Optimization and Parallelization of CBD models

Konstantinos Theodorakos

Universiteit Antwerpen

Konstantinos.Theodorakos@student.uantwerpen.be

Abstract

This document presents methods to parallelize Causal Block Diagram models by using dependency graphs and model transformation techniques. Depending on the intention and the structure of a CBD model, output traces can be produced for various parallel languages and frameworks. The resulting source code is based on Domain, Task Decomposition and Pipeline parallel patterns.

The pipeline parallel programming pattern is applied in the case of delay dependencies between the blocks of the original CBD model. The generated pipeline uses computational overhead calculations in order to generate almost equally spaced stages.

Given enough simulation time steps, a speed up to 6x can be achieved, even for relatively small CBD models. The resulting parallelized source code uses the Intel Thread Building Blocks, OpenCL, OpenMP frameworks that run on modern CPU, GPU and DSP compute devices. After the execution of the output code, it is also possible to generate a custom OpenGL graph transformation to visualize the input and output values into a three dimensional cartesian system.

Keywords: CBD, Parallel, Dependency Graph, Parallelization, Computational Overhead, Optimization, Block diagrams, Domain Decomposition, Task Decomposition, Pipeline, GPGPU, Compute Device, OpenCL, OpenMP, Intel Thread Building Blocks

1. Introduction

Why should we attempt to parallelize a CBD model or any piece of code? The answer is simple but it is based deeply on a computer's architecture. It is the electrical power required to drive a modern central processing unit. As also shown in Figure 1, an system with a 2-processor architecture performs exactly the same as a system with 1 processor, but with the 40% of the required power.

Causal Block Diagram models (CBD) are graphs with connected operation blocks.The blocks can be algebraic expressions like the "Adder" and the "Product" block or provide the notion of time in the computation space like the "Delay", "Integrator" and "Derivator" blocks do.

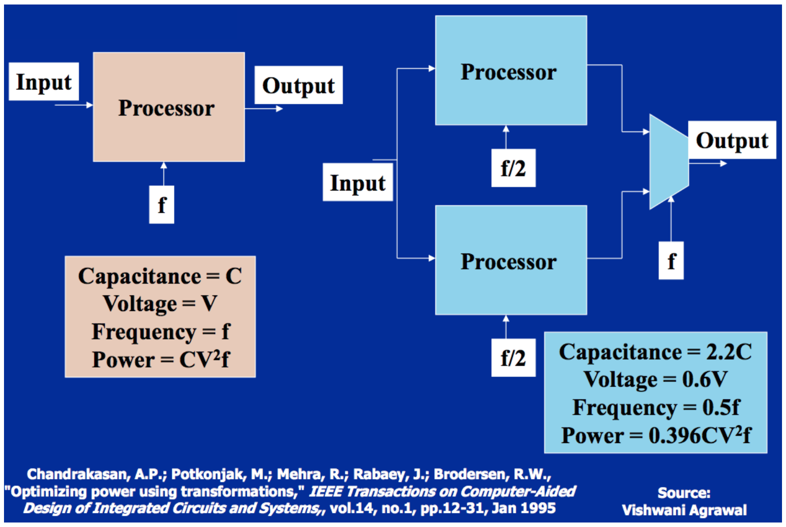

Input
Processor
Output
f
Capacitance = C
Voltage = V
Frequency = f
Power = CV²f
Input
Processor
f/2
Processor
f/2
Output
f
Capacitance = 2.2C
Voltage = 0.6V
Frequency = 0.5f
Power = 0.396CV²f
Chandrakasan, A.P.; Potkonjak, M.; Mehra, R.; Rabaey, J.; Brodersen, R.W., "Optimizing power using transformations," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,, vol.14, no.1, pp.12-31, Jan 1995
Source: Vishwani Agrawal

Figure 1: The system with 2 processors requires 40% of the energy of the one with the single processor, but with the same performance

So what is the intention of Parallelising CBD models? There are several reasons why we would like to parallelize Causal Block Diagram Models:

- As fast as possible simulation of large CBD models.
- Design Space Exploration (DSE). With DSE we can determine which parameter settings of input values can get optimal results.
- Shooting Problems (i.e find(a, b, c) when output = 2.5).
- Real time simulation.
- To perform real-time "brute force" calculations for systems that get sensory data in high numbers of SPS (Samples per Second).

- Resulting source code for CBDs when using model transformations is already optimised for fastest execution.

Parallel computing provides cooperation in a system by using shared resources which can be processors/cores of each processor, the main Memory and external compute devices like Graphic Processing Units (GPU), or Digital Sign Processors (DSP).

The rest of the paper is organised as follows:

- Section 2 provides background information about the Causal Block Diagram models, Parallel computing/devices and architecture as well as general information about the pipeline pattern.
- Section 3 presents the data decomposition technique for CBDs and GPU/CPU techniques in applying this pattern.
- Section 4 describes the Task Decomposition method, its downsides and why it wasn't selected for CBD model transformations.
- Section 5 examines the Pipeline pattern and its stages in depth and analyses the author's methodology to decompose tasks and generate equal pipeline stages on a CBD model. It also contrasts code generation and decomposition techniques from other authors.
- Section 6 provides some details about the implementation: parallel languages and framework used in the related work.
- Section 7 presents the experimental simulation results of CBD models and discusses the execution times in full detail.
- Finally Section 8 concludes and mentions about the Future Work.

2. Background

2.1. Causal Block Diagrams

Causal Block Diagram models are graphs with connected operation blocks. In our case a Time Slicing simulator is being used in order to simulate a system of interconnected signals. The blocks can be algebraic expressions like the "Adder" and the "Product" block or provide the notion of time in the computation space like the "Delay", "Integrator" and "Derivator" blocks do.

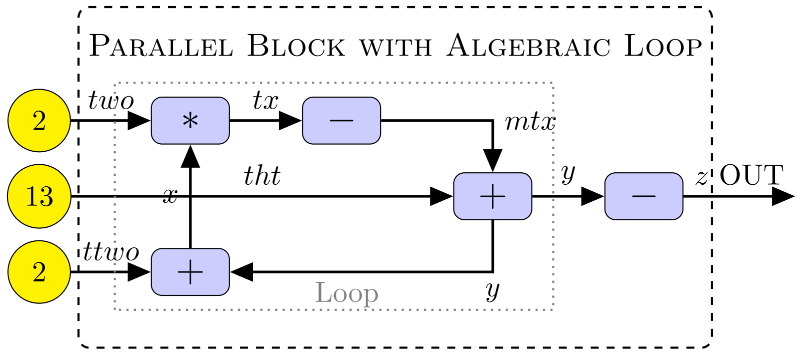


Figure 2: A CBD Model with a linear algebraic loop

The time slicing simulator, in order to successfully perform every execution iteration, requires an order in which the blocks need to be computed. It can be determined by abstracting the block diagram into a dependency graph.

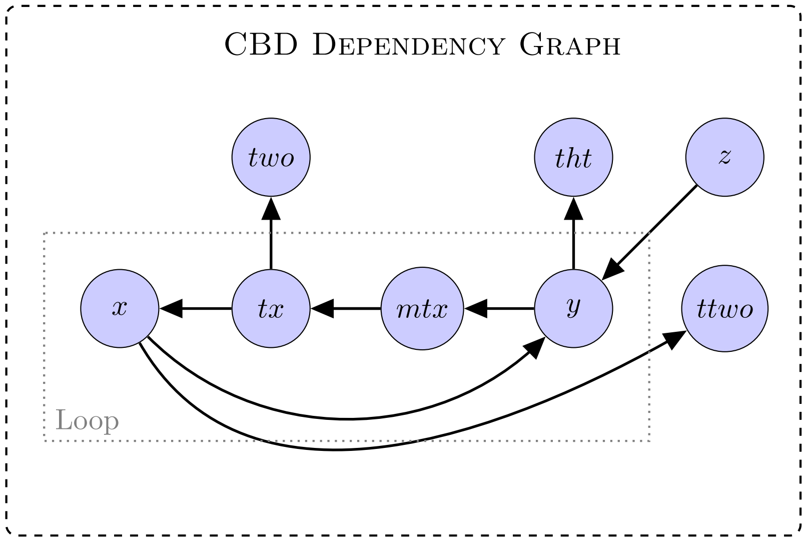


Figure 3: The dependency graph of a CBD with an algebraic loop $O(n + e)$

The dependencies of signals that contain a form of time delay, don't show up in a dependency graph, as these blocks relate signal values at different time instants. The **Topological Sort algorithm** can give an order in which the blocks can be evaluated to provide correct simulation results.

2.2. Parallel Computing

"The free lunch is over" as computer engineers commonly say. This means that it is no longer possible to maintain high level of efficiency and optimization on the hardware scale. The programmers should learn to take full advantage of the highly parallel architecture in computer's hardware. History though has proven that achieving efficient parallelization is a complex problem, some cases this achievement would reach "magical" level proportions if we attempt to create a generalised solution that would apply efficient parallelism on every possible case of model transformations. However, in the case where the construct that needs to be parallelized is well defined and under predictable limits, then a targeted parallelization methodology would probably be feasible. In this paper we will study how to take advantage of the CBD formalism and attempt to acquire legit and effective parallel code transformations from different kinds of CBD models.

Parallel computing provides cooperation in a system by using shared resources. This is usually being done by threads or processes. Threads are more lightweight than the processes because they have less overhead. Threads manage to cooperate via:

1. Communication. Sharing information with the usage of common buffers-memory spaces.
2. Synchronization. Having atomic and protected regions while having common "check points" in order to prevent race conditions. Semaphores and Mutexes can help in preventing data corruption by providing locking mechanisms.

2.3. Parallel Computing Devices

2.3.1. GPU

A Graphics Processing unit is a device that usually has a collection of Computational Units and distinct memory. Thus it is considered as an unique device under an explicit context. Whenever there is a need to perform a computation using the Graphics Processing Unit, it is required to enqueue an execution command into the GPU's event based queue. In OpenCL, GPUs contain multiple levels of memory. The global/constant memory space can be shared between all the processing units of a GPU but it has the disadvantage of high overhead during I/O operations.

However, the GPU's cores can be split into work groups. These work-groups (which can be divided using 1, 2 or 3 dimensional space) contain the

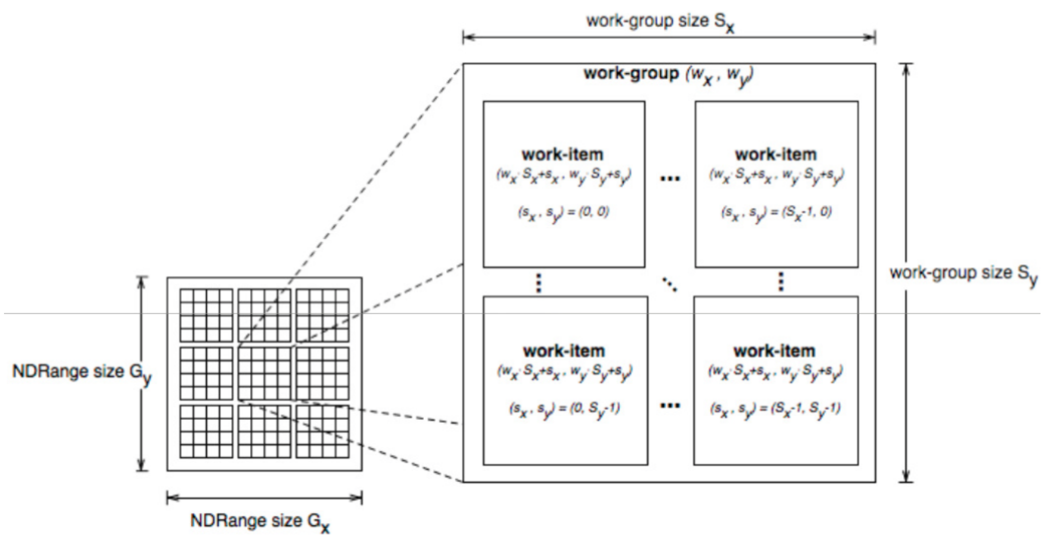


Figure 4: The workgroup abstraction for GPU compute devices by the OpenCL framework

notion of the local memory (see Figure 4 for a 2-dimensional representation). This memory is faster than the global/constant memory and it is shared only between the work items of the same workgroup. Now, the fastest and most encapsulated memory is the private memory. It is contained within each work item/compute unit, it is not visible by other compute units and it helps in executing complex calculations by acting as a temporary and fast memory placeholder for local computations. The main advantage of the external compute devices is that, by default they are optimized to execute in massive and parallel task and data computations, based on double precision floating point scalar variables, vectors, raw buffers and image objects.

2.3.2. Multi-core Processors

In a shared memory computer typical applications are being run within the context of one process. Each process contains one main thread and several additional threads. These threads perform read/write operations into the shared address space of the process or the main Memory(Figure 5). In order to avoid corrupted results when multiple threads perform R/W operations over the same address space, syncing and some sort atomic operation regions are required. We must be careful though, when have to apply synchronization techniques . If syncing threads is overlooked and overused, it may introduce unnecessary slowdowns and overhead that would even cause the execution time to grow exponentially.

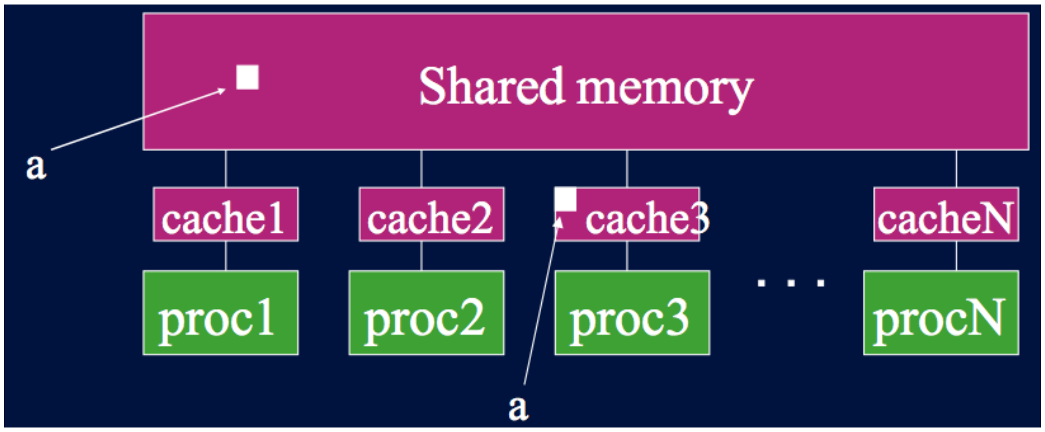


Figure 5: The shared memory model. Threads can have private and shared memory space

2.3.3. Pipeline

A Pipeline pattern connects tasks in a **producer-consumer** relationship. Conceptually all stages of the pipeline are active at once, each stage can maintain a state that can be updated as data flows through them. A linear pipeline is the basic pattern but more generally, a **set of stages** could be assembled in a **directed acyclic graph**. It is also possible to have parallel stages.

The stages of the pipeline can be generated by using functional decomposition on all the required tasks of an application. Though this approach leads to a fixed number, of stages so pipelines are generally not arbitrarily scalable.

A pipeline is a linear sequence of stages. Data flows through the pipeline, from the first stage to the last stage (usually is called a filter). Each stage performs operations on the data. The data are partitioned into pieces called items. A serial stage can process one item at a time, though different stages can run in parallel.

Pipelines are appealing because:

- Early items can flow all the way through the pipeline before later items are even available.
- Pipeline composition is straightforward. The output of a pipeline can be fed into the input of a subsequent pipeline.

- A serial pipeline stage maps a serial I/O device.
- Pipelines deal naturally with resource limits. The number of items in flight can be throttled to match those limits (using tokens as also used in the related work). It is possible for a pipeline to process large amounts of data, using a fixed amount of memory and threads/processes.
- Linear structure makes it easy to reason about deadlock freedom, unlike topologies involving cycle or merges.
- Each stage can be debugged separately.

A pipeline with only serial stages has a fundamental speedup limit, similar to Amdahl's law in throughput. The throughput of the pipeline is limited to the **throughput of the slowest** serial stage because every item must pass through that slow stage one at a time. In asymptotic terms, pipelines provide an asymptotic speedup. Though there is a hidden constant factor that can make a pipeline worth the effort. A pipeline with four perfectly balanced stages can achieve a speedup of 4. However this speedup will not grow further with more processors: it is limited by the **number of serial stages**, as well as the **balance** between them.

Parallel stages make a pipeline more scaleable (Figure 6). A parallel stage processes more than one item at a time. A parallel stage is different from a serial stage with internal parallelism, because the parallel stage can process multiple input items at once and can deliver output items out of order.

Parallel pipelines introduce a complication to the serial stages. In a pipeline with only serial stages, each stage receive items in the same order. But when a parallel stage intervenes between two serial stages, the later serial stage can receive items in a different order from the earlier stage. Some applications require consistency in the order of the items flowing through the serial stages and usually the requirement is that the final output order must be consistent with the initial input order. Data compression is an example.

3. Data Decomposition for CBDs

A GPU compute device can be enqueued with two types of jobs: in-order and out-of-order. In the out-of-order case we can have faster execution times but the computations are being performed throughout all their range by the compute units freely, without a specific sequence in execution.

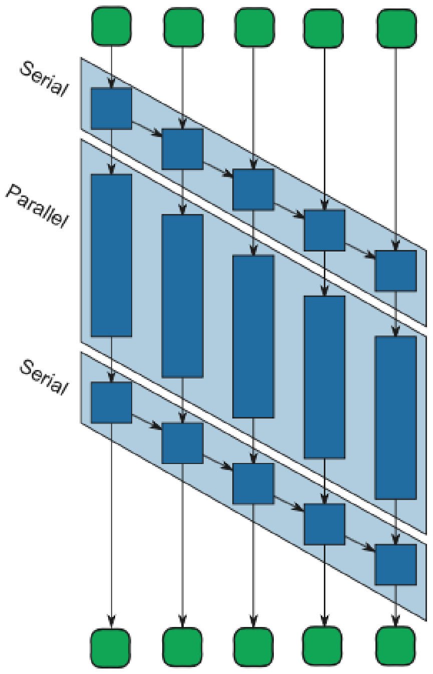


Figure 6: A pipeline that contains both serial and parallel stages

One of the limitations that the compute devices like GPUs may face on hybrid computing is that there is a certain upper limit on the **max size** of the **buffer memory chunk of data** that can be transferred from the main system to the GPU. As also found in practice, in the case of the 15 CBD blocks, the GPU could load at maximum enough data for around 25 million iterations whereas in the case of a CBD block model with 5 blocks it can reach up to 52 million iterations.

After that point though, ”memory out of range exceptions” occur. This is probably due to a hardware limitation. But how this issue can it be resolved? The solution would be to use the **iterator pattern**(Figure 7). The Data Decomposition part should remain the same but whenever there is a need to enqueue more data than the hardware memory buffer can handle in one transfer, data could be divided into a series of manageable sized chunks. In this case, after the first chunk of data has been transferred to the GPU and returned with the processed data, the GPU can be enqueued with the next chunk of data until it finally computes all the divided chunks.

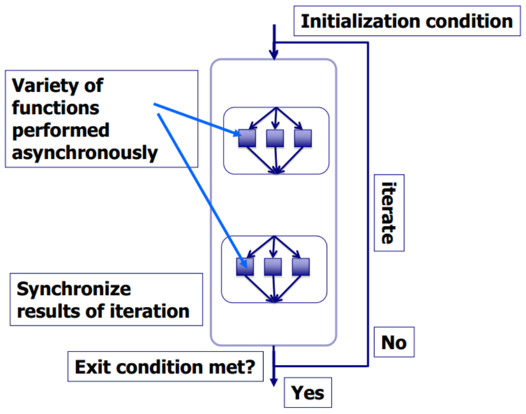

Initialization condition
Variety of functions performed asynchronously
iterate
Synchronize results of iteration
No
Exit condition met?
Yes

Figure 7: The iterator pattern can help in processing large amounts of date by splitting the parallelization into manageable data chunks

A technique, commonly used to compose and split tasks using the Data composition is the ”Parallel For” (Figure 8). This construct achieves - in some cases automatically - equal decomposition of the operations required not on a GPU or other discrete compute device, but on a typical computer system with the ”Shared Memory Model”. This way, using threads, a CPU can be utilized to almost its full extend and achieve increased performance at higher levels than compared to a GPU device, especially when the computational delay of each task is relatively light and a lot of communication between threads/tasks is required. In the related work, the ”parallel for” pattern was utilised using the OpenMP library.

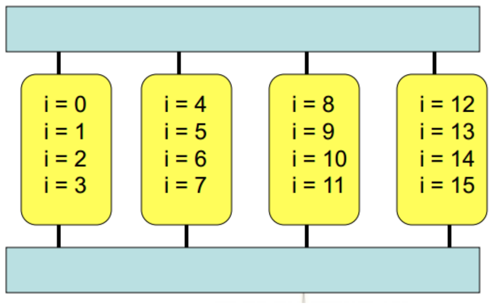


Figure 8: Data Decomposition with the ”ParallelFor” construct

One of the techiques used in splitting and assigning tasks to different threads/processes is the Round Robin algorithm. This allows each thread to execute in ordered turns, like dealing cards on a card game. Usually round robin is faster than splitting the job size of each thread into predefined chunks of iterations. This task splitting technique was used in the case of the parallel For pattern, in order to distribute tasks/iterations to each one of the requested threads. However, as also found in practice, the Domain Decomposition technique results in faster compute times compared to the Round Robin algorithm (around 1-5%).

4. Task Decomposition for CBDs

Task based decomposition is a very common and effective tactic in applying concurrency. But in the case of the execution of a CBD model simulation, task based decomposition is **not optimal**. The resulting task scheduling set often contains irregularities in its final form, which means that it is harder to detect concurrent patterns and to fully "parallelize" the execution. There are still "utilization gaps" as shown in Figure 9. However if the decomposed tasks are divided equally enough, considering the computational delay of performing all the operations of each task **computational delay**, we can apply the pipeline parallel pattern to try to achieve a better level of parallelization.

5. Pipeline Decomposition for CBDs

In the paper [4], a recursive branch-and-bound algorithm is being used to compute effective schedules. As it is mentioned, the core of the algorithm tries to decompose a dependency graph into strongly connected components and then attempts to find the optimal schedule by further decomposing each strongly connected component. While this algorithm is very effective on single-core systems, by its nature of being a **recursive algorithm**, this means that in an extended enough model, the available solution **state space will grow exponentially**. In contrast, having to simulate extended CBD models with many millions of time steps is where parallel approaches truly shine.

In the paper[5], it is mentioned that flattening a block diagram impacts performance since all the methods that perform operations on the entire flat diagram, can be very large. While this is true on serial execution, in a parallel execution environment, this is a desirable side effect. **Having a flattened** level of a series of commands **allows creating efficient pipeline stages to be easier**. Also not having multiple sub levels of function calling

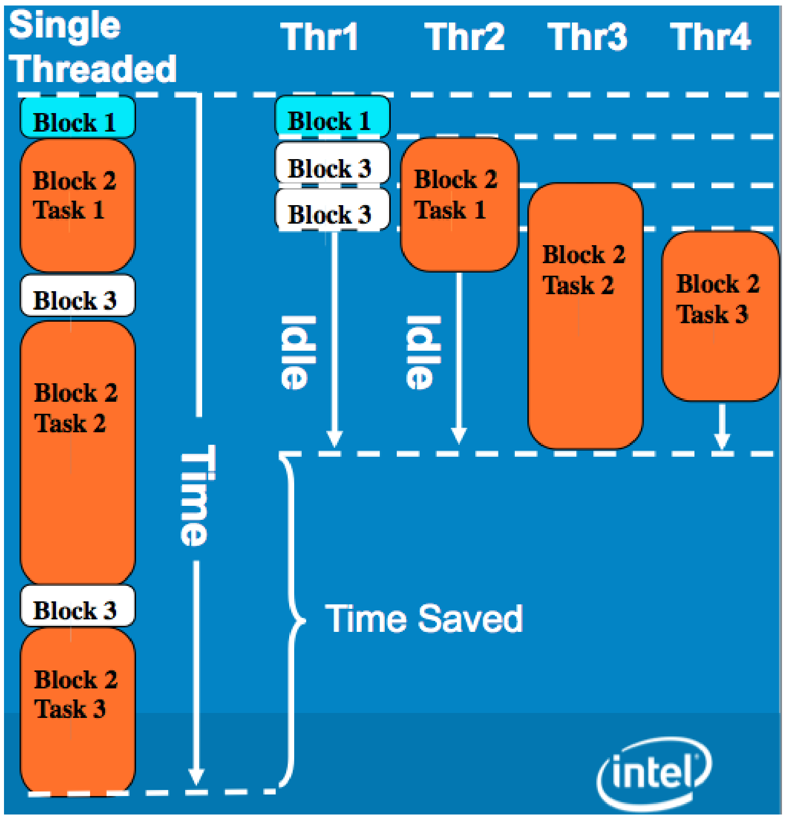


Figure 9: Even though the task decomposition technique increases the overall utilisation, it isn't 100% efficient due to irregular patterns and "gaps" between the execution of the tasks.

commands allows for better alignment with a Pipeline and a Data decomposition technique, because by nature, these techniques lead to flattened constructs which also favour the hardware (especially in the case of Share Memory programming, were we want to use the local thread/processor cache as much as possible). Threads preferably should perform operations on their local/private memory space and they are favoured even further if they don't have to deal with many levels of deep function calls.

Finer granularity on individual tasks provides more options when attempting to group many tasks, especially in the case of serial pipeline stages. For similar reasons, generating modular code isn't optimal for a parallel execution scheme. Flattening of course destroys modularity and may cause IP issues but in cases where such concerns don't exist, **having modular code**

would hinter performance of a parallel execution even to a point were it can't be optimised after a certain threshold. Modularity is also examined by the paper [6] with the MRCT task schedule generation algorithm. With the TRCM algorithm, the resulting task set pays greater respect on the timing metric compared to MRCT where it is primarily based on modularity. As shown in TABLE I. of the same paper, the TRCM execution has lower latency versus the MRCT, which is another indication that modularity introduces a decline in the performance compared to the flattening methodology.

Now, in our case we will use a much simpler solution. We will extend the normal task schedule generation that a CBD inherently has. So, we just want to apply the Depth First Traversal algorithm (Figure 10) on the original CBD model, in order to get the Dependency Graph. After that we will apply the topological sort algorithm to get a valid sort order by also achieving the minimum calculation order (order O(n+e)).

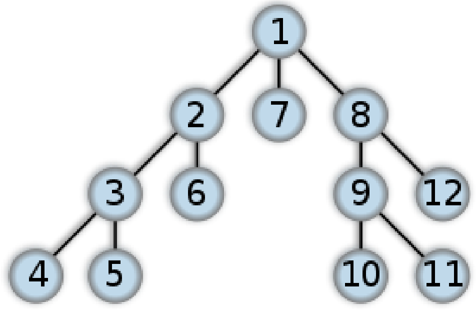
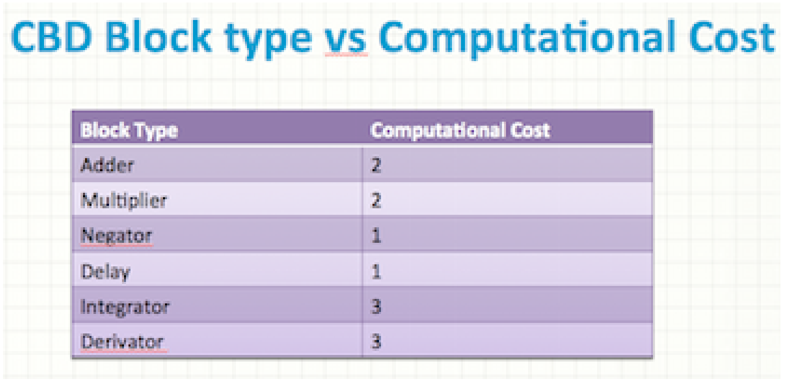


Figure 10: The order that the Depth First Traversal algorithm follows

We will use a relatively naive technique to compare 4 possible distributions and pick the most equally spaced one (Figure 12). On the first 2, we start from the first element of our set. Lets say we need 4 pipeline stages. We consider the first element to belong to the first pipeline stage. After that, we will add the next computational value to the total sum of the group and also the next task into the first group. We will keep doing that until the sum of the group just exceeds the average computational weight of the whole pipeline. The next element will be put into the second group and the sum of the 2nd stage will start as zero of course. We will keep doing that until we reach the end of the task scheduling set. This is the first distribution. For the second distribution we follow exactly the same procedure as the first but with one difference.

After we add the task that makes the current set to exceed the average weight of a stage, then, we remove that last task from the stage and we add it to a newly created stage. Thus, in the 2nd case, the sum of all the computational weight values for the 1st pipeline stage should be less than the "ideal stage size" and of course in the 1st case, the computational delay sum of the 1st pipeline stage elements will be greater than the "ideal stage size". For the third and fourth case we perform similar actions as the first and second way of dividing the tasks but with one difference. We start from the bottom of the task set and we move to top, not from the top/first to the bottom. The ideal stage size in order to create stages with exactly equal computational delay is calculated by:

$$IdealStageSize = \frac{\sum_{j=1}^{taskCount} computationalCost_j}{numStages}, \tag{1}$$
$$0 < numStages \leq taskCount, numStages \in \mathbb{N}$$

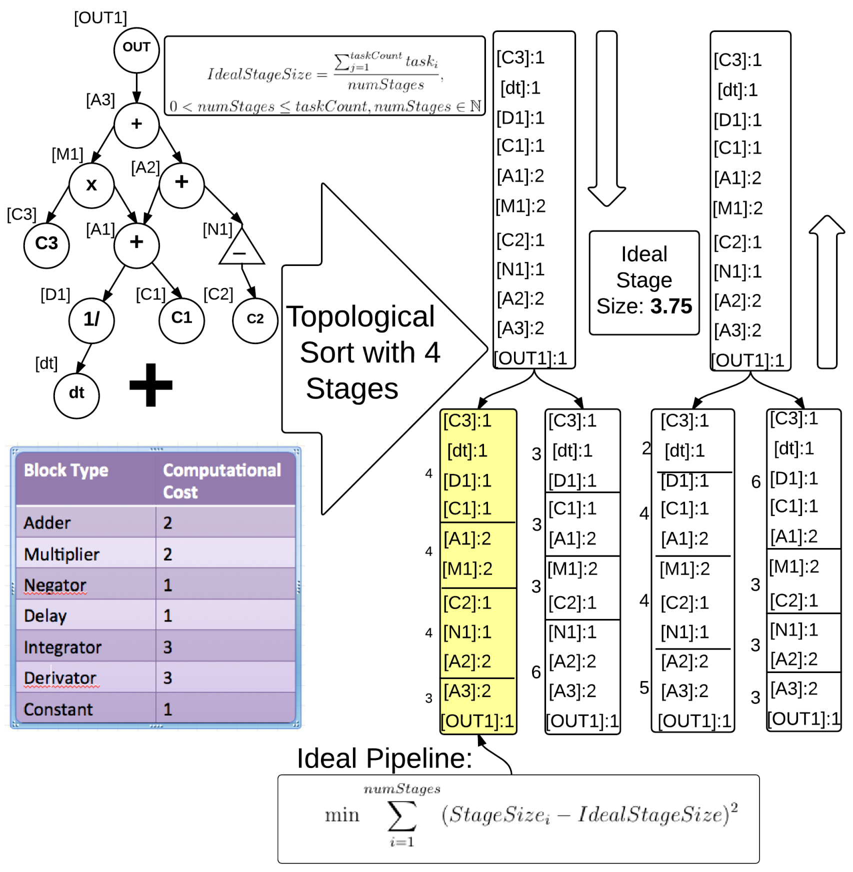

CBD Block type vs Computational Cost

Block Type	Computational Cost
Adder	2
Multiplier	2
Negator	1
Delay	1
Integrator	3
Derivator	3

Figure 11: Sample computational Weight for CBD blocks

Finally we end up with 4 set of pipeline stage groupings. All is left now is to use a similar to standard deviation technique in order to detect which one of the 4 cases has a distribution closer to the ideal average stage size. We can do that by adding all the: (sum of all the stage elements - average stage sum size) power of 2. We will end up with 4 numbers, one for each case. The distribution that has the smaller sum value means that it has the least deviation from the mean value. This means that it is the closest one to the ideal pipeline stage set. So essentially, to select the optimal case, we need the:

$$\min \sum_{i=1}^{numStages} (StageSize_i - IdealStageSize)^2 \tag{2}$$


[OUT1]
OUT
[A3]
+
[M1]
[A2]
x
+
[C3]
[A1]
[N1]
C3
+
−
[D1]
[C1]
[C2]
1/
C1
C2
[dt]
dt
$IdealStageSize = \frac{\sum_{j=1}^{taskCount} task_i}{numStages},$
$0 < numStages \leq taskCount, numStages \in \mathbb{N}$
Topological Sort with 4 Stages
Ideal Stage Size: 3.75
[C3]:1
[dt]:1
[D1]:1
[C1]:1
[A1]:2
[M1]:2
[C2]:1
[N1]:1
[A2]:2
[A3]:2
[OUT1]:1
Ideal Pipeline:
$\min \sum_{i=1}^{numStages} (StageSize_i - IdealStageSize)^2$

Block Type	Computational Cost
Adder	2
Multiplier	2
Negator	1
Delay	1
Integrator	3
Derivator	3
Constant	1

Figure 12: The procedure to generate a ”close to optimal” pipeline stage task distribution for the given CBD model

6. Impementation

Whenever a linear algebraic loop is detected with the usage of a reverse dependency graph, a Gaussian Elimination solver is being used. It manages to ”lift up” the blocks that are related in the loop and are being regarded as one ”singleton” block.

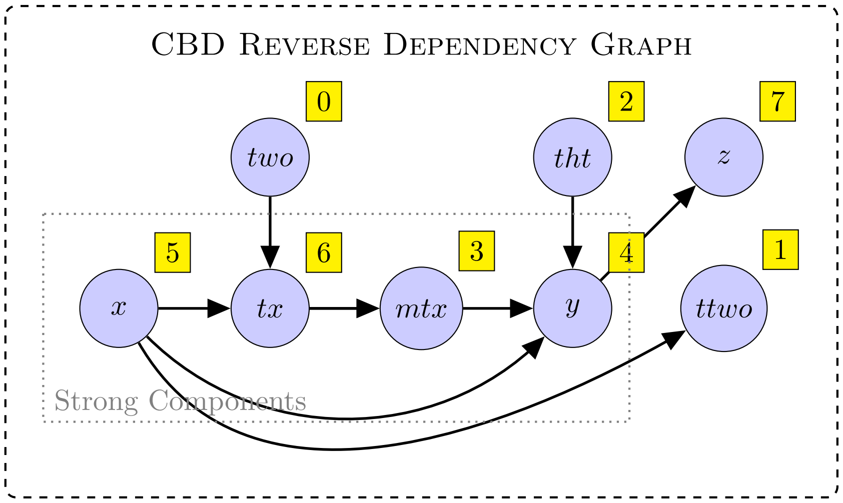


Figure 13: The Reverse Dependency graph of a CBD with an algebraic loop

As part of the CBD model transformations, Ansi C output code is being generated automatically and uses the GLUT/OpenGL framework to display a 3-dimensional preview of the input and output signal values (Figure 14). It is possible to rotate, scale in-out, hide-show the grid of the 3d cartesian visualisation. The following frameworks were used for the code generation:

- **OpenMP:** OpenMP Application Program Interface (API) is a portable, scalable model that gives shared memory-parallel developers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to supercomputers. OpenMP supports multi-platform shared-memory parallel programming in C/C++ on Mac OS X, Windows and Unix based operating systems. OpenMP uses pre-compiler directives and the notion of ”structured blocks” in order to distinquish private pieces of ”code” from the main thread’s code. Structured blocks use one or more statements to declare an entry/starting point as well as en exit point.

- **OpenCL:** OpenCL (Open Computing Language) is a multi-vendor open standard for parallel programming of heterogeneous systems that include CPUs, GPUs and other processors. OpenCL provides a uniform programming model by abstracting hardware details and thus providing to the software developers portable code that can run from handheld devices up to high-performance compute servers.

- **Intel TBB:** (Intel Thread Building Blocks) is an Open Source library created by Intel Corporation. It supports a wide range of platforms, operating system and even non-Intel based CPU hardware. Specifically for the Parallel Pipeline pattern it deals with 3 types of stages:
  1. **parallel**: process incoming items in parallel
  2. **serial out-of-order**: process items one at a time, in arbitrary order
  3. **serial in-order**: process items one at a time, in the same order as the serial in-order stages in the pipeline.

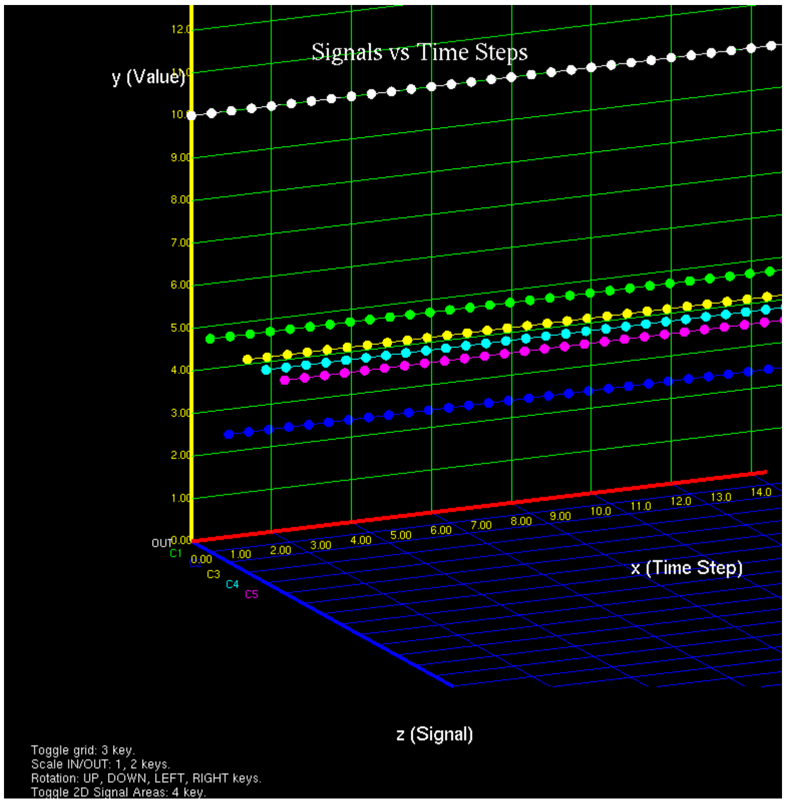
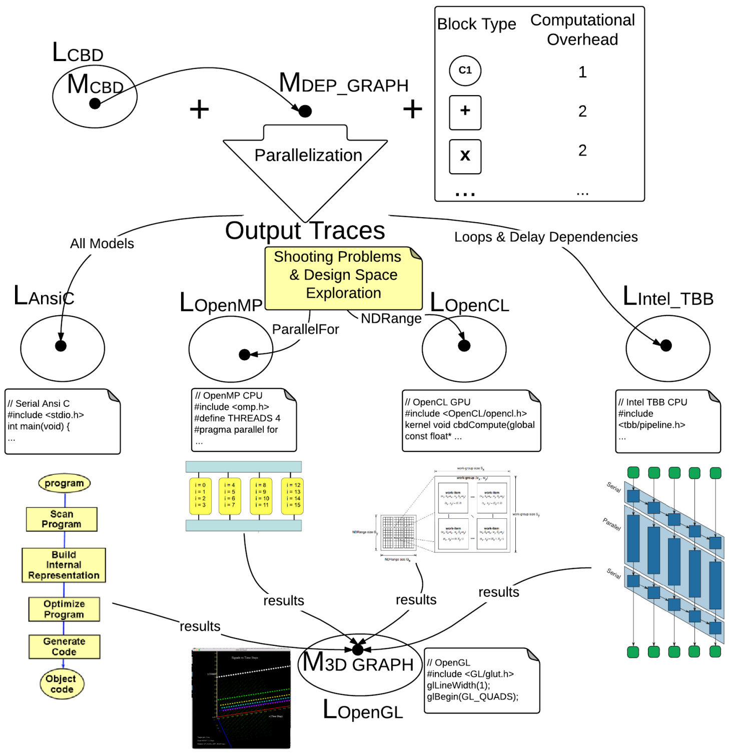


Figure 14: The generated 3D graph output using the OpenGL framework to draw the input and output signals

The good news is that the two kinds of serial stages have no impact on asymptotic speed up. The **throughput of the pipeline again is limited by the throughput of the slowest stage.** The advantage of the serial-out-of-order is that by relaxing the order of items, it can improve locality and reduce latency in some scenarios by allowing an item to flow that would otherwise have to wait for its predecessor.



```
// Serial Ansi C
#include <stdio.h>
int main(void) {
...
```

```
// OpenMP CPU
#include <omp.h>
#define THREADS 4
#pragma parallel for
...
```

```
// OpenCL GPU
#include <OpenCL/opencl.h>
kernel void cbdCompute(global
const float* ...
```

```
// Intel TBB CPU
#include
<tbb/pipeline.h>
...
```

```
// OpenGL
#include <GL/glut.h>
glLineWidth(1);
glBegin(GL_QUADS);
```

Figure 15: The Complete parallelization workflow

7. Experiments

7.1. Experimental Setup

The test machine that was used for the experiments, was a Mac Book Pro Mid 2012 (Processor: 2.6 GHz Intel Core i7, Memory 8GB DDR3, IGP: Intel HD Graphics 4000 1024MB, External GPU: Nvidia GTX 650M 1024MB).

We examine 2 test cases, with relatively small CBDs, both not having any delay type of block. The first had **6 blocks** and a Linear Algebraic Loop (Figure 7.2) and the second **16 blocks** (Figure 7.2) and no algebraic loops. The results show that given enough complexity and multiple iterations, a parallel universe of a CBD can outperform a serial based.

7.2. Results

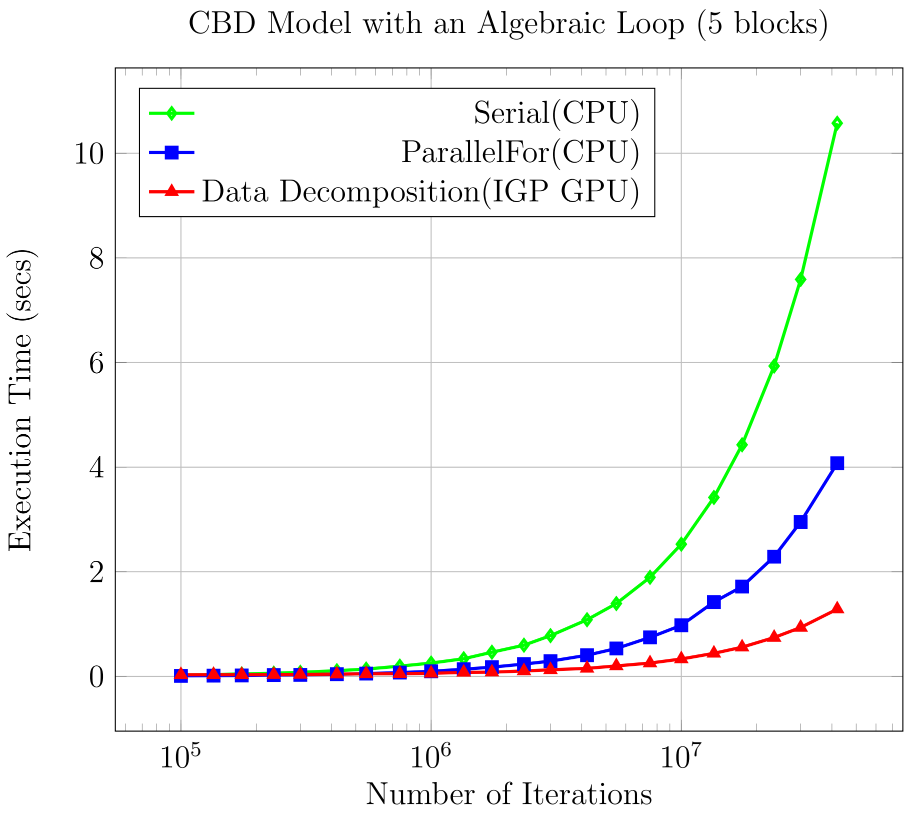


Figure 16: Simulation execution times for CBD Model with 5 Blocks and 1 Algebraic Loop. Machine used: Mac Book Pro Mid 2012 (Processor: 2.6 GHz Intel Core i7, Memory 8GB DDR3, IGP: Intel HD Graphics 4000 1024MB, External GPU: Nvidia GTX 650M 1024MB)

One interesting conclusion from the experiment results regarding the 2 different GPUs that a Mac Book pro has, is that, in most of the cases, the integrated GPU was faster at the execution than the discrete GPU (the

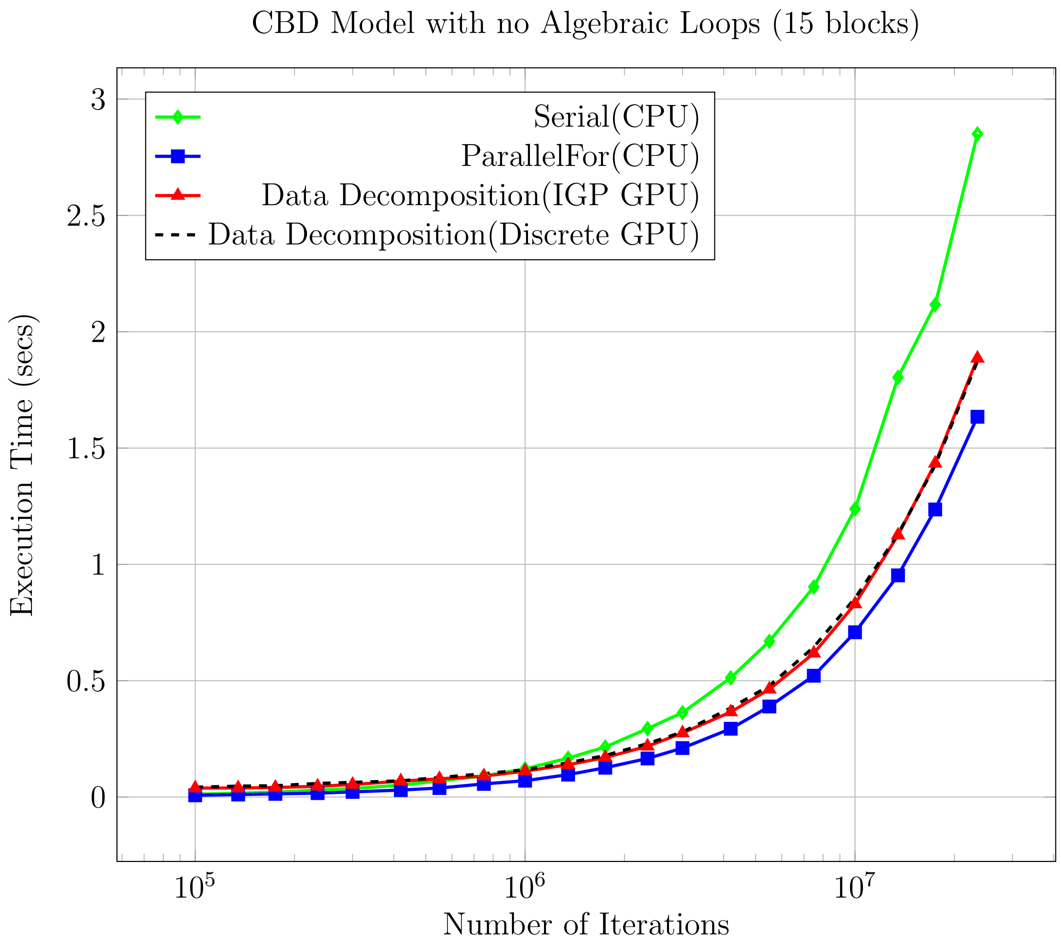


Figure 17: Simulation tims for CBD Model with 15 Blocks and no Algebraic Loops. Machine used: Mac Book Pro Mid 2012 (Processor: 2.6 GHz Intel Core i7, Memory 8GB DDR3, IGP: Intel HD Graphics 4000 1024MB, External GPU: Nvidia GTX 650M 1024MB)

discrete is specialized in 3D graphics processing). This is due to the fact that on a Graphics processing unit, the most "expensive" operation in terms of latency is the transportation procedure of all the initial raw data from the main memory to the GPU memory.

The internal GPU is faster on this operation because, in terms of hardware architecture, it is an integrated circuit on the mother board (or inside the

CPU itself in some cases) and by default, it is faster in I/O communication with the CPU than with the external GPU. Even though it doesn't use high quality fast Ram chips as discrete GPU devices do,the integrated GPUs use the same memory space as the main system (essentially the main RAM). So the operations to move data in and out of the memory buffer between the main system and a GPU is actually a virtual operation since both constructs use the shame memory address space.

In the CBD Model that doesn't contain any Algebraic Loops (Figure 7.2) but has 16 computational blocks, we can see a small but steady increase in performance, both in ParallelFor for CPU and in Data Decomposition for GPUs. When the iterations get close to 10^8 we can see almost a 1.5 times speed up for both parallel techniques. In this model also the Internal GPU is almost as fast as the external/discrete GPU. It seems that even though the IGP is slower in execution, the fact that there is no need to transfer input data from the host (the computer) to the compute device (for the Discrete GPU only) since it uses the same memory space, we result in almost identical performance.

Now in the case of the CBD Model with an Algebraic Loop (Figure 7.2)and 6 blocks, we can see that we have almost no gain in speed when the iterations are below 1 million. After that point though, we can clearly see that the performance gains resemble the mathematical formula $f(x) = x^2$. Especially in the case of Data Decomposition on the GPU, the execution time for nearly 10^8 iterations is almost 8x fold faster. The problem though is that after 10^9 iterations, the GPU must obtain divided memory chunks of data because it throws and out of memory exception. In the case of Parallel For with OpenMP (and a CPU with 4 cores), we can still see a good performance gain (for 10^9 iterations we get almost 2.5x speed up), not as good as in the case of the GPU, but we have the advantage of avoiding two type of overheads. One is splitting the input data into manageable chunks and the second is moving the data from the main memory to the compute device memory (we don't need those because Parallel For uses the same memory space in RAM for all the threads). The CBD Model tested was a relatively small (6 blocks) and still it is clearly visible that we can get very good performance gains, provided that our simulation has enough iterations.

8. Conclusions and Future work

The conclusion is that, it is indeed feasible to perform parallelization transformations of Causal Block diagrams that transform into multiple different frameworks and take advantage of the underlying hardware (see Figure 15 for the whole suggested parallelisation workflow). Data and domain decomposition was found that it provides **speed up** from **1x** (so it doesn't delay the serial execution) up to **6x** depending on the scale of the CBD Model and the number of iterations of the simulation. This make this appliance ideal in case of **Shooting Problems** and **Design Space Exploration**. One important fact is that, even though a deep knowledge of hardware architecture and parallelization patterns is required for parallel model transformations, the need to optimise the output parallel code at low level is required only once. These improvements can be reapplied automatically on all of the future output traces of the input CBD models.

For future work, a processor performance model of a cpu architecture will be also the input (together with the CBD Model) in the Parallelization workflow. This way, using parallelized simulations, the overhead of mapping CBDs to source code can be calculated. The choice of the technique (Data Decomposition, Parallel For and Parallel Pipeline) will depend on the intention, the type of the model as well as its limitations. I.e in the case of CBD models without Algebraic loops, we can use all these 3 techniques. But in Continuous Time CBDs, where Delay/Integrator and Derivator Blocks are being used, the only option is a Parallel Pipeline.

9. References

[1] OpenCL API 1.2 Reference Card, page 1

[2] OpenMP API 3.1 Reference Card, page 1

[3] Structured Parallel Programming - Pipeline, page 99, 253

[4] The semantics and execution of a synchronous block-diagram language, page 35

[5] Modularity vs Reusability: Code Generation from Synchronous Block Diagrams, page 1504

[6] Task Synthesis for Latency-sensitive Synchronous Block Diagram, page 10

[7] Heterogenous Computing with OpenCL, page 10

[8] OpenCl Programming guide for Mac, page 130-135