Internal Report 2011–06 September 2011

Universiteit Leiden
Opleiding Informatica

Exploratory research on embedding CUDA code
into hetrogeneous MP-SOC achitectures
programmed with the Daedalus framework

Rick van der Zwet

BACHELOR'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Exploratory research on embedding CUDA code into heterogeneous MP-SoC architectures programmed with the Daedalus framework

Rick van der Zwet
<hvdzwet@liacs.nl>
Leiden Institute of Advanced Computer Science, The Netherlands
License: Creative Commons Attribution

September 15, 2011

Abstract

The objective of this Bachelor Thesis is to explore the possibilities of using *NVIDIA CUDA* enabled *GPU* Processors within the `HDPC` framework. The `HDPC` framework is one of the heterogeneous *MP-SoC* architectures programmed with the `Daedalus` framework.

This paper will focus on the transfer overhead introduced by using the *GPU* and how to best cope with this introduced latency.

In this study it was found that using the *GPU* instead of the *CPU*, the overall execution times will decrease if the execution pattern has specific characteristics with regards to token size and processing complexity.

1 Introduction

Traditionally computation is the domain of the general purpose *CPUs*, but new computation stars are arising on the horizon. Starting from 2007 *NVIDIA* has released the *NVIDIA* Compute Unified Device Architecture (*CUDA*)

framework. This framework allows programming on a *CUDA* enabled *NVIDIA* Graphics Processing Unit (*GPU*).

The *GPU* uses specialized co-processors to offer specialized computation methods for highly parallel computations claiming faster computation times with regards to the *CPU*. The faster computation time is achieved by partitioning a (multi-threaded) program into blocks of threads that execute independently from each other.

The *GPU* is not meant as replacement for the *CPU*, but to be used in Stream Processing. This means that the *GPU* needs a supporting *CPU* for control and setup. The supporting *CPU* offloads specific computing tasks to the *GPU* and processes the results from the *GPU*. This specific computing tasks on the *GPU* are based on Single Instruction, Multiple Data (*SIMD*). *SIMD* is used to exploit data level parallelism [Flynn1972].

The co-processors on the *GPU* do not share memory with the *CPU*. The *CPU* has its DDR memory located on the motherboard [1] (*Host Memory*) and the *GPU* has its own dedicated DDR memory which is located on the *GPU* itself (*Device Memory*).

As the *GPU* and the *CPU* do not share memory, memory transfers are needed to transfer the data from the *Host Memory* to the *Device Memory* and back. These transfers need to be processed by the communication channel between the *GPU* and the mainboard.

There are multiple technologies in use to connect the *GPU* to the mainboard. The *GPU* is normally connected to the motherboard using a The Peripheral Component Interconnect Express (*PCIe*) bus or Accelerated Graphics Port bus (*AGP*). The *AGP* *GPU* cards are not worth considering as their (Device to Host) bandwidth is roughly 10 times lower than the *PCIe*. [2]

To make sure the *GPU* processing is faster than the *CPU* equivalent, the time needed by the *PCIe* communication channel to transfer the data should not exceed the speedup gained by the highly parallel execution on the *GPU*.

The *GPU* has a fixed amount of *Device Memory* available. A computation which needs more memory requires the data to be split. The resulting data chunks need to be well-balanced with regards to transfer time and com-

[1] The motherboard is also called mobo, mainboard, system board or logic board.

[2] `http://en.wikipedia.org/wiki/List_of_device_bandwidths`

putation time. Making the data chunks too big will cause the *GPU* to spend most of the time waiting for the data to transfer. Making the data chunks too small will cause the *GPU* being busy doing many small transfers and not actually making efficient computations.

Now consider a program designed as a Kahn Process Network (*KPN*), a model of computation introduced by Dr. Gilles Kahn [Kahn1974]. This *KPN* has computation components (processes) which are connected via communication channels. In a *KPN* there is no notion of a global schedule that dictates the relative order of execution.

To make the program running as fast as possible it is advised to run a process on the most suitable hardware platform available. This takes both communication speed and the type of the communication channel into account.

Now consider a *KPN* which has two processes which are connected via one communication channel. Both processes are scheduled to run on the *CPU* sharing the *Host Memory*. Passing a *Host Memory* pointer between two *CPU* processes reduces the communication channel overhead to almost 0. However, if one process is scheduled on the *GPU* and the other is scheduled on the *CPU* the communication channel time overhead is non-trivial.

The right chunk size of data which is transferred at ones and the number of such data chunks, transferred during the execution of the program have (great) impact on the performance of the *GPU*. Finding the right balance between these factors is critical to consider the *GPU* as computation platform next to the alternatives available.

To allow every computation component in the *KPN* to be executed on a specific platform, glue-code is needed to connect them together. For a general purpose computer architecture such as Intel and AMD x86(-64) this glue-code is provided by the Heterogeneous Desktop Parallel Computing (`HDPC`) framework developed at `LIACS` [Far08].
`HDPC` currently supports the hardware platforms *CPU*, *GPU*, Field Programmable Gate Array (*FPGA*) and Cell Broadband Engine Architecture (*Cell BE*) to be used for process execution. The `HDPC` framework is depicted in Figure 1 and discussed in more detail in Section 2.1.

The `HDPC` framework uses as input a generated *KPN* by the `Daedalus` framework developed at `LIACS` [DAC2008]. The `Daedalus` framework is able to convert a sequential C/C++ program to parallel *KPN*. The `Daedalus`

framework in this paper will be used to generate our experiment *KPN*. The HDPC framework has to choose on which platform to place a process for execution. Due to the balance problem mentioned above, the choice to use the *GPU* or *CPU* for execution of a process is challenging.

This Bachelor Thesis will explore the proposed techniques of embedding *CUDA* code into the Daedalus framework by [Far08]. By exploration this Bachelor Thesis will find out if the *CUDA* enabled *NVIDIA GPU* can be used in the Daedalus framework to speedup the overall computation time.

The results of this exploratory research can also be used by programmers who want to take maximum advantage of the capacities of the *GPU* architecture.

The document is organized as follows; Section 2 will introduce the *CUDA* and HDPC terminology. The problem itself will be explained in Section 3, to continue with computation of theoretical values in Section 4. Testing will be done with the test setup depicted in Section 5 and implemented using Section 6. All experimental results are found in Section 7, explaining the differences between the theoretical values and the experiment results. Experiment conclusions are found in Section 8. And finally recommendations will be given for further research in Section 9.

2 Background

2.1 HDPC

Specifying an application using a parallel Model of Computation (*MoC*) such as the *KPN MoC*, is a time-consuming and error prone task which is currently not well understood by developers. Therefore, we need tools that allow us to continue writing sequential programs and automatically derive parallel specifications.

The Daedalus framework tools developed at LIACS are designed for this purpose. The pn tool generates a *KPN* from a sequential C/C++ program. The Embedded System-level Platform Synthesis and Application Mapping (Espam) tool will use the generated *KPN* together with the platform and mapping specification as input. The Espam tool generates several autonomous

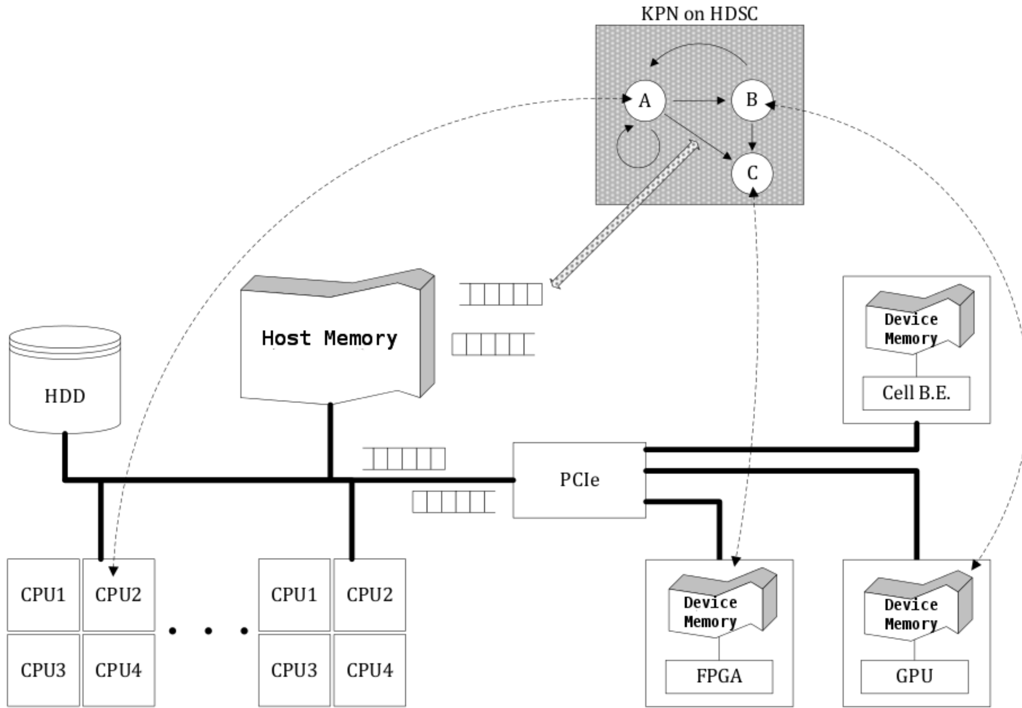


Figure 1: An `HDPC` framework example. A *KPN* running on our framework with three interconnected processes, A, B, and C all execute their functions on a device connected to the same machine. Communication channels and the *FIFO* mechanism are implemented in main system memory (*Host Memory*) and are under control of `HDPC` code. Transfer of data between different devices happens by reading from the *Device Memory* associated with the processing node into *Host Memory*, then writing this data in due time to the *Device Memory* of the consuming node. In the case of execution on the *CPU* this transfer is either simply a memory copy or a pointer change as all data is in the same address space. Picture taken from [Far08, pg.4]

processes that transfers data between each other through *FIFO* communication channels. The `Daedalus` framework is depicted in Figure 2.

`HDPC` uses above mentioned tools and generates code for a general purpose computer such as the Intel or AMD x86(-64). The processes of a *KPN* can then execute on a heterogeneous multitude of platforms.

HDPC extends upon Espam by generating back-end code for a desktop computer that acts as the controlling and coordinating arbiter between the processes of a *KPN*. The processes can then execute not only on the *CPU* cores but also on various computing devices like the FPGA, Graphics Pro-

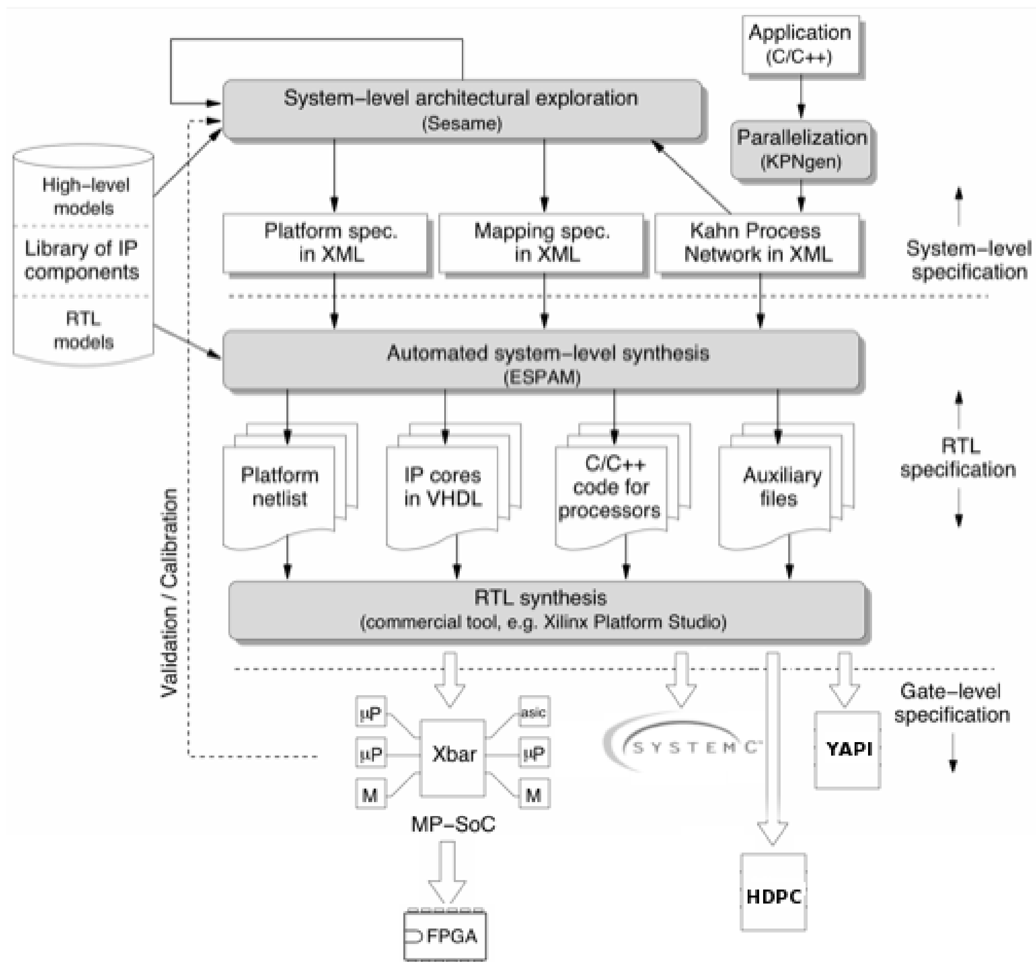


Figure 2: Daedalus grand overview

cessing Unit (*GPU*), or the Cell B.E. to take advantage of their respective strengths.

For each process in a *KPN*, a thread on the host *CPU* is created. If computations inside the node are executed on the host machine, the *CPU* (of a multi-core system) is used for the actual computation. For external devices like the *GPU*, the host thread is only responsible for control flow, transfer of data to- and from the device and starting execution of the computation on the device; nothing more.

2.2 CUDA

NVIDIA offers the *CUDA* Software SDK allowing programming on the *GPU* free of charge in binary format, the source code is not provided. The binary format introduces some difficulties with regards to optimizations and limit calculations as the *CUDA* Software SDK is essentially a black box, where internal implementation on new releases might change without notice [Halfhill2008].

CUDA is a general purpose parallel computing architecture that leverages the Parallel Compute Engine in *NVIDIA GPUs* using the *CUDA* Instruction Set Architecture (*ISA*) and the Parallel Compute Engine in the *GPU*.

The *ISA* and the Parallel Compute Engine vary between releases of *GPUs*. To provide a unified programming interface with backward compatibility the *CUDA GPU* driver and the *CUDA* SDK provides a programming interface for the developer. This allows the developers to perform parallel programming on *NVIDIA CUDA* enabled *GPUs*, without the need of releasing a new version for every new release of the *CUDA* driver or software.

The more cores the *GPU* has the less time it needs to execute the program as threads can be scheduled on more cores. This advantage is shown in Figure 3.

To program the *CUDA* architecture you will have to use C with *CUDA* specific extensions, which will be processed by *NVIDIA* provided PathScale Open64 C compiler and you will also need to have a *NVIDIA* compatible GPU installed with the correct driver version. For debugging purposes the software can be compiled to run in a simulation mode, which is executed on the *CPU* only. The simulation mode executes very slow on the *CPU*, but makes it possible to follow the execution path and spot errors on memory reference for example.

There are 2 levels on which you can program *CUDA* code, the so called high-level API and the low-level API. As those names suggest, the low-level is used for more control over the code, but it also requires a fair amount of bookkeeping to make sure the processes keep data synchronized and consistent.

In order for the *CUDA GPU* to allow calculation it (1) needs to transfer the data from the *Host Memory* to its own *Device Memory* (2) Initiate the

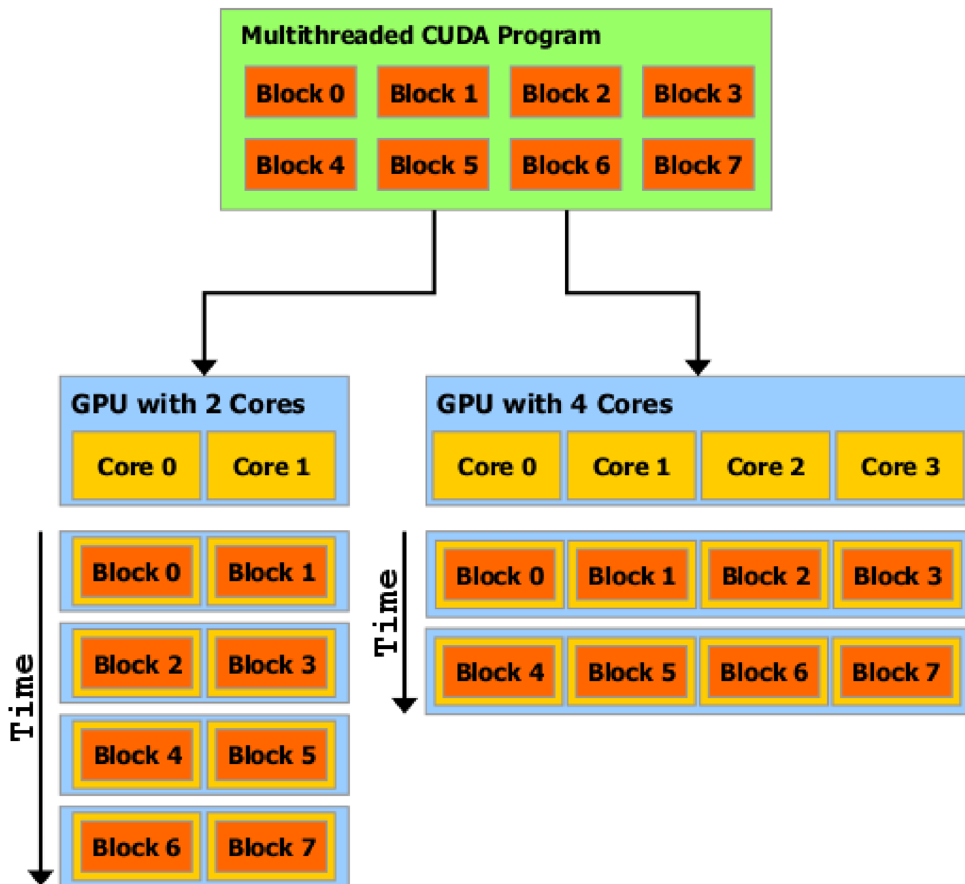


Figure 3: A GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

computation (3) and do the actual computation. (4) On completion transfer the data back from the *Device Memory* to the *Host Memory* . This flow is depicted in Figure 4.[3]

For more background information refer to the *CUDA* Getting Started Guide [CUDA-GS] and *CUDA* C Programming Guide [CUDA-PG].

[3]The original figure is CC-BY Tosaka and found at `http://en.wikipedia.org/wiki/File:CUDA_processing_flow_%28En%29.PNG`

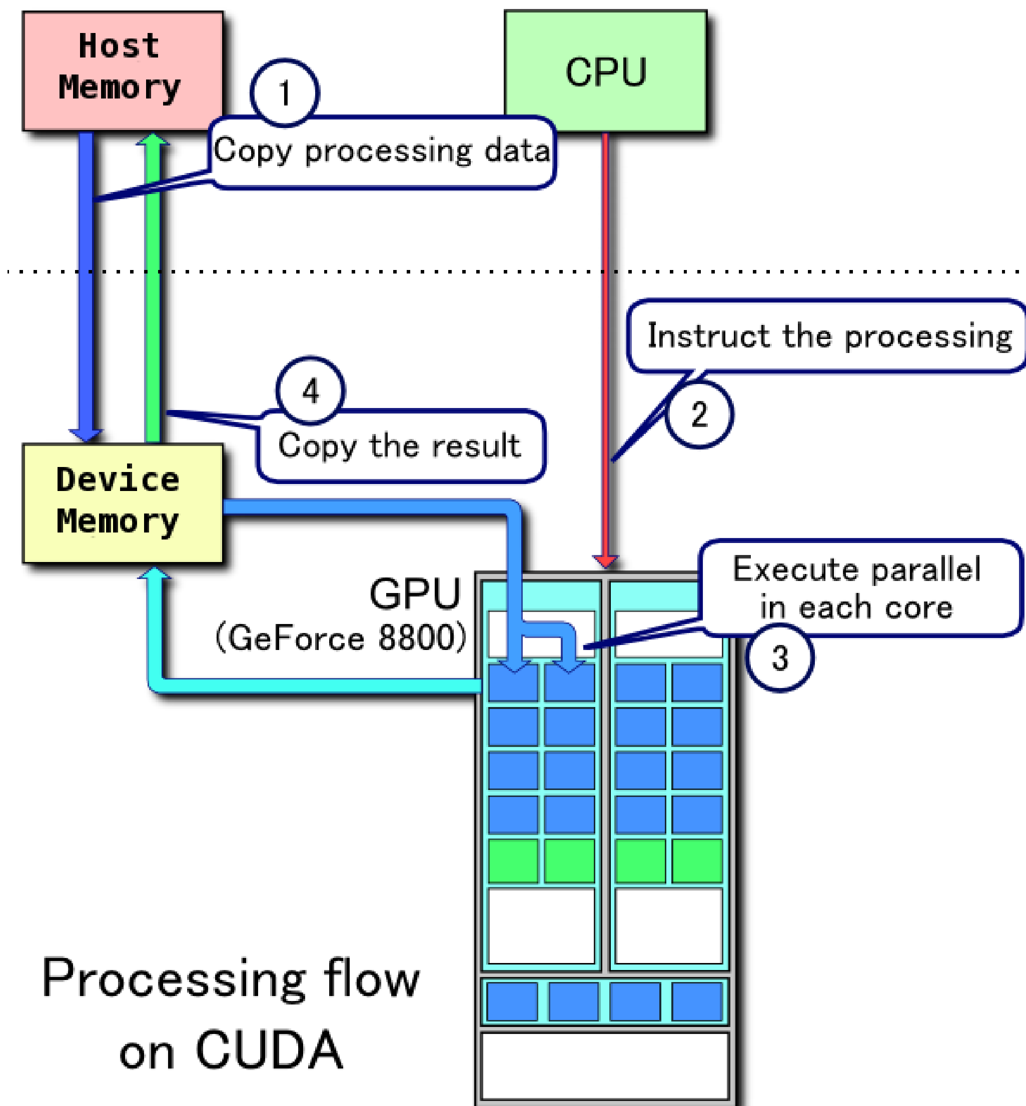


Figure 4: Processing flow on *CUDA GPU*. The part above the dashed line is physically located on the mainboard and the part below is located on the *GPU*.

3 Problem description

We want to execute a simple program `Producer` ⇒ `Calculator` ⇒ `Consumer`, modeled as *KPN*, with processes and communication channels as depicted

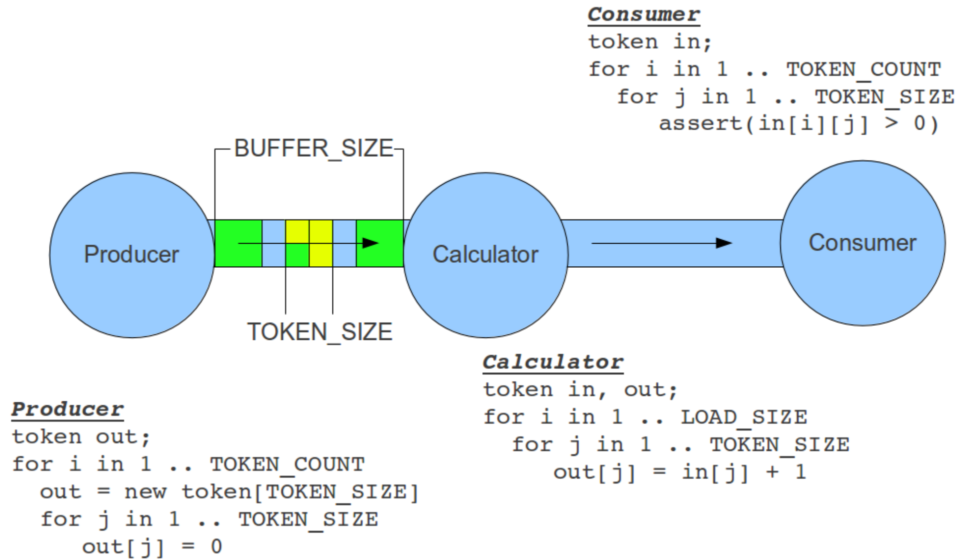


Figure 5: The design of `Producer` ⇒ `Calculator` ⇒ `Consumer` using 3 processes (`Consumer`, `Calculator` and `Consumer`) and 2 *FIFO* communication channels, including pseudo-code implementation of the processes.

in Figure 5. In this program there is single process `Producer` and a single process `Consumer` which will be executed on the *CPU*. The single process `Calculator` in the `Producer` ⇒ `Calculator` ⇒ `Consumer` program however will be executed either on the *GPU* or the *CPU*.

Within the program `Producer` ⇒ `Calculator` ⇒ `Consumer` the `Producer` generates a fixed amount of entries (`ENTRY_COUNT`). These entries are packed in tokens with each token containing a specific amount of entries (`TOKEN_SIZE`). As soon as a token is ready it will be written to the first communication channel. The `Producer` will repeat this process till a fixed amount of entries is reached (`TOKEN_COUNT`).

The first communication channel transfers the tokens from the `Producer` to the `Calculator` acting as First-In-First-Out (*FIFO*) buffer with limited token capacity (`BUFFER_SIZE`).

The `Calculator` on its part performs a *blocking read* on the first communication channel. A *blocking read* will read a token from the communication channel if there are tokens waiting, else it will wait for one token to arrive. Next, the `Calculator` will perform a computation on every individual entry within a token. This computation has a constant workload set for all entries. This workload is specified in number of loop iterations by parameter `LOAD_SIZE`. When the computation is done the result is written in the second communication channel. The `Calculator` will repeat the process, so the `Calculator` will proceed processing a new entry if one is available in the first communication channel, else it will perform the *blocking read* on the first communication channel.

This computations of the `Calculator` (on all entries within a token) do not share any dependency and can thus be executed in any order. This property is needed as the operations within the `Calculator` need to be (highly) parallel in order to be a suitable candidate for *GPU* computing.

The second communication channel has the same characteristics as the first communication channel, but different end-point. The second *FIFO* buffer will transfer tokens from the `Calculator` to the `Consumer`.

The `Consumer` will do a *blocking read* on the second communication channel and will make sure the result calculated by the `Calculator` is actually valid.

Please observe that there *is* a strict execution order between tokens. The `Calculator` can only process one token at a time and there is only one `Calculator` process connected to the first and second *FIFO* communication channels. The *FIFO* buffer of the communication channels enforces a strict order in the tokens by design.

For the `Calculator` there are two platforms considered to run the code on. First the *GPU* will be discussed and secondly the *CPU*. Both the *GPU* and the *CPU* have two communication types for the attached *FIFO* channels. Data can be transferred by either physically movement of data (*COPY*) or by pointer reference type (*POINTER*) as defined by the `HDPC` framework [Far08, pg. 6].

As the *CUDA GPU* is not able to access the *Host Memory* directly, the *GPU* needs to be prepared by the supporting *CPU* before execution. The two strategies of preparing memory for the *CUDA GPU* are implemented as follows:

1) *COPY*: Assign memory on the GPU device and copy the content from *Host Memory* to *Device Memory*. When the computation is done copy the result stored in *Device Memory* back to *Host Memory*.

2) *POINTER*: Assign special *CUDA* allocated memory in *Host Memory* which can be used for both the *CPU* and the *GPU*. Next give the pointers to the newly allocated source and target address spaces to the *CUDA GPU*, the back-end driver will dynamically transfer the data to the *GPU* when needed.

"Preparing" the *CUDA* will cause an initial start-up penalty, the same applies for memory transfers between the *CUDA GPU* and the *Host Memory*. This "setup-overhead" can eventually become larger than the gained computation speedup, causing the overall computation to be slower.

The drawback of the *COPY* method comes with choosing the "wrong" transfer sizes, which causes unnecessary many *COPY* transfers being initiated. "Minimize data transfer between the host and the device" is advised by *CUDA* [CUDA-BPG][pg. 15] but no estimation is given on the best transfer sizes to use, in order to make best use of the *GPU*.

The drawback of the *POINTER* method comes with the fact that the transfer logic is handled in the back-end, which cannot be given any hints on what kind of data to expect with regards to size and type. This will cause the build-in optimizer to guess the best transfer sizes making it potentially (very) inefficient.

For our program `Producer` ⇒ `Calculator` ⇒ `Consumer` we now have four different execution strategies we can choose from when executing the program, namely the executing strategies *CPU COPY*, *CPU POINTER*, *GPU COPY* and *GPU POINTER*. But which execution strategy leads to the lowest execution time?

Finding the answer to this question requires finding a *Tipping Point* where the extra communication overhead time needed by different execution strategies will be less than the gain achieved by the computation speedup.

Finding this *Tipping Point* is not-trivial as it requires writing code for specific hardware architectures with the same software architecture in mind. The `HDPC` framework can help here structuring the code in such a way that both cases will be comparable.

In order to make the decision about the execution strategy, two questions emerge: 1) Would it matter if we know the amount of entries to process (`ENTRY_COUNT`) and the complexity of the computation (`LOAD_SIZE`) in advance? 2) Giving this knowledge can we apply basic design rules when developing *KPN* process nodes which need to run on the *GPU*?

4 Theoretical Calculation

The interface between the *GPU* card and the *CPU* is the PCIe x16 bus. The Bandwidth of the *PCIe* x16 bus operates at a theoretical maximum of $5GHz$. This gives a theoretical maximum bandwidth of *500MB/s * 16 = 8GB/s.*[4]. This theoretical maximum causes for example a transfer of 8GB to the *GPU Device Memory* and back into the *Host Memory* to add a 2 seconds overhead in transfer time.

The amount of Clock Cycles an arbitrary operation takes on the GPU Cores is found at [CUDA-PG][pg. 94, table 5-1]. The Clock Cycles per arbitrary operation is ranging from 8 to 48, depending on the Computation Capacity of the *GPU*.

To calculate the time needed for transfer and computation on the *GPU*, a variety of constants are needed in the calculation. These constants are found in the [CUDA-PG], [CUDA-BPG] and the data-sheet found on the manufacturer website. The Experiments in Section 7 uses the `NVIDIA GeForce GTX 295` *GPU* card, thus the theoretical computations below are based on this card.

The Theoretical Bandwidth on the `NVIDIA GeForce GTX 295` is calculated using[CUDA-BPG, pg. 13] and the values given by *NVIDIA*:[5]

$$999MHz * 10^6 * 488bit/8bit * 4/10^9 = 223.8GBps \quad (1)$$

In Formula (1) the Memory Clock is $999MHz$, the Memory Interface Width is 448 bits. The multiplier 4 comes from the quad data rate configuration, which is derived from the *SLI* feature. This concludes that the internal

[4] `http://en.wikipedia.org/wiki/PCI_Express`
[5] `http://www.nvidia.com/object/product_geforce_gtx_295_us.html`

bandwidth for the `NVIDIA GeForce GTX 295` is much higher ($\approx$ 25 times) than the *PCIe* Bandwidth.

Accessing the global memory costs between 400 to 600 Clock Cycles as found in [CUDA-BPG, pg. 47], the `NVIDIA GeForce GTX 295` runs at $1242MHz$, has 30 Multiprocessors (*MP*) and 8 $Cores/MP$ making a total of 240 cores. The Compute Capability is 1.3, so it will execute 8 additions/-comparisons per Clock Cycle per Multiprocessor[CUDA-PG, pg. 94].

For the calculations below, the memory access is assumed to be perfect. Perfect memory accesses on both the *GPU* and *CPU* are achieved if the data is aligned properly and accesses are sequential to avoid misses and extra transfers/calculations to make it fit in the architecture. For the *CUDA GPU* the differences can be as large as $\approx$ 8 times [CUDA-PG][pg. 164, figure G-1].

Consider processing 2 million `int32_t` integers = 64 million bits = 8 million bytes. Every entry will be processed by a single thread, this requires an total of 2 million threads. An single thread will perform 1 addition (ex: $output = input + 1$).

Assume the process is able to transfer the whole memory content (all 2 million integers) from *Host Memory* to *Device Memory* (and back) at ones. The process will then take the following approach: 1) Transfer all integers to the *Device Memory*. 2) compute concurrently for every single integer; a) load the value from memory b) Perform one addition and c) store the result. 3) When all computations are done transfer the result to *Host Memory*. Looking at the time needed for every individual step of the computation:

1) The transfer from *Host Memory* to *Device Memory* will take $0.5ns$ per entry:

$$4byte/8GBps = 0.5ns \tag{2a}$$

2) One Memory Access (read or write) operation takes as much as 600 clock cycles whereas one addition operation only takes 1 clock cycle. Due this big differences the addition clock cycles will not be considered in the calculation. This makes the total needed for Memory Access read and write using 30 Multiprocessors to be:

$$2 * 2 * 10^6 * 600/30 = 1 * 10^7 \ Clock\ Cycles \tag{2b}$$

the cores are running at $1242MHz$ so this will take in total:

$$\frac{1 * 10^7}{1242 * 10^6} = 8 * 10^6 ns = 8ms \tag{2c}$$

The total internal transfer time to the *GPU* is very small:

$$\frac{0.008GB}{223.8GB} = 0.03ms \tag{2d}$$

For that matter the transfer time to the *GPU* can be ignored with respect to the calculation times. Which makes an average time to perform an addition on 1 integer to be:

$$8ms/2 * 10^6 sec = 4ns \tag{2e}$$

3) The transfer from *Device Memory* to *Host Memory* will takes per integer $0.5ns$:

$$4byte/8GBps = 0.5ns \tag{2f}$$

The sum of all answers in Formula (2) will be $0.5ns + 4ns + 0.5ns = 5ns$ per integer.

If the *GPU* time is reduced to $0ns$, the average time will still be $1ns$ for transfer only. However, the average time does not equal the latency time. The latency is significant as it cannot start computing before all integers have been transferred. For this example it causes a $1ms$(!) delay before the computation on the *GPU* can start.

The actual average time will increase as the theoretical average time does not yet include the following details; 1) The "setup costs" on the supporting *CPU*. One example of "setup costs" is the *GPU* driver initialization. 2) The time needed on *GPU* for context switches and thread initialization. 3) Overhead caused by (multiple) Memory Transfers if data is not transferred at ones. 4) Overhead caused by hardware not fully utilized e.g. waiting for data or instructions. 5) Overhead caused by the `HDPC` framework used for programming.

NOTE: In the theoretical example of processing 2 million integers the *GPU* computation power is not fully utilized. For a best case scenario all memory fetches are pipe-lined on the *GPU* and during the "waiting" time the *GPU* executes an addition. Best case scenario the computation load could be increased to 600 clock cycles. This way a Multiprocessor can fire off half of its cores to fetch Memory and the other half to run the computation and "flip" between this configurations.

5 Construction of Test Environment

NVIDIA CUDA enabled (consumer) *GPU* cards are available in numerous architectures and variety of cards with regards to main factors such as number of MultiProcessors, Compute Capability and Clock Speed. Building a uniform test environment is preferred to rule out operating system and software/toolkit differences.

In this experiment the `Producer` and `Consumer` run on the *CPU* and the `Calculator` can either be a *CPU* process or a *GPU* process. This approach will eliminate any difference in the setup. The `Calculator` will simulate its load by using a simple addition on the entry. The `Consumer` will validate the output of the `Consumer` to ensure the `Consumer` is not cheating or generating errors in memory transfers.

The experiment Scheduling and Memory transfer code is automatically generated by the `Daedalus` framework with target `HDPC` framework, which for this experiment is `Producer` ⇒ `Calculator` ⇒ `Consumer` as shown in Figure 5.

The functions that the three processes (`Producer`, `Calculator`, `Consumer`) implement are manually coded, with maximum execution speed in mind.

During the compilation some compiler optimizations are disabled, to ensure that the compiler does not 'trick' the results by applying secret (unwanted) optimizations. Disabling the specific (loop) optimizations, makes sure the profiling code will run as intended.

The total execution time will be calculated and divided by the number of entries (`ENTRY_COUNT`) to get the execution time for a single entry. This will include the `Producer` and `Consumer` execution time which is an unknown constant for all setups. Converting all results to single entry execution times is merely a practical usage, as it is easier to vary total entry counts and still make comparisons.

There will be 4 different setups to experiment with, as mentioned in Section 3. This 4 setups are the result of combinations of the 2 architectures (*GPU*, *CPU*) and the 2 memory transfer strategies (*COPY*, *POINTER*). This gives the 4 combinations *GPU COPY*, *GPU POINTER*, *CPU COPY* and *CPU POINTER*.

The `BUFFER_SIZE` of the two Communication Channels be kept constant to the given value of 3 tokens. The Communication Channels are by design always available for the `Calculator`. The `Calculator` code also has an error check implemented for blocking pipes. This check will raise an error if the `Calculator` fails to read from the First *FIFO* Communication Channel (underflow) or fails to write to the Second *FIFO* Communication Channel (overflow).

In addition to test for the efficiency of the `HDPC` framework and allowing to benchmark the overhead of the `HDPC` framework a second test setup is manually coded. The second test setup mimics the behavior of the explained Processes, but without the use of the `HDPC` framework for the Communication Channels synchronization and concurrency for example.

6 Implementation

Using an Ubuntu 10.10 32 bit USB boot-able test environment, together with *CUDA* Toolkit 3.2 allows us to run on systems without the need of installing software on the test systems, to avoid issues like porting the code to a different Operating System and have the flexibility to test on any *CUDA* enabled system, without altering the system installed state.

The code is written in `C++` compiled using `gcc 4.4.5` and *CUDA* compilation tools 3.2 and *NVIDIA CUDA* Driver 260.19.26. Using compilation the code has been build with all compiler optimization possible, except for loop-enrolling to ensure the "dummy load" will be valid.

The subset of options on the GPU has been limited to `TOKEN_SIZE`, `ENTRY_COUNT` and `LOAD_SIZE`.

A "framework" in `sh` shell provides caches of already compiled binaries and allows displaying results in a reproducible matter. More on this "framework" is found in Appendix A.

The "framework" also avoid endless running experiments. If a configuration is running longer than (the configured) 10 seconds the configuration will be terminated and there will be no results for the specific configuration. This behavior is implemented to allow running the computations in batch without the need of waiting for useless configurations. A (very) badly configured

configuration can take more than 1 hour of computation time.

CUDA device information has been inquired using the `deviceQuery` Utility provided in the *CUDA* Examples Toolkit, whereas `facter`[6] 1.5.7 did gather system specific information.

7 Experiments

The experiment was run on 2 different platforms (*GPU*, *CPU*) using 2 different strategies (*COPY*, *POINTER*) testing 1500 different configurations. The most important configurations variables are shown in Table 1.

The parameters are determined based on try and error as exploring the whole scope requires the setup to be running for years. This is caused by the 10 seconds needed to compile and execute a single configuration. This (re)compiling is required for each experiment setup as values like `TOKEN_SIZE` and `BUFFER_SIZE` in the `HDPC` framework are set during compile time. All experiments setup are repeated at least 10 times to ensure consistency in results.

Configuration Flag	Values
`LOAD_SIZE`	10, 100
`TOKEN_SIZE`	100.000, 1.000.000, 10.000.000, 16.777.216
`ENTRY_COUNT`	100.000, 1.000.000, 10.000.000

Table 1: Main configuration inputs used for experiments

Specification for machine used to conduct experiments; *CUDA GPU* GeForce GTX 295,Capability 1.3, 30 (MP) x 8 (Cores/MP) = 240 (Cores) @ 1.24 GHz. The *CPU* specification 8 * Intel Core™ i7 CPU 920 @ 2.67GHz, 3GB Memory.

For the sake of simplicity in the upcoming calculations assume that the overhead generated by the `Producer` and `Consumer` is $0ns$ per entry. In practice this overhead is a fixed constant per entry, causing the actual time spend by the `Calculator` alone to be lower.

The y-axis on all Figures are set to the time needed in nanoseconds to

[6] `http://www.puppetlabs.com/puppet/related-projects/facter/`

process a single entry, where lower means better.

An important property to consider is the so called *Tipping Point*. A *Tipping Point* is the intersection point between two graphs. The *Tipping Point* indicates when to consider an alternative architecture (*CPU* or *GPU*).

NOTE: Not all graphs can be (visually) compared to each-other as the input parameters differ, take good care of the type used on the x-axis of the graph. The x-axis type is either `TOKEN_SIZE` or `ENTRY_COUNT`. *Also* take care of the different values of the fixed parameters `ENTRY_COUNT`, `LOAD_SIZE` and `TOKEN_SIZE`. The actual values used for the fixed parameters are shown in the title of the graph.

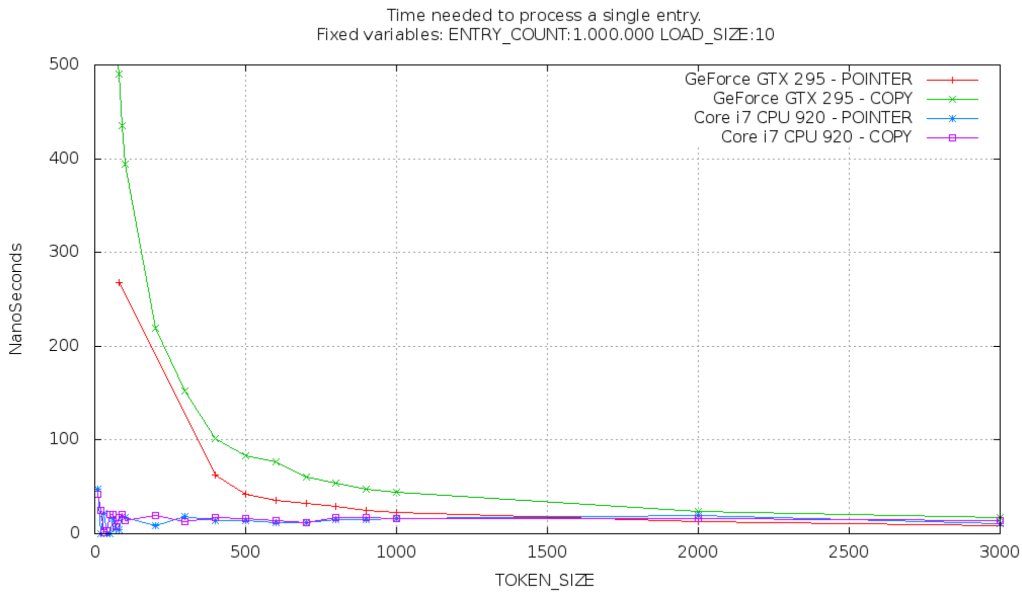


Figure 6: Experiment Result - The small value of the parameter `LOAD_SIZE` causes the *CPU COPY* and *CPU POINTER* to be almost equal.

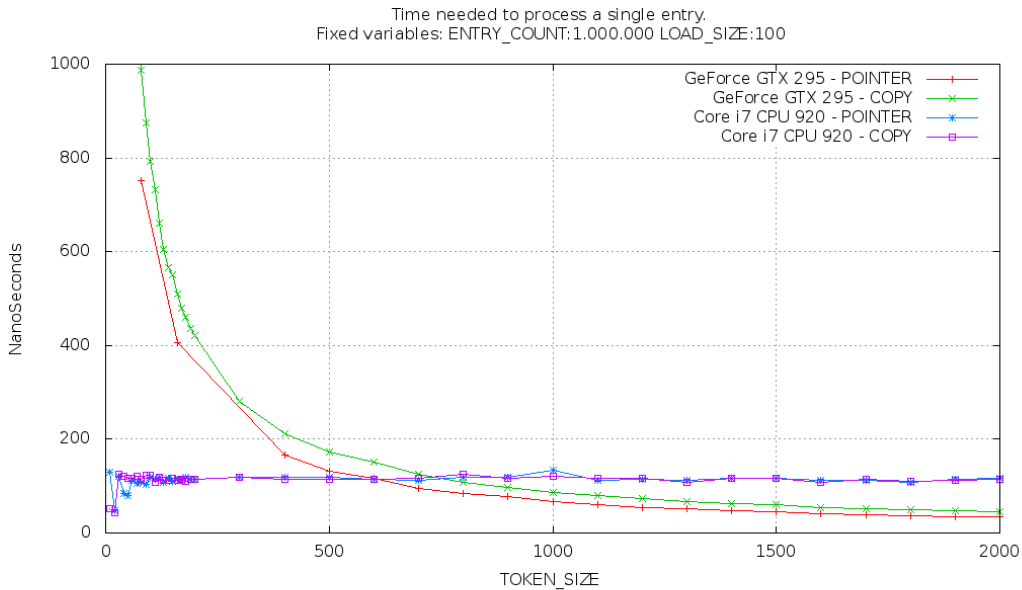


Figure 7: Experiment Result - Larger value of `LOAD_SIZE` causes the *COPY* and *POINTER* strategy for the *GPU* to be significantly different.

For small value of the `LOAD_SIZE` as shown Figure 6 there are almost no differences between the *COPY* or *POINTER* strategy on *CPU* except for the small `TOKEN_SIZE` values whereas the caches and context-switches are causing variances. For the *GPU* the *POINTER* strategy is always faster than the *COPY* strategy. This behavior is consistent for larger `LOAD_SIZE` values. The difference gets more obvious if the `LOAD_SIZE` increases as shown in Figure 7.

The reason the *GPU POINTER* and *COPY* graphs in Figure 7 are not shown for `TOKEN_SIZE` < 50 is because execution takes longer than the predefined cut-off time of 10 seconds.

Close inspection of the *GPU POINTER* graph in Figure 7 shows missing results in the range $50 <$ `TOKEN_SIZE` < 500. These values are missing as the program is unable to execute correctly this small `TOKEN_SIZE` input values. The reason for the behavior is unexplained.

Increasing the `LOAD_SIZE` causes the *Tipping Point* to "shift-to-the-left" in the graphs, seen relatively to the `TOKEN_SIZE` values (x-axis). For example on `ENTRY_COUNT` 1.000.000 and `LOAD_SIZE` 10 the tripping point is roughly at `TOKEN_SIZE` 3000 as shown in Figure 6, whereas for `LOAD_SIZE` 100 the

tripping is roughly at `TOKEN_SIZE` 750 as shown in Figure 7.

The best result found in Figure 6 is found at `TOKEN_SIZE` 2000 and is $14ns$ for the *GPU POINTER* strategy, all the other strategies are in the same range $14 - 16ns$. The best result in Figure 7 is again with the *GPU POINTER* strategy and is $42ns$. The *GPU COPY* is pretty competitive with $45ns$, but the *CPU* strategies are falling behind with $120ns$.

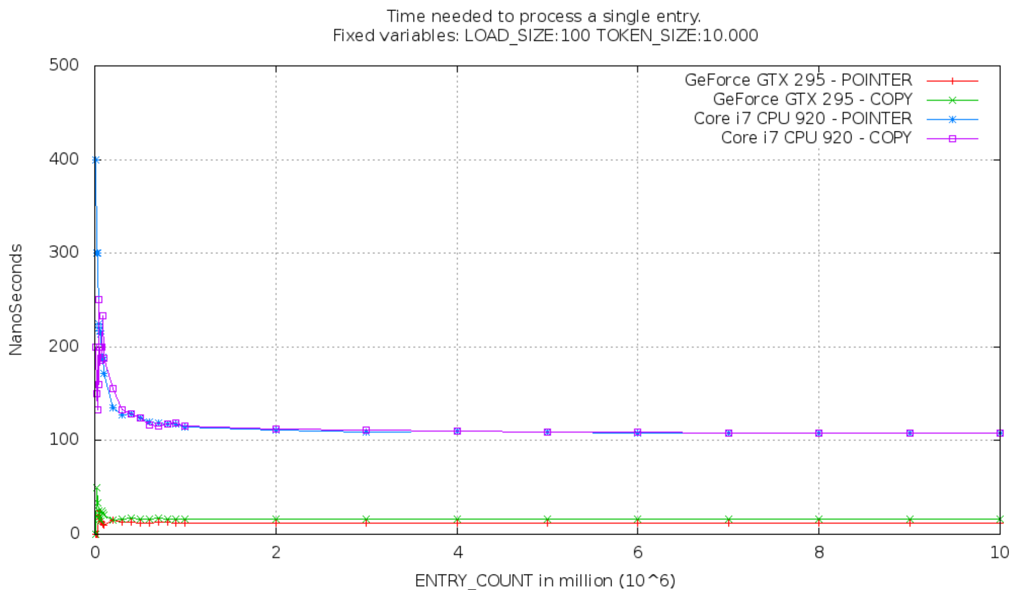


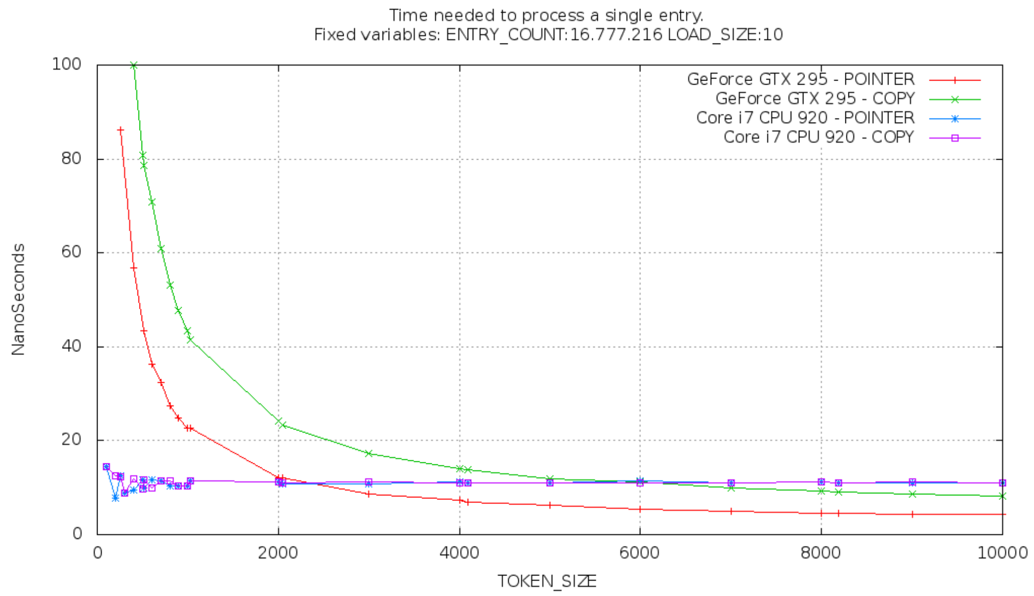
Figure 8: Experiment Result - Variants in the `ENTRY_COUNT` makes the execution stable after a while.

Figure 8 shows the hyperbola graph quickly approaches "a-stable-execution-time-per-entry-under-limit" no matter how many entries it get inserted. This gives best result,starting from $2M$ entries, for the *GPU* strategies of $14-16ns$ and for the *CPU* strategies $110ns$.

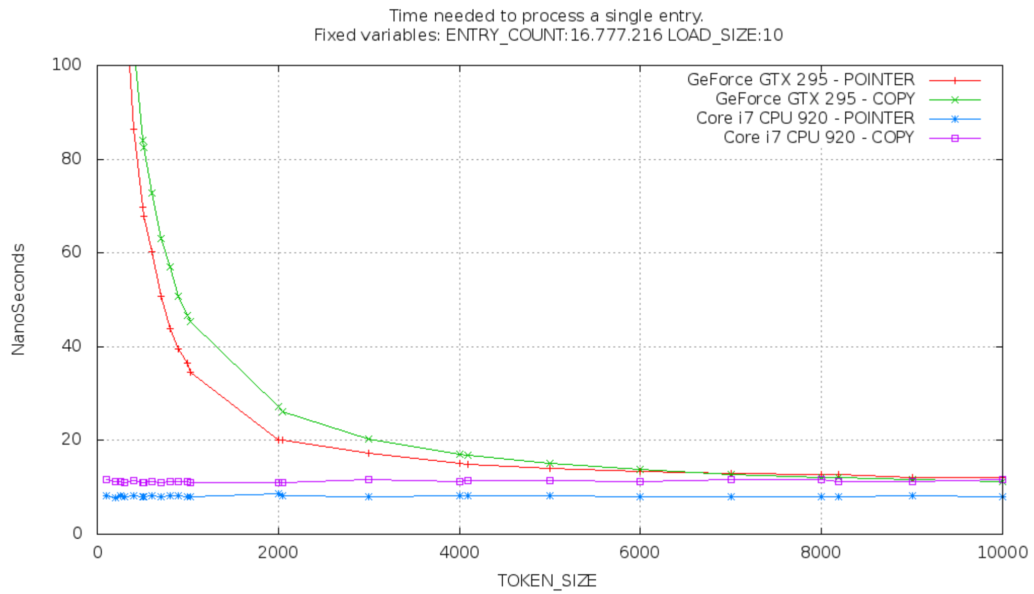
In Figure 9 it is shown that the *Tipping Point* for the `HDPC` implementation is better than the non-`HDPC` implementation (2500 vs 10000). A better *Tipping Point* allows the `TOKEN_SIZE` to be smaller with respect to the point where the *GPU* implementations is faster than the *CPU* implementations. The fact that the `HDPC` implementation is faster is because it uses a multiple *CPU* setup unlike the non-`HDPC` implementation, so the extra work done by scheduling and planning done by the `HDPC` pays off in a better performing program. In this experiment the `HDPC` version is almost 4 times more effi-

cient with regards to `TOKEN_SIZE` values. Even for small `TOKEN_SIZE` values (ranging from 2500 to 10000) the *GPU POINTER* strategy will speedup the computation. Looking in absolute values the *GPU POINTER* strategy will achieve a speedup of 2 times from $10ns$ to $5ns$ per entry.

The non-`HDPC` *CPU POINTER* graph in Figure 9 is lower in absolute execution time than the `HDPC` equivalent ($10ns$ vs $12ns$). The reason for this behavior is unexplained.

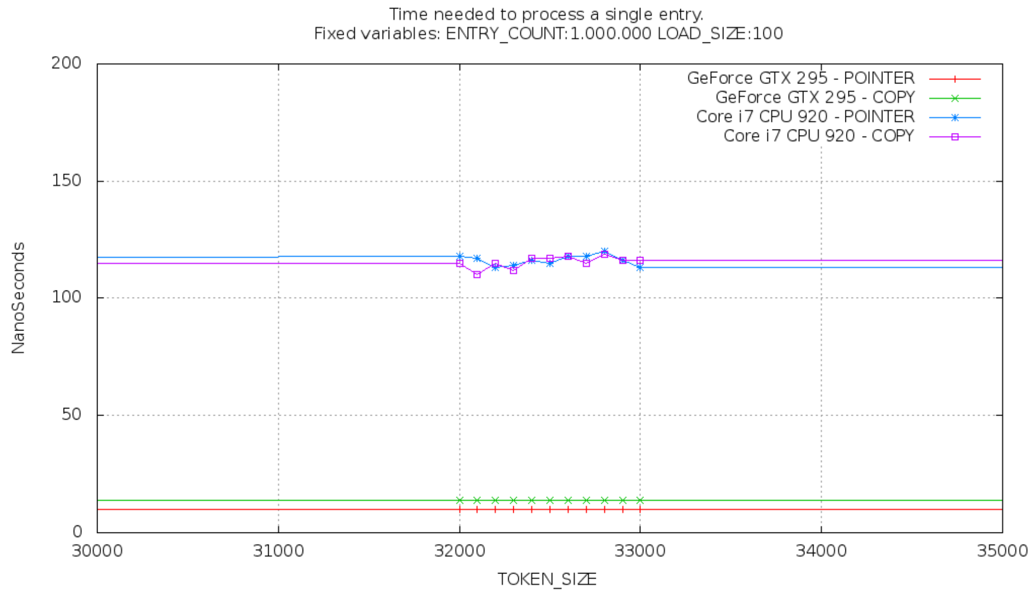


(a) HDPC

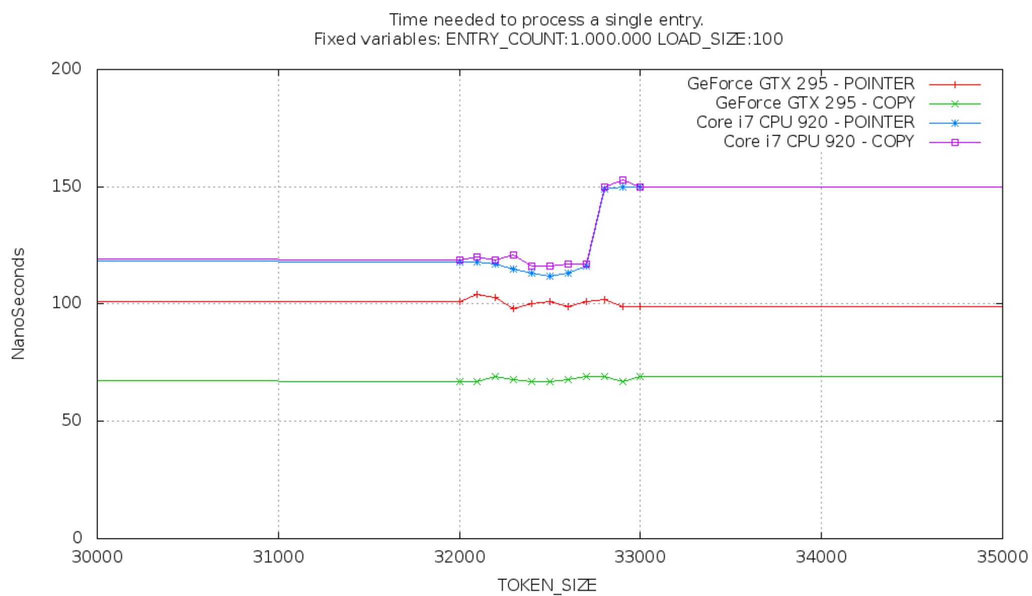


(b) non-HDPC

Figure 9: Experiment Result - The non-HDPC version lacks of multiple *CPU* support.



(a) HDPC



(b) non-HDPC

Figure 10: Experiment Result - *CPU* or Compiler optimization using `int16` for small `TOKEN_SIZEs` and `int32` for bigger ones, which only get detected in the non-`HDPC`.

Fig. #	Type	ENTRY_COUNT	LOAD_SIZE	TOKEN_SIZE	time
6	HDPC	1.000.000	10	2.000	14ns
7	HDPC	1.000.000	100	2.000	42ns
8	HDPC	1.000.000	100	10.000	14ns
9a	HDPC	16.777.216	10	10.000	4ns
9b	non-HDPC	16.777.216	10	10.000	10ns

Table 2: Best experiment results found for the *GPU POINTER* strategy. Column "time" list the time needed to process a single entry.

The expected theoretical value for `LOAD_SIZE` 10 is calculated using the method discussed in Section 4: $0.5ns+4ns*10+0.5ns = 41ns$. For `LOAD_SIZE` 100 this will become $0.5ns + 4ns * 100 + 0.5ns = 401ns$. Table 2 summarized the best results found in the experiments. The theoretical values are (much) higher than the best results found in Figures 6, 7, 8, 9a and 9b.

The reason for the (big) difference between the theoretical calculated values and the actual values is likely to be caused by better alignments of data on the *GPU*. The theoretical value calculated comes with the worst-case scenario of all data miss aligned. If the data were to be perfectly aligned, 1 read operation (600 clock cycles) would seeds $\lfloor 488bits/32bits \rfloor = 15$ threads of data. This makes the computation ≈ 15 times faster. These experiments come with partially aligned data, making it roughly 4 till 10 times faster.

The differences between the theoretical best values and the experiment best values also shows that determining accurate theoretical values (to be used in some analysis) is quite difficult and challenging. Systematic and (semi-)automated practical experiments are important to reliably validate the design decisions made with the theoretical calculations.

Looking at the experimental results there are three important observations; 1) The `LOAD_SIZE` has a more significant impact if the `TOKEN_SIZE` is small. 2) Calculation with `LOAD_SIZE` 100 seems to take roughly 3.5 times longer than the same experiment setup with `LOAD_SIZE` 10. The value 3.5 is likely tight to the specific *GPU* used, computing the exact value is unknown. 4) The best result is achieved with the largest number of entries and the largest `TOKEN_SIZE` and a moderate `LOAD_SIZE`.

NOTE: The *CPU COPY* and *CPU POINTER* strategies test results from the `HDPC` setup have a higher deviation than the non-`HDPC` graphs. The line plotted through the points in the graph is more irregular for the `HDPC` graph than for the non-`HDPC` graph as seen in Figure 9. This irregularity is best seen

in the TOKEN_SIZE range 0 till 1000. This deviation is most likely caused by the extra work required in the HDPC framework for Communication Channels synchronization, various statistics and context switching between *CPUs*. Pinpointing the exact component causing the spread turned out to be hard as adding more profiling code extra calls to the HDPC timer functions caused the program to behave differently and introduced even more spread. This gives a strong indication that the actual context switches are causing this spread.

NOTE: The compiler or hardware optimization can play tricks during the experiments. A specific *CPU* optimization was not detected in the HDPC code whereas it was detected in the non-HDPC code as this program was more simple for the compiler to understand. Figure 10b shows the interesting "jump" for the from $120ns$ to $150ns$ for the *CPU* based strategies at the non-HDPC exactly at the value of a short integer (int16) boundary $2^{15}-1 = 32767$.

8 Conclusion

The *KPN* generated by HDPC for Producer ⇒ Calculator ⇒ Consumer can use the *GPU* to speedup execution, if the amount of data to process is sufficiently high. A minimum 100.000 items (ENTRY_COUNT) is needed when using the NVIDIA GeForce GTX 295.

Choose the TOKEN_SIZE as large as possible. Every extra memory transfer from *Host Memory* to *Device Memory* or back has a negative impact on performance. Choose at least 1000 items to transfer on every memory transfer (TOKEN_SIZE), anything less is worthless on the *GPU* as alternative.

The dummy LOAD_SIZE used in the experiments is equivalent to a *very* large computation as global memory access was used to mimic high-load computations. A LOAD_SIZE of 10 is equivalent of roughly $600*2*10 = 12.000$ clock cycles on the *GPU*.

When you like to deploy your own code on *CUDA* run on initial check to test for complexity. The complexity (LOAD_SIZE) should together with your TOKEN_SIZE value give a "fair" amount of work for the *GPU* to compute.

To answer the first question formulated at the end of Section 3 (on page 14); Knowing in advance the amount of entries to process (ENTRY_COUNT) and the complexity of the computation (LOAD_SIZE) gives an indication of

whether to use the *GPU* at all. However, it does not give a definitive answer of the configuration parameter (`TOKEN_SIZE`) to use.

To answer the second question; The `TOKEN_SIZE` is the most important parameter during the process of designing and implementation of running (some) *KPN* nodes on the *GPU*. Determining the optimal `TOKEN_SIZE` during run-time is preferred, but requires more research of finding an algorithm able to make this run-time optimizations. Determining the (semi-)optimal `TOKEN_SIZE` in advance for a specific configuration,by means of exploration, allows significant speedups during run-time.

The `Daedalus` tools and the `HDPC` framework has been proved to be a useful tool to quickly and efficiently build a framework application. The *KPN* and the choice of `HDPC` for having a dedicated *CPU* thread for controlling the *GPU* implementation of a *KPN* process has a positive effect on the performance of the program. The `HDPC` has proven to be a flexible framework allowing model variations of the `TOKEN_SIZE` parameters in a consistent and controlled way.

One downside of the `HDPC` implementation is the static `TOKEN_SIZE` assignment during compile time, which require knowledge of the target architecture in advance.

The *GPU* is a very powerful co-processor, but its application area is somehow limited within general purpose computing. Off-loading "standard" *CPU* based processes and try to run them on the *GPU* without modification is not going to work. Significant effort needs to be spend in order to rewrite a process to fit in the highly parallel architecture of the *GPU*. Thus it is better to look at the data parallelism as basic-building block during the development of the application.

For the two *GPU* memory transfer strategies tested, the *GPU POINTER* strategy is preferred over the *GPU COPY* variant. 1) The *GPU POINTER* is equal or faster in overall execution. This makes the implementation in particular more interesting if smaller `TOKEN_SIZE` values are needed. 2) The *GPU POINTER* strategy has the advantage that memory transfer management is hidden in the *CUDA* framework making programming the code potentially faster and more dynamic. The wide variety of *NVIDIA GPU* cards around causes the *GPU COPY* code to be fairly specific as it requires knowledge of the card it is going to be executed on. The *GPU POINTER* implementation does not have this limitation.

9 Future Work

The *CUDA* Framework 4.0 has made improvements in concurrent computation, memory transfer and streamed transfers from one *GPU* to another *GPU* without invoking main memory, this is in line with the *Daedalus* framework, but is currently not implemented in the `HDPC` framework.

The *CUDA* Framework provides *GPU*-accelerated libraries for Basic Linear Algebra Subprograms (*BLAS*), Fourier Transform (*FT*), Spare Matrix and Random Number Generation (*RNG*) calculations. Using this libraries as basic building blocks in the *KPN* allows to use the power of the *GPU* more easily.

The *POINTER* strategy proved sufficient for simple programs with our `Producer` ⇒ `Calculator` ⇒ `Consumer` example. However for more complex concurrent examples the efficiency is still unknown.

The card used in this experiments is a high-end *GPU* Card, but not specially tailored for *CUDA* computations. The *NVIDIA* Tesla Cards are specially tailored for computations, running the experiments should give a whole different *Tipping Point*.

10 Acknowledgments

I would like to thank Hristo Nikolov at *LIACS* who was always ready to answer my questions and actively participating throughout my research with suggestions towards a better experiment setup and being a highly valuable knowledge in explaining the optimization quirks in the background both for the *GPU* and *CPU* as the system as whole.

References

[CUDA-PG] CUDA Programming Guide 3.2,
`http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/OpenCL_Programming_Guide.pdf`

[CUDA-BPG] CUDA Best Practices Guide 3.2,
http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf

[CUDA-GS] CUDA Getting Started Guide 3.2,
http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf

[Halfhill2008] Parallel Processing With CUDA, Tom R. Halfhill, 01/28/08,
http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf

[Far08] T. Farag. A framework for heterogeneous desktop parallel computing. Masters thesis, LIACS, Leiden University, 2008.

[DAC2008] Hristo Nikolov, Mark Thompson, Todor Stefanov, Andy Pimentel, Simon Polstra, Raj Bose, Claudiu Zissulescu, and Ed Deprettere, "Daedalus: Toward Composable Multimedia MP-SoC Design", Invited paper In Proc. "45th ACM/IEEE Int. Design Automation Conference (DAC'08)", pp. 574-579, Anaheim, USA, June 8-13, 2008.
http://www.liacs.nl/~stefanov/pdf/DAC_08.pdf

[Kahn1974] Kahn G. The semantics of a simple language for parallel programming. *ARTICLE of the IFIP Congress*, 74:471:475, 1974.

[Flynn1972] Michael J. Flynn,Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, 21:948:960, September, 1972.

A Re-run the experiment

In order to redo the experiment you need to setup a work environment. And Ubuntu USB Hard-Drive [7] setup is preferred. This setup is out of the scope of this Appendix but can be summarized as follows: 1) Run the standard Ubuntu Installer on a USB Hard-Drive. 2) Install the *NVIDIA CUDA* Driver and Development Tools. Find extra Ubuntu install hints in Appendix B. 3) Checkout the source code from the `LIACS` CVS Repository at `docs/students/hvdzwet/bachelor-thesis/experiment`. [8].

[7] USB Pen-Drive cannot be used, as the I/O is too slow when using it for compiling

[8] A public snapshot is found at http://rickvanderzwet.nl/svn/briareus/

The `HDPC` variant is found at `boost-hdpc`, the manual Implementation is found at `manualImp`.

You will first need to populate `id2device.txt` using the values you get from `deviceQuery` and `facter`, this file is used at a later stage for graphing purposes. Now run `new-experiment.sh`. This will execute all the variants used in this paper.

Next, to view the graphs run `view-plots.sh`. The output will point you to a directory where the graphs are stored. For settings boundaries on the graphs you can use the following environment settings `YMIN=0 YMAX=500 XMIN=30000 XMAX=35000`.

Please note that almost all scripts take a (ridiculous) set of (optional) environment variables to tweak running the experiments one way or the other to speedup execution times. So if you are planning to re-do experiments or run “interactively” look through the headers of the `*.sh` files to see what kind of flags you can set.

B Ubuntu Installation Hints

B.1 General Notes

1. You will get errors about libcuda missing if you have not installed the NVIDIA CUDA driver and using the standard Ubuntu provider driver instead.
2. If you do not want to remove your (default) Operating System, you can use a USB-DISK on modern computers. During the install of Ubuntu make sure to select the proper USB-DISK. Refer to your BIOS manual for instructions to boot from a USB-DISK.

B.2 Ubuntu 10.10

Make sure to install the packages to get started:

```
$ sudo apt-get install libboost-thread-dev libboost-dev build-essential \\
  facter gnuplot
```

The Ubuntu provided driver does NOT work with downloaded CUDA framework, neither the nouveau (open source driver) nor the provided proprietary driver. Install the dev-driver from NVIDIA found at http://developer.nvidia.com/cuda-downloads. Make sure to remove the nouveau first:

```
# <GOTO ttyv1 (CTRL+ALT+F1) and login>
$ sudo service gdm stop
$ sudo apt-get remove xserver-xorg-video-nouveau nvidia-*
$ <install DEV driver>
$ sudo reboot
```

Make sure to READ the release notes and add the vmalloc and uppermem in grub http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_Toolkit_Release_Notes_Linux.txt.

B.3 CUDA

Your profile need to be setup correctly to allow compiling:

```
$ export LD_LIBRARY_PATH=$LB_LIBRARY_PATH:/usr/local/cuda/lib
$ export LD_LIBRARY_PATH=$LB_LIBRARY_PATH:/usr/local/cuda/lib64
$ export PATH=$PATH:/usr/local/cuda/bin
```

The examples require a additional set of libraries:

```
$ sudo apt-get install libgl1-mesa-dev libxmu-dev libx11-dev
$ sudo apt-get install libglu1-mesa-dev libxi-dev libglut3-dev
```

Make sure to compile the helper libraries and the debug version of it:

```
$ make -C ~/NVIDIA_GPU_Computing_SDK/C
$ make -C ~/NVIDIA_GPU_Computing_SDK/C dbg=1
```