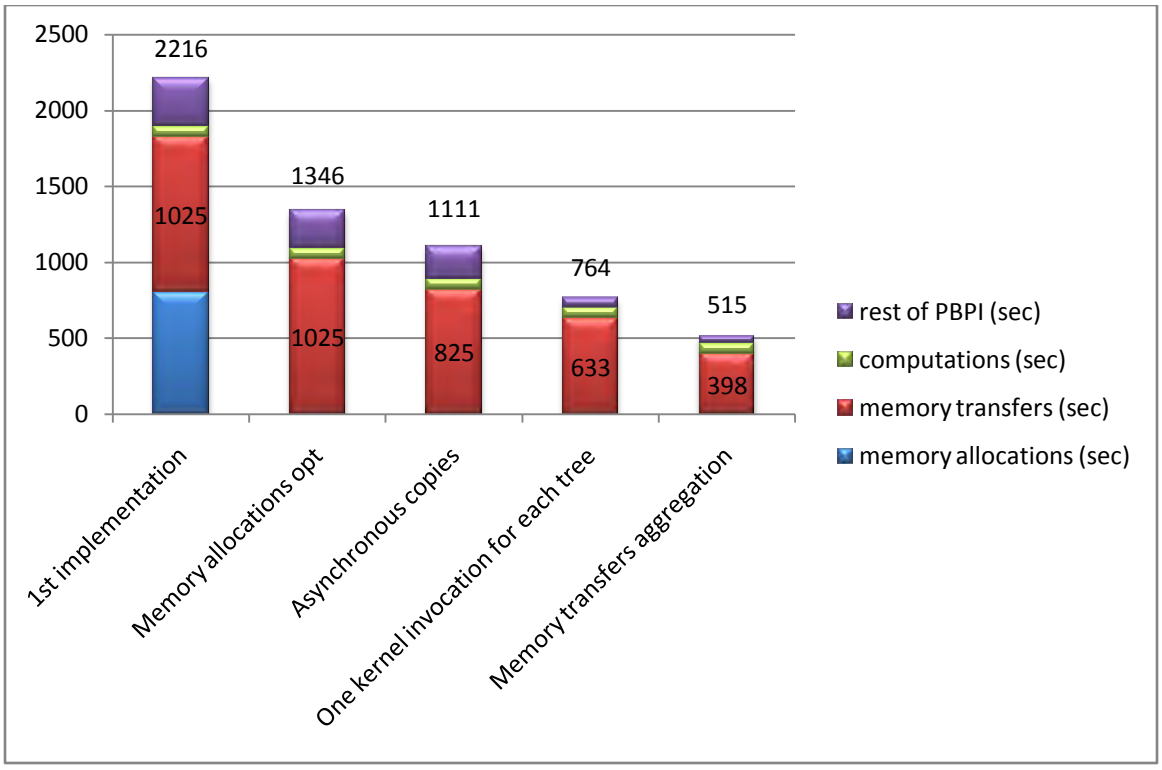


Figure 4.13 – Execution on GPU

4.10 – Minimization of results transferred to the CPU

Transferring data on GPU at once is the ideal when working on GPU and using on PBPI decreased the memory transfers time. But the copies of the results back to RAM still affects PBPI. We copy back to RAM each array of any internal node. These copies cost too much hurting our performance.

Having all the data on GPU's memory we do not transfer data for each node apparently. Furthermore we now work only on GPU's memory and make all the changes there. No changes happen on RAM anymore. But we still return all the result of every node back to RAM. Studying the way the tree works we see that we calculate the arrays of each node in order to reach the root of the tree to calculate root's array. PBPI takes the result of the root and prints in a file. This is the wanted result. Not using RAM anymore and needing only the root's result we do not have to return all the arrays of all nodes of the tree back to RAM. As we see in Figure 4.14 we can just compute the data for each internal node such as 13, save the new results on GPU's memory and copy back to RAM the array of node 16 only.

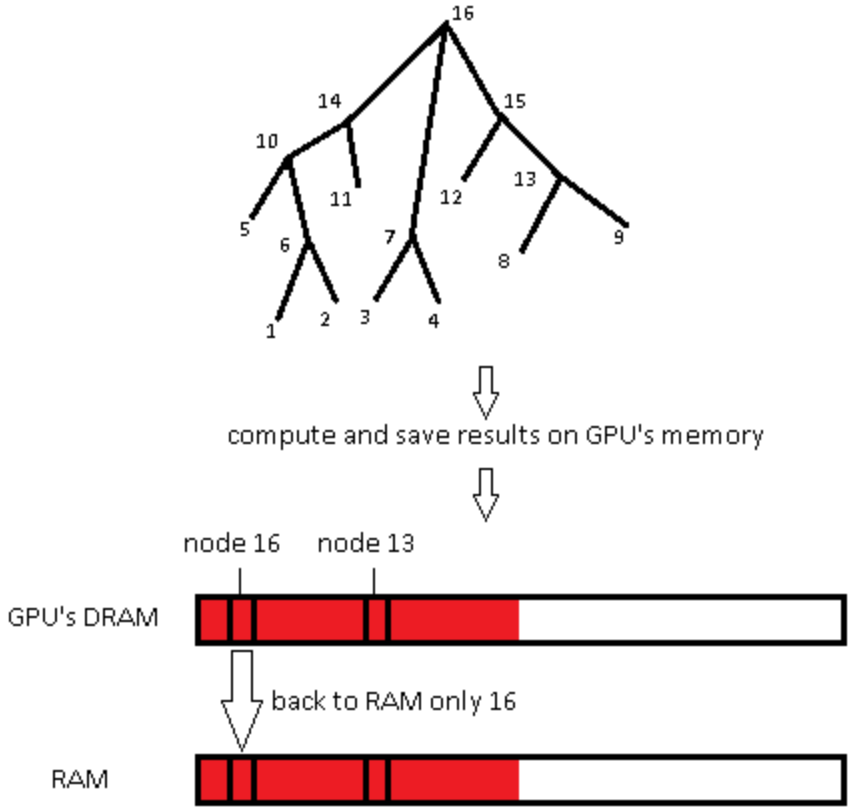


Figure 4.14 – one transfer back to RAM

Making this optimization we saved a lot of time. Now the memory transfers take 27 seconds only instead of 398 as we see in graph 4.15. We have avoided the transfers from GPU's memory to RAM which are the most expensive. This can be approved using the NVIDIA's bandwidth test:

- Copying data via PCIe from RAM to GPU's DRAM: 5500MB/s.
- Copying data via PCIe from GPU's DRAM to RAM: 4800MB/s.
- GPU to GPU copies: 150000MB/s.

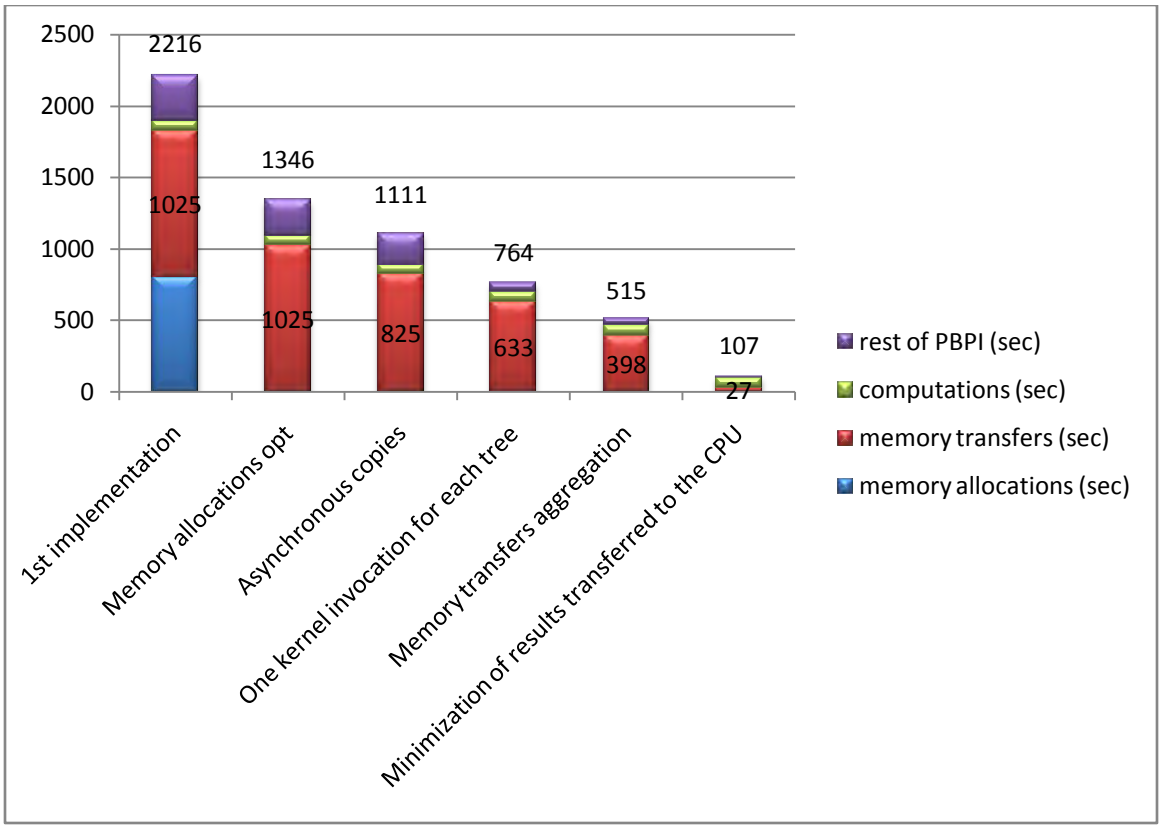


Figure 4.15 – Execution on GPU

This was the last optimization we did. In next chapter we will compare the performance of PBPI using the GPU against the CPU.

4.11 – GPU speedup of PBPI

4.11.1 – Methodology

For the time comparison between CPU and GPU we used the back218_L10000np256.nex input file with the defaults options:

- 4 Markov Chains
- 1,000,000 repeats for each chain
- 1,000 step for each repeat
- 5 run of the PBPI

We used the default setup because is a larger and more realistic experiment and this makes the benefits of using the GPU more obvious. We will execute PBPI on CPU with 1 thread, 2 threads and 4

threads using MPI and we will compare the execution time of these runs with the time execution on GPU. First we will compare the computations time only and then the total execution time of PBPI.

4.11.2 – Computations between CPU and GPU

The comparison between the core i7 860 and GeForce GTX480 for computation time only is shown in graph 4.16. The computation times of CPU and GPU are:

- 1 threaded CPU: 23h 59m 23s
- 2 threaded CPU: 12h 09m 57s
- 4 threaded CPU: 6h 44m 50s
- GPU: 1h 59m 52s

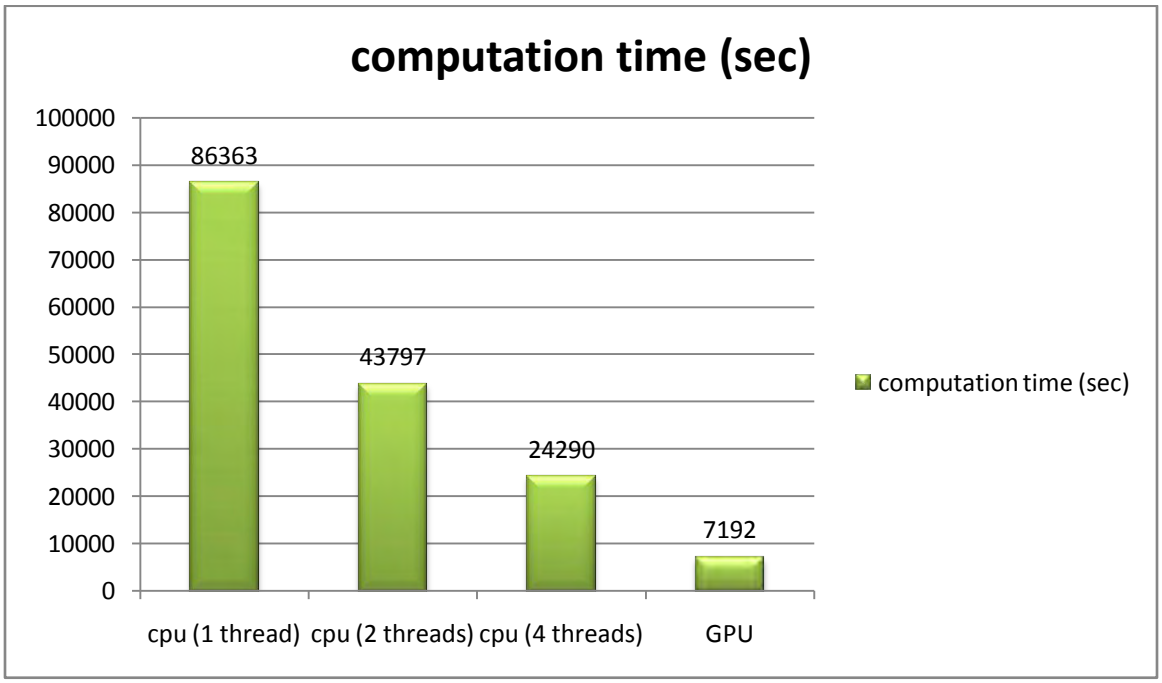


Figure 4.16 – Computation time

We see that GPU is faster than any CPU setup. It is significantly faster if we execute the computations on 1 thread on CPU but GPU is even faster if we execute the computations on 4 threads on CPU. The computations on GPU are 12 times faster than a single threaded CPU and 3.4 times faster than a 4 threaded CPU. The speedup of GPU against CPU is:

- 1 threaded CPU: 12x
- 2 threaded CPU: 6.1x
- 4 threaded CPU: 3.4x

Having a good speedup against CPU we will examine if the GPU is also faster against CPU for the total execution of PBPI. There is no importance if the computations are faster on GPU but the total execution is slower.

4.11.3 – Total execution of PBPI between CPU and GPU

We will compare the total execution time of PBPI between CPU and GPU as shown in graph 4.17. We used the same CPU and same GPU. These times are:

- 1 threaded CPU: 24h 51m 21s
- 2 threaded CPU: 13h 37m 46s
- 4 threaded CPU: 8h 06m 04s
- GPU: 2h 55m 08s

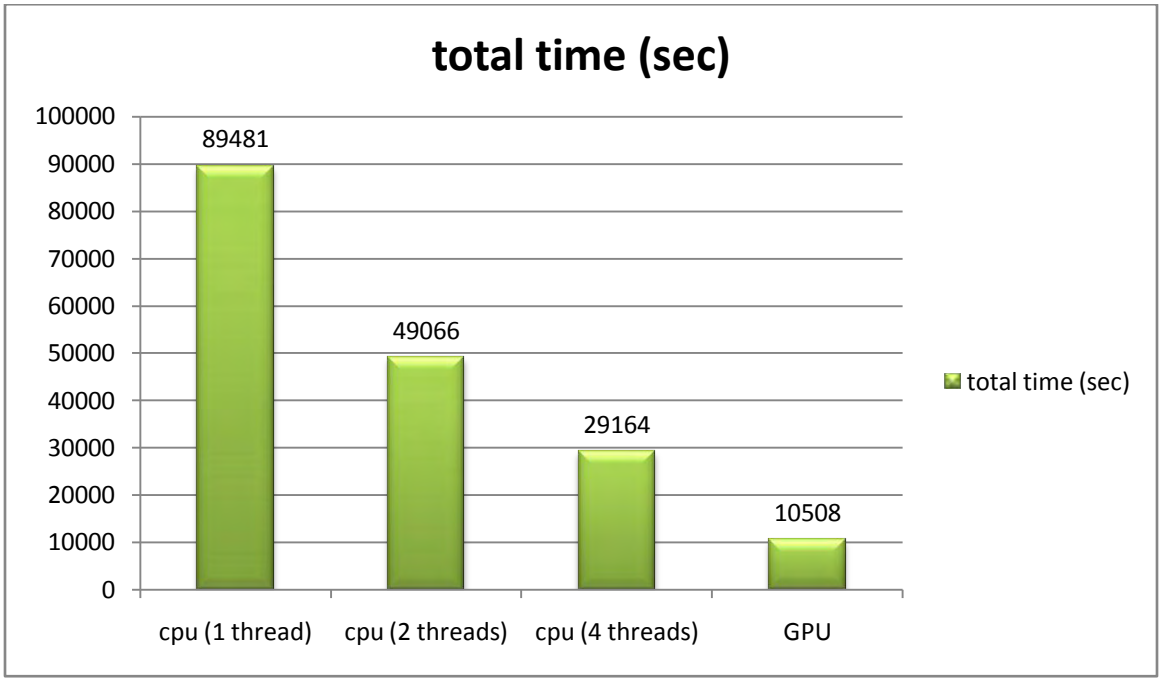


Figure 4.17 – total time of PBPI

The implementation on GPU is faster than the implementation on CPU and specifically GPU is 8.5 times faster a single threaded CPU and 2.7 times faster than a 4 threaded CPU. The speedup of GPU against CPU is:

- 1 threaded CPU: 8.5x
- 2 threaded CPU: 4.7x
- 4 threaded CPU: 2.7x

We see that GPU is faster than CPU again. The speedup may be less than the previous one but GPU is still much faster. If we want to reach GPU's performance with a CPU we must add more processors. But we cannot add more processors on the same system. So we have to create a cluster increasing system's cost and complexity.

We also have to say that the reason we did not show the time of 8 threads on CPU is that we did not have any performance improvement. The intel i7 860 may support 8 threads, 2 in each core, in parallel due to HyperThreading but in PBPI this is of no use. These 2 threads compute the same thing and they do not finally run in parallel. In order to test PBPI with 8 threads we should use an 8-core CPU.

Chapter 5 – Conclusion

PBPI is one of the most intensive computational applications. Although it has an algorithm improvement, is still too slow when running on a single thread on a CPU. Porting PBPI on a GPU gave us a very good performance. The implementation on GPU is even faster when we run PBPI on 4 threads on CPU. We finally managed to run PBPI fast enough on a GPU which was our initial target on a system of low cost and complexity exploiting the throughput of the GPU for parallel calculations.

GPU helped us to achieve a very good performance on PBPI but this performance needs a lot of optimization. GPU may have a better efficiency than a CPU but also a GPU needs a lot of work in order to achieve a great performance. Optimizing memory transfers is the most crucial thing when working on GPU. It usually needs to make your own memory manager in order to transfer all your data in one copy which is the ideal.

Also PBPI gave us the opportunity to see that we cannot use recursion on GPU. But this is not a big problem because the CPU can help to access structs recursively and then we can make our own struct on GPU with these data and accessing it repeatedly.

The next step is to execute PBPI in 2 GPUs but not in the same system because having two GPUs on the same system these two GPUs share the bandwidth of PCIe. We want to run PBPI in two different systems with a GPU in each of them. The application will be executed in two threads on CPU (one thread on each CPU of each system) each of them will be responsible for one GPU.

Annex

This annex contains results of PBPI executed on GPU with overclocked settings. Overclocking our system may give a little more power on our applications. Overclocking is considered as a dangerous act because it may cause system failure. But nowadays manufacturers are programming the hardware in that way that it can protect itself even if the user wants to push his system to the limits.

We will represent the way we tested our system. For these tests we used:

- An intel core i7 860.
- The MSI TWIN FROZR II GeForce GTX560Ti, a graphics card that is pre-overclocked from the factory.
- Asus Maximus III Extreme P55 chipset.
- Supertalent 4GB RAM, DDR3 2133MHz 8-8-8-24. The system was with the default settings, so the RAM was running at 1600MHz 9-9-9-24.

We used the back218_L10000np256.nex input file to compare the default and the overclocked settings for the execution of PBPI with the following options:

- 4 Markov Chains
- 50,000 repeats for each chain
- 1,000 step for each repeat
- 1 run of the PBPI

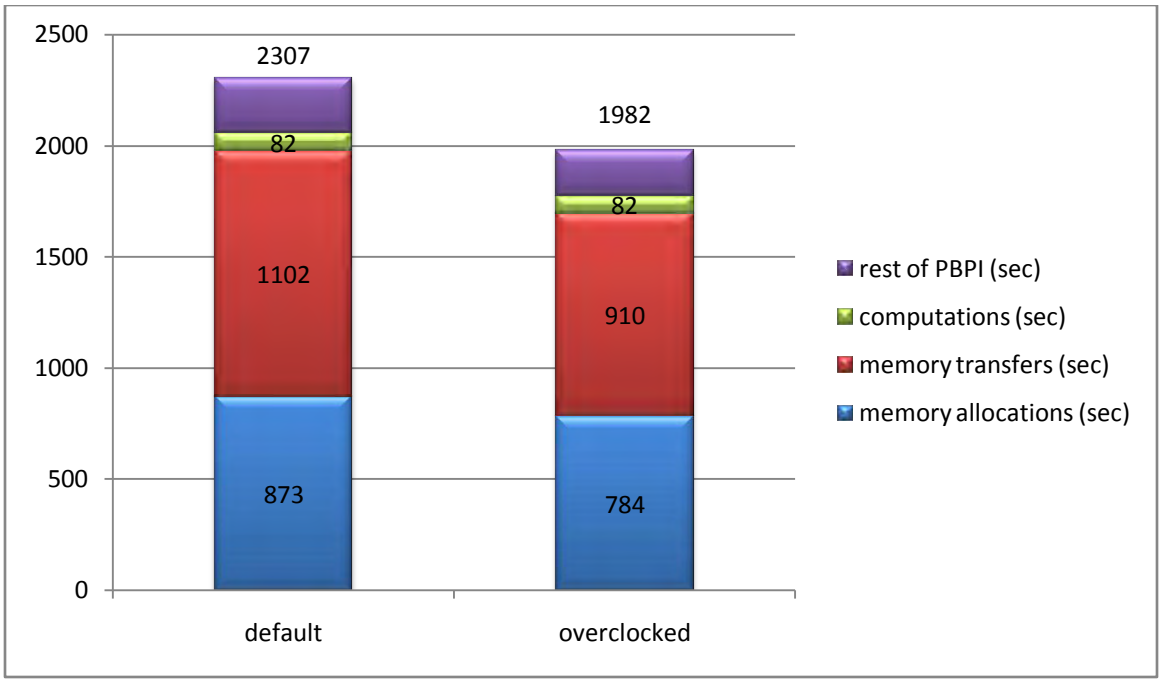
The most time consuming time functions for CUDA are the memory copies from and to GPU. These transfers occur over PCIe which is not fast enough. The bandwidth that offers the PCIe 2.0 x16 is:

- Copying data via PCIe from RAM to GPU‘s DRAM: 5500MB/s.
- Copying data via PCIe from GPU’s DRAM to RAM: 4800MB/s.

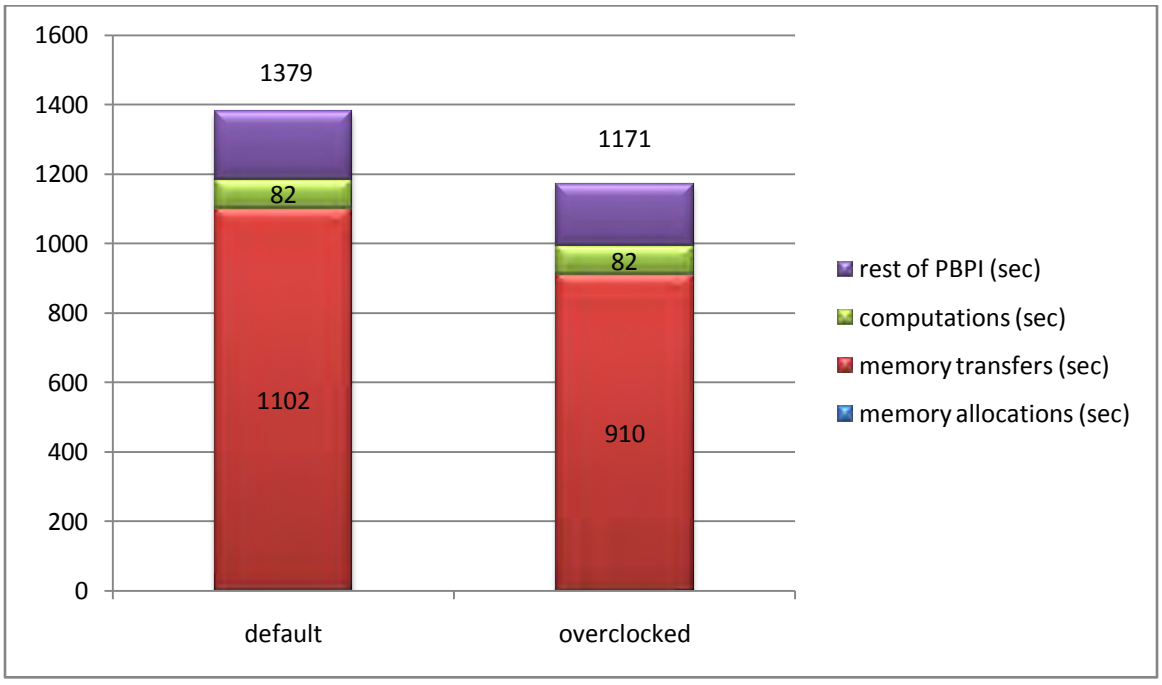
First we will test PCIe to see if there is any gain from it. The default setting is 100MHz. We pushed it to 140MHz which is a very high number for the 1156 socket and P55 chipset but not unstable for our motherboard. The new bandwidth is:

- Copying data via PCIe from RAM to GPU‘s DRAM: 6500MB/s.
- Copying data via PCIe from GPU’s DRAM to RAM: 5300MB/s.

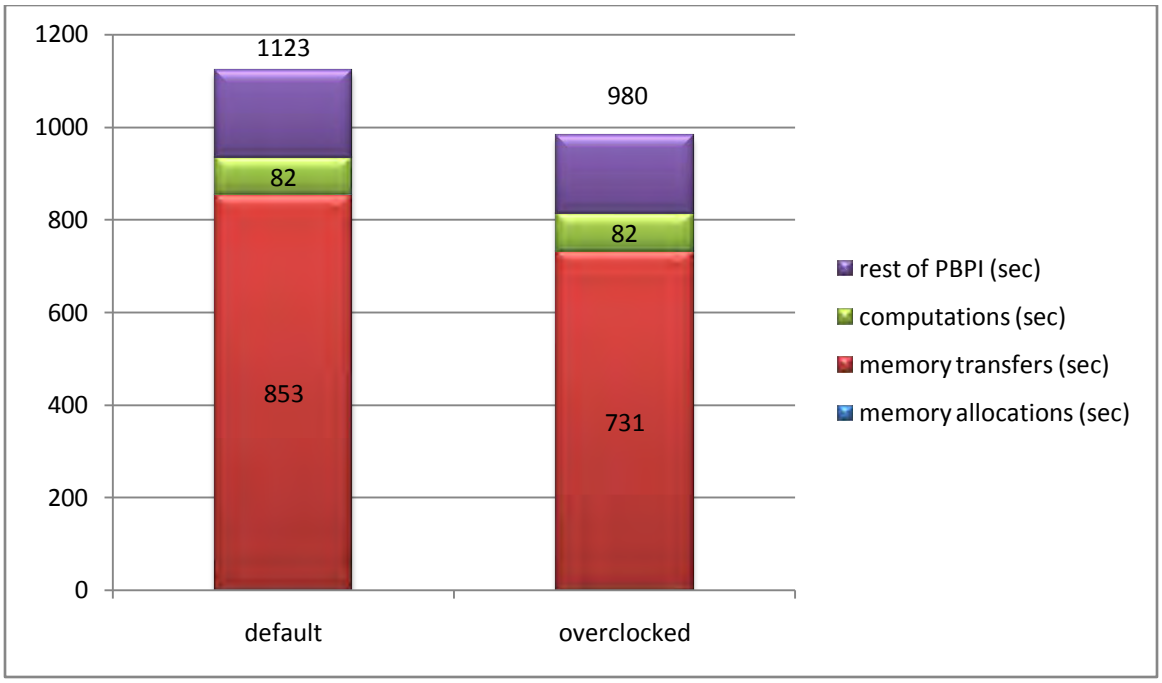
The next step is to test all our versions of PBPI on GPU to see the gain of each version. We can see that gain in the next 6 graphs.



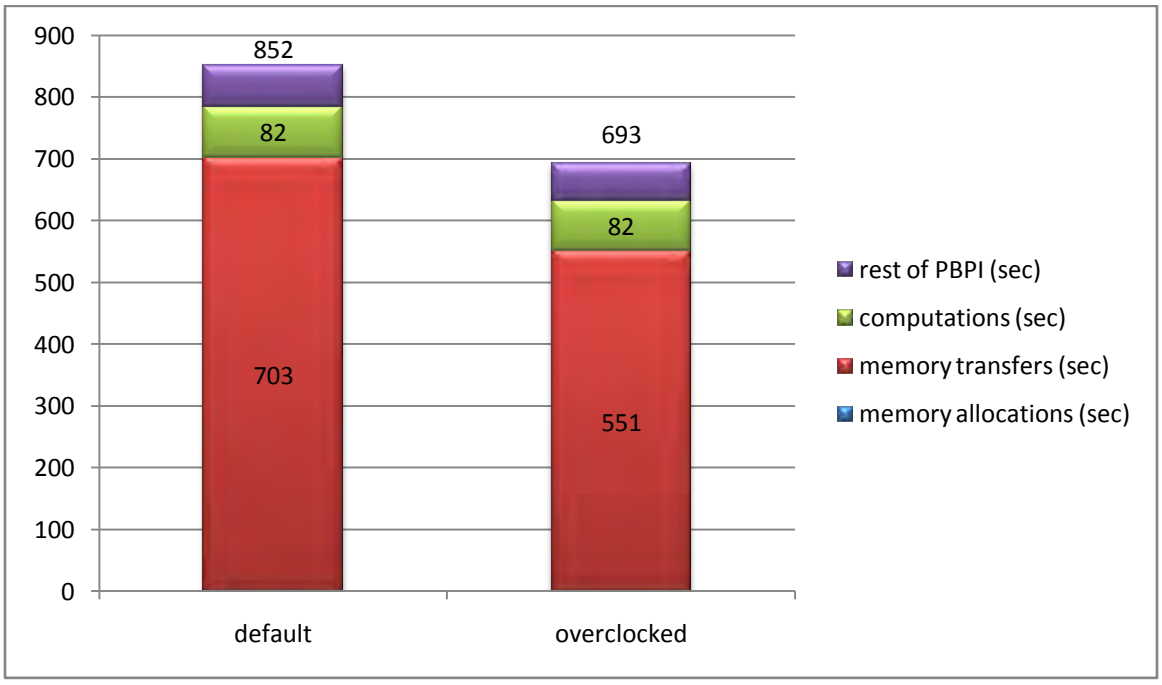
Initial implementation on GPU



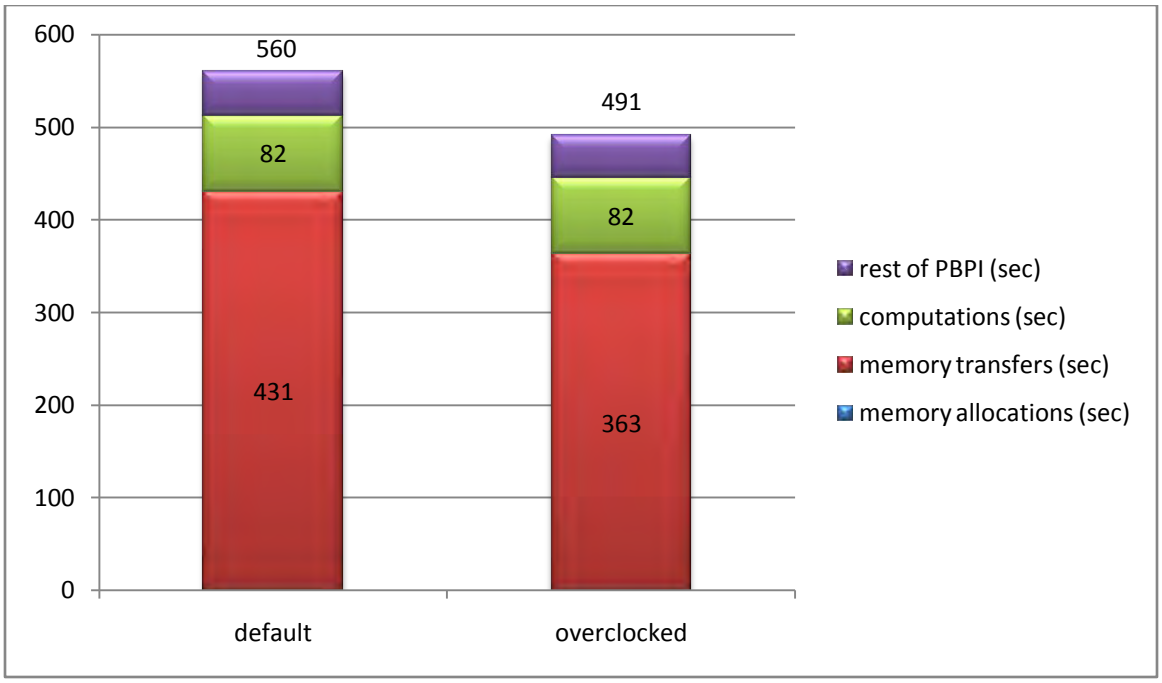
cudaMalloc optimization



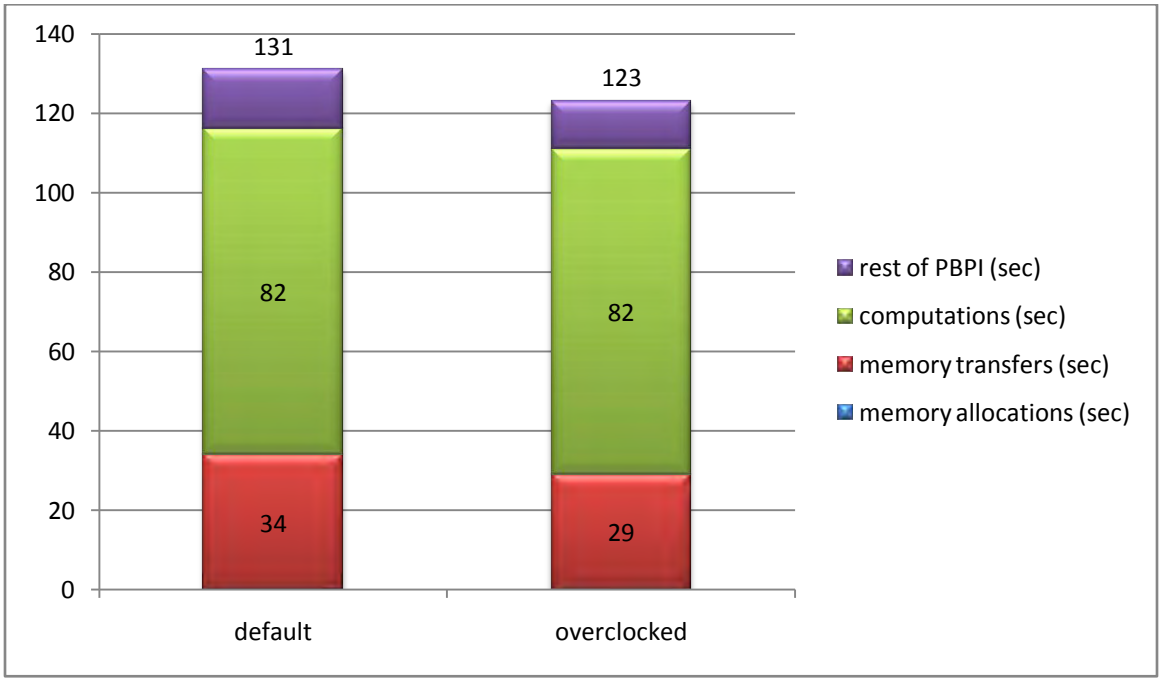
Asynchronous memory transfers



One kernel invocation for each tree



Memory transfers aggregation



Minimization of results transferred to the CPU

We infer that overclocking can help improve our performance in most cases. From the graphs we see that when we make many transfers it helps a lot. For example overclocking PCIe gave us a good performance improvement for implementations discussed in 4.4, 4.6, 4.7 and 4.8 chapters. These 4

implementations have many memory transfers and adjusting the MHz of PCIe gave us a performance boost. In contrast with these 4 implementations, the implementations discussed in 4.9 and 4.10 chapters, do not depend on PCIe overclocking especially the last one. These results show how important is to transfer our data in one copy. Not having a big gain from PCIe overclocking means that the performance of these implementations does not depend on memory transfers too much.

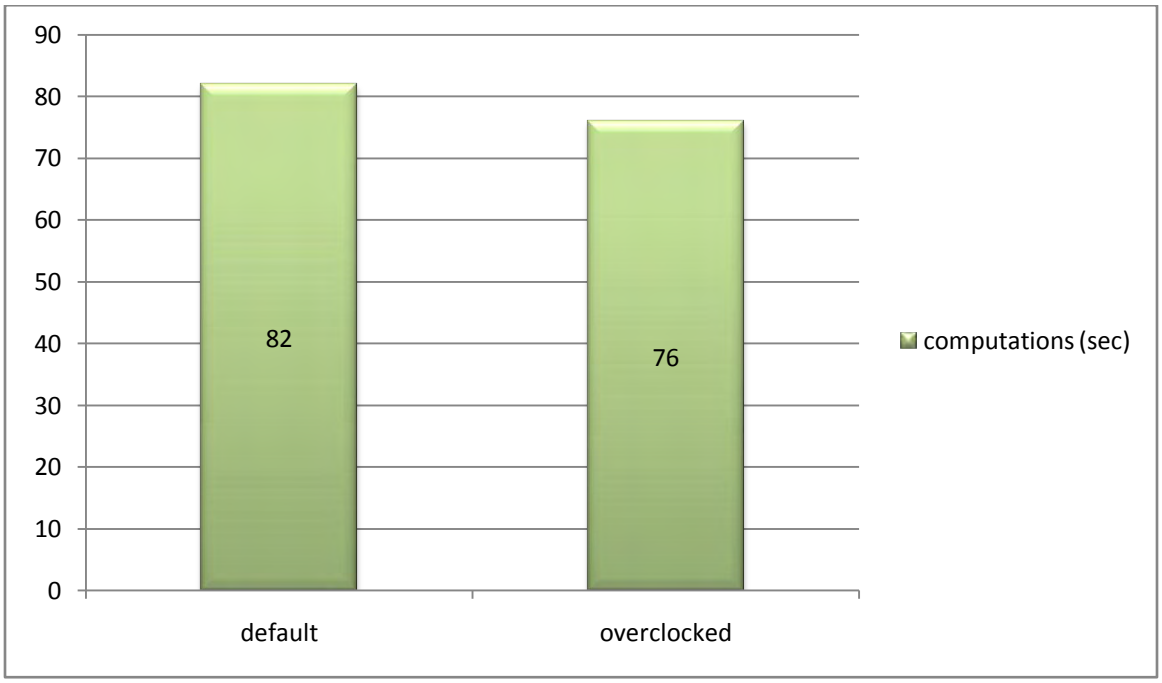
The second test we did is to overclock the GPU. The defaults clocks of MSI TWIN Frozr II GeForce GTX560Ti are:

- Core clock: 880MHz
- Shader clock: 1760MHz
- Memory Clock: 4200MHz

We tweaked the GPU's BIOS, changed the clocks and we saved the new BIOS with these clocks:

- Core clock: 950MHz
- Shader clock: 1900MHz
- Memory Clock: 4580MHz

The results for the computations only are shown in the graph. There is no need to measure the entire time of PBPI because the GPU executes only the computations.



Overclocking our GPU may have decreased the computation time but there in not a big difference. This is because the GPU executes the computations in parallel. If we had a single thread running we would see a big difference but now each thread executes a few instructions in parallel with

others and a small increment in MHz does not really affect the performance too much. In contrary if we could run more threads in parallel then we would see a big difference.

References

[1] "Maximum parsimony (phylogenetics)", http://en.wikipedia.org/wiki/Maximum_parsimony_%28phylogenetics%29

[2] "Maximum likelihood", http://www.icp.ucl.ac.be/~opperd/private/max_likeli.html

[3] Huelsenbeck, J. P., Larget, B., Miller, R. E., et al., "Potential applications and pitfalls of Bayesian inference of phylogeny", 2002

[4] "Phylogenetic tree of life", http://en.wikipedia.org/wiki/File:Phylogenetic_tree.svg

[5] Xizhou Feng, Kirk W. Cameron, Duncan A. Buell, "PBPI: a High Performance Implementation of Bayesian Phylogenetic Inference", 2006

[6] Bradley P. Carlin, Andrew Gelman , Radford M. Neal, "Markov Chain Monte Carlo in Practice: A Roundtable Discussion", http://www.stat.columbia.edu/~gelman/research/published/kass5.pdf, 1997

[7] Felsenstein, J., "Evolutionary trees from DNA sequences: a maximum likelihood approach", 1981

[8] "CUDA", http://www.nvidia.com/object/cuda_home_new.html

[9] "CUDA", http://en.wikipedia.org/wiki/File:CUDA_processing_flow_%28En%29.PNG

[10] Tom R. Halfhill, "Looking Beyond Graphics", 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_Looking_Beyond_Graphics.pdf

[11] "Intel Core i7-800 Processor Series and the Intel Core i5-700 Processor Series Based on Intel Microarchitecture", http://download.intel.com/products/processor/corei7/319724.pdf

[12] "Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem)Based Processors", http://download.intel.com/design/processor/applnots/320354.pdf, 2008

[13] "Using Intel® VTune™ Performance Analyzer to Optimize Software on Intel® Core™ i7 Processors", 2009