A Framework for Heterogeneous Desktop Parallel Computing

Tamás Faragó
Leiden Embedded Research Center
Leiden Institute of Advanced Computer Science
tfarago@liacs.nl, fafarago@gmail.com

Abstract

With industry moving to a multi-core design in search of performance gains, developers must incorporate this paradigm shift into their designs as well. We have so far mostly avoided the multi-threaded approach fearing the associated concurrency problems. Previous research at LIACS has created tools that automatically identify independent processes and their data flows inside an application. In this paper, we present a framework that uses these tools to generate not only concurrent versions of sequential applications on normal desktop machines, but also exploits resources of other, external devices, and truly, painlessly transforms the simple desktop computer into a computing powerhouse.

1 Introduction

Until a few years ago, the traditional approach to higher performance was to simply wait for the next generation of processors. With *Moore's Law* of exponential growth still holding, there was no need to invest in expensive tuning and developers enjoyed a "free lunch" of hardware improvements.

However, this era has come to an end with ever decreasing performance gains on single-core processors. Major processor manufacturers and architectures are running out of room with traditional approaches to boosting CPU performance, and they have moved to multiple-core architectures. Improvements can still be realised, but only with effort: software must be designed and implemented with concurrency in mind. *"The Free Lunch is Over"* [1].

Concurrency is hard. Developers are not comfortable with the programming paradigm, very few tools exist to assist, and the currently favoured implementation mechanism of using threads is dangerous. Any not-well written concurrent program is full of potential race conditions, deadlocks; and fixing these

is extremely hard. Another major challenge in implementing concurrency is scalability: how does the application perform as additional cores are added?

Amdahl's Law can give us predictions about scalability.

$$speedup(p) = \frac{1}{s + (1 - s)/p}$$

It states that the overall speedup as a function of the number of processing cores p depends on the serial portion of the code s, the critical path, and the parallelisable portion $(1 - s)$. This function shows a diminishing return as the number of processing cores increase, with each new unit contributing less and less to overall speedup. Even this formula is highly optimistic as it ignores any overheads and assumes full processor utilisation.

Only in "embarrassingly parallel" problems where little or no effort is required to find a corresponding parallel version ($s \approx 0$), can each new core contribute fully to the speedup. This is the case of data-parallelism.

Data-parallelism is relatively easy with already lots of research dedicated to the problem. Processor manufacturers have long supported the loop-parallelism approach through special instruction sets like MMX, SSE and various vector operations. Extensions for various platforms of the *openMP / MPI* standard are readily available, while companies such as RapidMind [2], and AccelerEyes [3] provide their own proprietary parallel platform API.

We are interested in task-parallelism. In this paper we propose techniques for solving above-mentioned concurrency problems in a systematic and automated way.

Part I

Research at Lerc group

Dr. Gilles Kahn introduced the Kahn Process Network (KPN) [4] terminology. KPNs are a distributed Model of Computation (MoC) where a group of autonomous processes communicate through FIFO channels. In KPN there is no notion of a global schedule that dictates the relative order of execution.

If we can identify the data dependencies between the statements of a sequential program, then we can create a KPN, map processes on different computing cores, and use this deterministic parallel KPN specification to execute concurrently, hopefully achieving higher performance than the sequential version [5, 6, 7, 8].

Specifying an application using a parallel MoC such as KPN, is a time-consuming and error prone task which is not well understood by developers. Therefore, we

need tools that allow us to continue writing sequential programs and automatically derive the parallel specification.

The DAEDALUS framework tools developed at LIACS are designed for this purpose. Accepting a KPN network as input derived from a sequential C/C++ program by the *pn* [7] tool, which facilitates migration from a sequential application to a parallel specification, the Embedded System-level Platform Synthesis and Application Mapping (ESPAM) [8] tool generates several autonomous processes that transfers data between each process through communication channels. Our group focuses on embedded systems, therefore for prototyping purposes ESPAM currently targets the FPGA technology. However, the tools are flexible enough to use different platforms as a backend.

Part II

Heterogeneous Desktop Parallel Computing

The framework presented in this paper, Heterogeneous Desktop Parallel Computing (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. The framework currently runs on the Microsoft Windows® Operating System.

HDPC improves upon ESPAM by generating backend code for a desktop computer that acts as the controlling and coordinating arbiter between the processes of a KPN. The processes can then execute on various computing devices like the FPGA, Graphics Processing Unit (GPU), or the Cell B.E. to take advantage of their respective strengths.

For each process - node - in a KPN, a thread on the host CPU is created. When the actual function(s) - computation(s) - 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.

Figure 1 visualises our approach; 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 and are under HDPC's control. Transfer of data happens by reading from the device's memory associated with

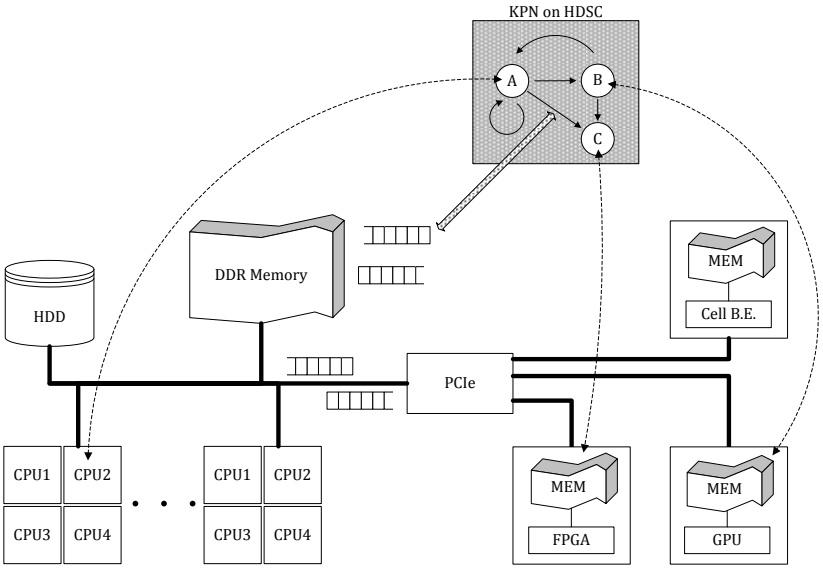


Figure 1: The Hdpc framework

the processing node into main memory, then writing this data in due time to the consuming node's memory. 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.

Our main purpose in designing Hdpc was to create a lightweight framework with minimal impact on performance and ensure correct execution of the running processes. While Espam only focuses on FPGAs as a backend, with Hdpc we can target a wide range of platforms.

As our framework acts as a controlling and coordinating framework of a KPN, all on a single machine, we have several additional advantages over the traditional task-parallel approach. KPNs for example do not allow, or even consider, global variables for communication between processes. As in Hdpc all communication happens in the same shared-memory system, the use of global variables is permitted. These can be used for example as read-only values for loop-bounds, constants, etc. Use of globals for communicating data between processes is possible as well, but as this happens outside of our framework it is the programmer's responsibility to ensure data integrity.

2 Design Flow

In this section we give an overview of our system design and the steps needed to create a working multi-processor application. Then, we explain the design

choices present to the programmer, their advantages and drawbacks. Finally, we show the usage of our framework through the simple example in Section 3 of Algorithm 1 on page 10, complete with the KPN graph in Figure 4 on page 10.

In Appendix E on page 40 we list the Application Programming Interface (API) available, as well as the UML diagram in Figure 17 on page 48.

We must stress that currently our framework is not yet integrated into the Espam toolchain. After integration, most of the presented steps detailed below will be obsolete. The designer will only have to make some choices regarding implementation philosophy and not need bother setting up the concurrent processes.

To build our network we need two specifications in the XML format:

1. An *Application Specification* that describes an application as a KPN, e.g. a network of concurrent processes communicating through FIFO channels.
2. An *Platform Description* that gives the topology of a multiprocessor platform. The topology decides which nodes are mapped to which computing units.

Once these specifications are available, we can start implementing the framework. In this section we explain the different choices available. In Section 3 we present how to construct a multi-threaded application in the form of a KPN using the Hdpc framework.

2.1 Communication Components

KPNs assume unbounded communication buffers but this is not possible in a physical implementation due to resource limits. The problem of deciding whether a general KPN can successfully complete with bounded memory is undecidable [9, 10]. However our tools only consider a subset of process networks, derived from SANLPs [8], which can be executed given a finite amount of memory.

The communication buffers are implemented using FIFO channels arranged as circular-buffers in memory. From an implementation point of view, a circular-buffer is not only more light-weight than a standard queue, stack or linked list, but also prevents memory fragmentation due to its "fixed" nature, and allows for data reference through pointers.

The only drawback is that we not only have to keep track of the read and write locations of a circular buffer, but also keep track of some usage counter. Otherwise when the read and write locations are the same we cannot discern the full state of a buffer from an empty state. The channels are internal to our framework, i.e. invisible to the developer, and created automatically of the proper type when nodes are connected through their ports and edges.

At the specification level KPN assumes unbounded FIFO buffer sizes. Reading is blocking, writing is non-blocking. At the implementation level, with limited resources, bounds must be set for each of the FIFOs, in which case writing to a full buffer is also a blocking action. The process can only continue when there is enough space in the buffer to finish writing all pending data.

Our framework includes two communication types for FIFO channels:

1. *Phyisical movement* of data. The read method fetches an element from the channel, and physically copies it to a local variable of the current process. Immediately after the transfer is finished the channel's use-count is decremented and that memory location is available for future access.
   Writing to a channel is analogous to reading. An element is physically moved from a local variable of the process to the channel and the usage count is incremented.

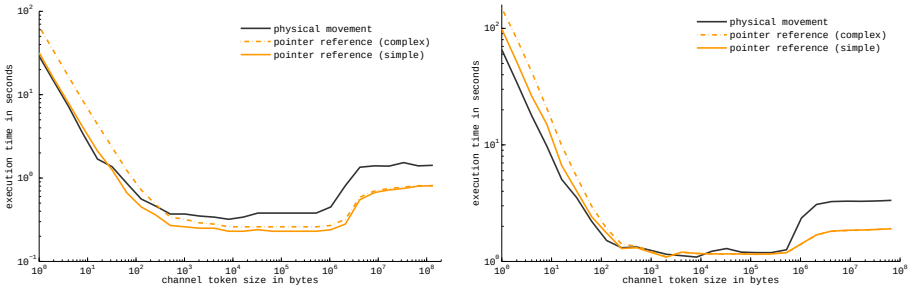
2. *Pointer reference* type is implemented through an acquire/release method. A read from the channel marks the location of the first token as in use and keeps a "lock" as long as data is needed from the channel. Similarly, a channel write acquires a "lock" at the write position.
   It is possible to acquire multiple consecutive locks for a channel as long as data/free space is available.
   All locks need to be *explicitly* released when a processing node is done with it in the current iteration.

There is a fundamental difference between these two types of channel access as we explain below.

In a shared-memory system (CPU to CPU), there is no need to copy data between the cores, everything can be referenced through pointers. Acquiring a read-pointer for the input and acquiring a write-pointer for the output argument(s) of a function, directly working at those locations, and releasing them when finished will remove this unnecessary overhead of copying data.

However, when executing processes on a heterogeneous system, this "trick" will no longer work. A GPU or an FPGA have their own separate memory spaces and we do need to physically move data. Even in this case it can however be beneficiary to acquire a read on the input channel. We can then copy directly from there to the device and release afterwards instead of first copying it some process-local memory and from there to the device. Of course, which method to use depends on device-transfer speeds and the control-flow dictated frequency of access.

Figure 2 shows the throughput of our framework comparing the physical movement and pointer reference method. For the second, two choices exist. The dotted line allows for more complex operations - explained in more detail in Appendix E.1.3 - but at the price of some additional overhead. For all tests



(a) block-free reads

(b) possible blocking on read

Figure 2: throughput on 1GB of data depending on token size

1GB of data was transferred through channels of different token sizes. What we can see is that when no blocking is possible the pointer reference method starts to achieve a higher throughput from somewhere between 16 and 32 bytes in the best case. This is logical since when you are working with pointers they are 4 - or 8 - bytes in size. Copying double words twice - once from the buffer to local memory, then to use the value - is faster than retrieving the pointer (same double word size) and an extra lookup for every access to this variable. It is important to keep in mind that the usefulness of pointer reference only starts to show with bigger chunks of data moving through the queues. Possible blocking - Figure 2b-, or complex usage increase the break-even point to about 256 bytes. The increase in time for large tokens is due to the cache sizes in the CPU.

The performance increase of the acquire/release mechanism also greatly depends on the channel size. The *pn* tool generates channels with a minimal guaranteed deadlock-free execution. If these sizes are used, we will actually observe lower speeds. This is because as transfers are faster, now more time is spent on communication and Operating System (OS) idiocracies such as context switches. Increasing the channel size when resources allow is the solution.

The minimal deadlock-free buffer sizes calculated by the *pn* tool only hold when the data is actually consumed from a channel before a write-operation is attempted. The pointer reference method can acquire a read lock and then a write lock before releasing the channel, thereby deadlocking execution. Therefore, channel sizes should always be increased at least by one token.

To summarise the communication components: physical movement of data is faster for smaller tokens, and less error-prone as it does not require manual release of locks. Pointer referencing is faster, needs larger channels, larger tokens, and requires additional control to release these locks.

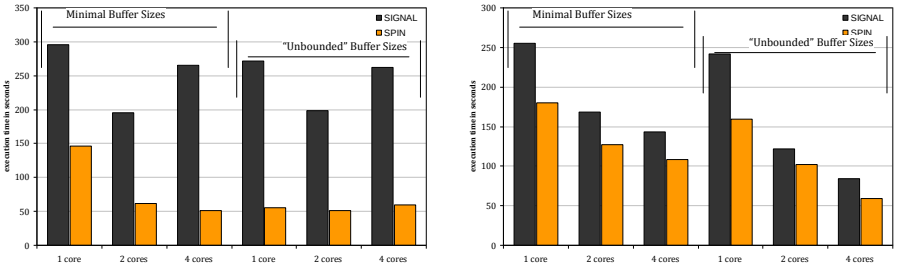
2.2 Blocking Read/Write Components

When communicating tokens through a channel, its state needs to be guaranteed. A consuming node has to block when the channel is empty and a producing node has to block when the channel is full. This control mechanism is implemented in HDPC within the channel component.

We can choose between two implementations:

1. *Signaling* uses semaphores provided by the Operating System. A process blocks and enters an idle state when the semaphore is not signaled; e.g. the channel is empty/full.
2. *Spinning* uses a single shared variable, accessed through atomic operations. This variable is accessed by both nodes of a connected channel to respectively increment and decrement the buffer count. A process blocks and enters a spin-state checking the buffer count until it has reached zero or the buffer size, depending on a read or write operation.

There is a big difference between these implementations. Using atomic variables is more than fifty times faster than semaphores [11]. This gives a much higher throughput as processes are notified much faster of the availability of a channel than when using semaphores. Figure 3 shows this difference in performance. How much this difference is in practice depends on the number of channels (in the graph fMRI has a lot more).



(a) fMRI application (b) MJPEG application

Figure 3: Difference in performance of *SPIN* vs *SIGNAL* for two applications

A disadvantage of spinning is the spin itself. In case there are other threads, the Operating System will not schedule those as frequently as it would do otherwise due to the continuous polling of the shared variable of the spinning thread. The method is also most certainly not power efficient. In our experiments the spinning method consistently produced better results. However the programmer is free to choose the signaling method when big amounts of data are communicated, especially to/from different computing devices, whose transfer rate will cancel the slow speed of the OS semaphores.

2.3 Platform Backend Components

To assist developers in easily using different computing devices besides the shared memory CPU and disk IO operations, we have added several platform backend libraries.

These libraries provide a general framework for communication and transfer to and from the device. Currently, these include a GPU[1], an FPGA[2], in addition to the standard CPU backend.

We allow developers to add their own backend which can then easily be used in future libraries. The disk IO library can easily be overridden through inheritance to support any custom file operations not just transfers of binary data.

Only `read()` and `write()` operations are exposed to the user. These transfer tokens to and from the device and accept as argument a source and destination pointer with at least one pointing to the address space of the device where the data resides when reading or will reside in the case of writing.

Developers willing to implement additional backends need to define additional functionality to (de)initialise the device as well as memory management. These API functions are explained in the Appendix, Section E.4.

We must note that the usage of the available backends is not mandatory. Users can choose to write their own communication, allocation, etc. code inside a node. However, when execution of a given node is to be changed to a different platform, large amounts of code need to be rewritten instead of simply switching one backend for another.

2.4 Network Correctness

One of the useful features of our framework is that the generated networks are checked for correctness, both compile-time and run-time. By using the template framework of the C++ programming language type-correctness is always enforced and we use the already available compiler features to show errors early at design time.

At runtime, HDPC performs two kinds of checks. Firstly, all connections are verified. This includes checking whether a connection is being redefined by accidentally connecting an already connected port again to some other node, and verifying that all ports have a valid connection. Secondly, during runtime type-checks are performed that make sure that tokens of the correct type are read from a channel; e.g. tokens of type integer are read from a channel of type integer.

[1] NVIDIA branded graphics card with CUDA support
[2] Virtex II PCI board

After execution of HDPC we verify that all the produced tokens in a channel are really consumed - that it is empty - only in which case we assume the execution to have performed correctly. All the runtime checks can be turned off at compile time, imposing zero overhead for execution.

3 Constructing and Implementing KPNs

Below, we show the creation of HDPC using the example in Algorithm 1 and Figure 4.

Algorithm 1 Example source code

```
int main() {
  double data;
  for (int i = 0; i < N; i++) {
    functionA(i, &data);
    for (int j = 0; j < N; j++) {
      functionB(data);
    }
  }
  return 0;
}
```

3.1 Construction

Construction of HDPC is based on the KPN graph generated by *pn* and the mapping choices of the nodes. We can observe two process, A and B. Process A has a single outgoing port, where B additionally has a self-loop; in total 2 incoming ports and one outgoing. So we create two instances of the Process class with the appropriate number of ports:

```
#include hdpc/process.h

Process A(0, 1);
Process B(2, 1);
```

Now these two separate nodes need to be connected. Connections can be created using the functions `attachoutput()` and `attachinput()`. The destination ports are retrieved through calls to `getInPort()` and `getOutPort()` of the other node respectively. Developers are free in their choice to connect output or input ports, and even use both at the same time. HDPC will complain

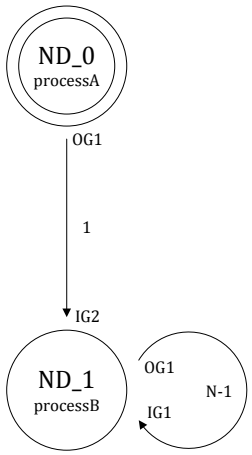


Figure 4: Example KPN graph

when an already existing connection is being re-
connected due to some user error.

```
A.attachoutput<double>(0, B.getInPort(1), 1);
B.attachoutput<double>(0, B.getInPort(0), N-1);
```

Creating the same connections, but now through the input ports. The size of the FIFO of the channel from A to B is one:

```
B.attachinput<double>(0, B.getOutPort(0), N-1);
B.attachinput<double>(1, A.getOutPort(0), 1);
```

Next, the choice must be made on what type of platform A and B will execute their calculations. Suppose we have selected to execute node A on the GPU and B on the host machine, the CPU. We create the two instances of these specific platforms. The constructor for the GPU accepts an optional parameter for which graphics card to use when there are multiple available on the system. Each backend needs its respective implementation included.

```
#include hdpc/platforms/cuda.h
#include hdpc/platforms/cpu.h

GPU gA(0);
CPU cB;
```

Now that everything is set up we can start our framework and the concurrent processes. Each process needs to be started and the computing unit attached. We will allocate memory on the GPU through our framework so we do not have to worry about this. Additionally we have fixed execution of process B to the third core in our multi-core setup and allowed Windows decide the best mapping for A - dynamically moving the thread from core to core as it sees fit.

```
Handles h[2];
h[0] = A.start(0x0, gA, processA, true);
h[1] = B.start(0x4, cB, processB);
WaitForMultipleObjects(2, h, TRUE, INFINITE);
```

When integration of HDPC is finished into the ESPAM framework everything will be correct by design. Checking as performed in Section 2.4 on page 9 will no longer be needed.
All there is left now is implementing the two processes including control flow. Running ESPAM automatically produces code for these processing nodes.

3.2 Implementation (Process Code)

Now that the network has been constructed we construct the actual nodes themselves. Channel communication components as well as computing back-end transfer functionality is used here. Control flow code was generated by

Each process receives as argument a reference to the `Process` class. All operations are performed through this reference. As discussed in Section 2.1 we can choose between physical movement or pointer reference for the data communication mechanism between the nodes.

- Physical movement needs a port to access and a reference to a variable for the data. `readFromPort()` and `writeToPort()` provide this functionality.
- Pointer reference works very similarly to physical movement. However, using this approach we need to acquire both the read and write references before executing the actual function. This is obvious since the function operates directly at these memory locations.
  We acquire a lock for the channel through `getRead/WritePointer()` and subsequently release them by calling `releaseRead/WritePointer()`.

  `releasePorts()` releases *ALL* ports for which an acquire operation has been made in the current iteration. The function can be used instead of the explicit release functions and is usually put at the end of an iteration. Using `releasePorts()` in this way sacrifices speed of release for less control flow. More explanation is given in Appendix E.3.4 on page 44.

The code for `processA` and `processB` is below.

After node *A* executes we need to read the results from the device. Only after this can we write the results to our internal buffer. Note that if we used the pointer method the read function would have been hidden implicitly used. In that case `getWritePointer()` returns a reference to device memory-space and on `releaseWritePointer()` the result is automatically transferred.

```
void processA(Process &proc) {
  // Output Arguments
  double out_1ND_0;

  for (int c0 = 0; c0 <= N-1; c0++) {
    for (int c1 = 0; c1 <= 0; c1++) {
      functionA(c0, proc.getDeviceOutMem(0));
      proc.getProcess()->read(proc.getDeviceOutMem(0), &out_1ND_0, 8)
      proc.WriteToPort(0, out_1ND_0);
    } // for c1
  } // for c0
}
```

B is using its self-loop to pass `in_0ND_1` around. Note that real code generated by the *pn* tool optimises this propagation out, the code using an actual channel is just for example purposes.

```
void processB(Process &proc) {
  // Input Arguments
```

```
  double in_0ND_1;

  for (int c0 = 0; c0 <= N-1; c0++) {
    for (int c1 = 0; c1 <= N-1; c1++) {
      if (c1 == 0) proc.readFromPort(1, in_0ND_1);
      if (c1 >= 0) proc.readFromPort(0, in_0ND_1);

      functionB(in_0ND_1);
      if (c1 >= 0) proc.writeToPort(0, in_ND_1);
    } // for c1
  } // for c0
}
```

4 Future Improvements

HDPC is a framework that allows for building multithreaded applications in the form of KPN targeting (heterogeneous) desktop platforms. Currently, building an application is done by hand, however, the goal is to be integrated into the ESPAM tool and the DAEDALUS[12] design flow which will lead to a highly automated design process.

The *pn* tool generates improved KPN networks in the sense that channel multiplicity is removed by the introduction of self-loops. Currently, the latest version of ESPAM ignores these self-loop hints and implements them as normal communication channels. However, self-loops have several properties that can greatly improve performance.

Firstly, a special case of a self-loop is when the channel is only of size one, called by *pn* a "sticky fifo". These channels can completely be removed and implemented as a single data element. Another important property of self-loops is that the same process will do the read and write operations. Therefore, these channels can never be empty when reading or full when writing tokens - assuming the generated KPN is correct - and have no need for any blocking mechanism. These non-blocking self-loop channels can be stripped of such overhead, further improving performance.

As briefly discussed before in Section 2.2, communicating with external computing devices, such as a GPU or FPGA, is preferably done through signaling semantics. Currently the choice is global for all channels of a KPN. It would be desirable to be able to choose the blocking mechanism per communication channel, either spin or signal.

Apart from these improvements, HDPC can be improved further by adding statistics capabilities. We could keep a count of the number of reads/writes, total data flow through a channel, watch execution time and time spent waiting for

tokens, etc. Further improvements are possible in the debugging functionality, detecting deadlocks in the case of incorrect Kahn Process Networks, etc. similar to the YAPI [13] simulation platform.

We will explore these possibilities in the future.

Part III
Case Studies

To show the benefits of our approach, we present three experiments and the results we have obtained. We have implemented and executed two image processing applications and a scientific data analysis program using our system design flow, *pn,* ESPAM, and HDPC synergy presented in Section 2 and 3.

These applications are a Sobel edge detection algorithm, a Motion JPEG (MJPEG) encoder and fMRI analysis. In Appendix A on page 24, we give more technical details about these applications as well as their KPN graphs.

All experiments were performed on an Intel® Core^{tm}2 Quad CPU Q6600 at 2.40GHz and 4GB of system memory running the Microsoft® Windows Vista^{tm} Enterprise Operating System (OS). We have verified the correctness of the results through a comparison of the MD5 sum of the sequential and parallel outputs of the case studies. Results are an average of ten runs of the application.

In the graphs, the dark bars show execution times with physical movement of data. The orange bars show the pointer reference method. In the (static) KPNs we consider, buffer sizes large enough such that they do not affect performance by blocking on a write: "unbounded".

We have set no constraints on which cores of our multi-core host machine the nodes could be executed; it was left to the OS to choose the best balance.

5 Sobel

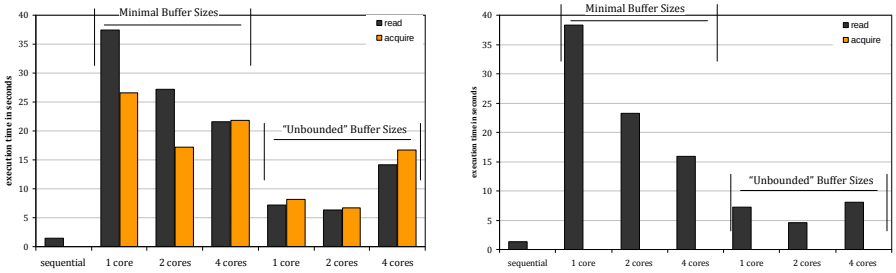
Sobel is an edge detection algorithm where a 3×3 window is slid over the image to calculate the gradient of the pixel with its neighbours. This means a very fine-grained control flow as can be seen in Figure 10 on page 24 in the Appendix, and therefore a substantial communication overhead on any parallel approach. Experimental image was 3072×2688 pixels in size[3].

We did not experiment with the acquire semantics on the version of Sobel with self-loops. As we had to manually translate ESPAM code for our framework, keeping track which channel keeps data when, was too much work.

Results in Figure 5 on the following page confirm our expectations. The sequential version is consistently much much faster and the high communication ratio kills any exploitable parallelism.

We can observe a few interesting trends for the Sobel case study:

[3]Maserati Quattroporte. URL: `http://upload.wikimedia.org/wikipedia/commons/c/c3/Maserati_Quattroporte_(IAA_2005).jpg`



(a) channel multiplicity (5 threads)

(b) self-loops (5 threads)

Figure 5: Sobel edge detection performance

- Increasing the channel size improves performance. Especially when channels of a single token size are present that can otherwise block very quickly.
- Executing Sobel on all four cores is slower than only using two cores of the CPU for unbounded buffer sizes. We believe this might be due to a bad mapping by the Operating System. After manually mapping all 5 processes manually fixing nodes to cores, execution time decreased from 14 seconds to 7.8s. Performance was still slower than a version only using two cores, but not by much. We believe that the very light computational complexity of Sobel the overhead of keeping every the cache coherency of the cores and core handshaking is too high.
- The pointer reference - release / acquire - method is slower as expected from the throughput graphs since the Sobel algorithm only deals with integers. Why then is actually reading for one and two cores slower? The MJPEG case study will give us some answers.

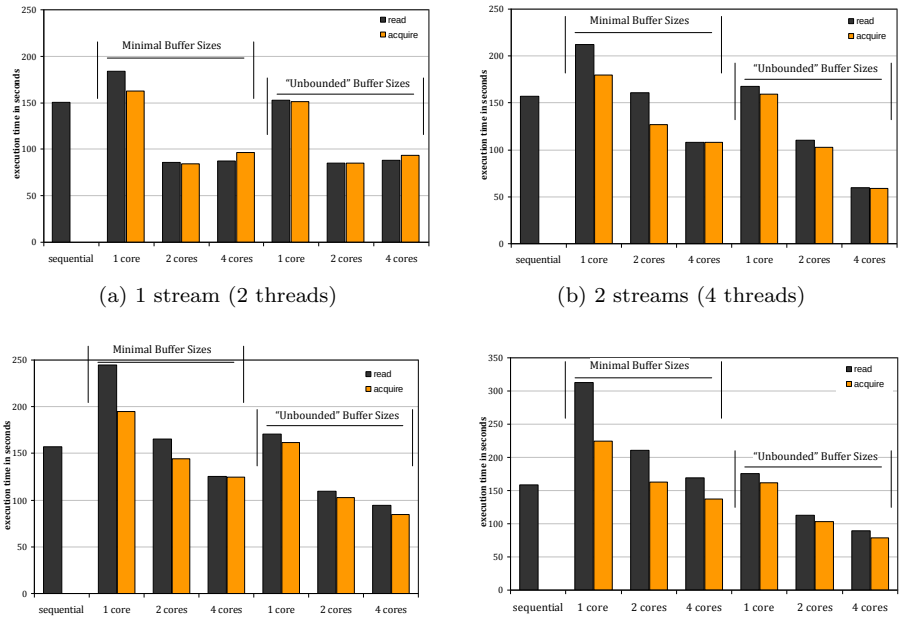
Based on the experiments with the Sobel application we have concluded Sobel should be executed sequentially, there is no performance gain from a parallel implementation. The fine-grained control and minimal token sizes introduce substantial impact and overhead for task-parallelism to be effective.

In [8] 2.2x increase in performance was shown for Sobel. This however was achieved on an FPGA platform with truly distributed memory and computing without any overhead or shared memory buses.

6 Motion JPEG

For the MJPEG encoder we have experimented with several versions of the application. In here we explored process parallelism and data parallelism. We have created four versions of the MJPEG encoder with one, two, four and,

eight streams mapped to two, four, six, and 10 processes respectively. Manually partitioning the source code so that data-parallelism can be expressed as task-parallelism. ESPAM will find these independent processes, and generate the appropriate control flow. MJPEG, especially the 8-stream version (Figure 11b) contained a lot more nodes than there were cores on the system. We have combined several nodes together, but still used ten threads for execution.
The application was executed to encode 32 frames of size 22336×2688[4].



(a) 1 stream (2 threads)

(b) 2 streams (4 threads)

(c) 4 streams (6 threads)

(d) 8 streams (10 threads)

Figure 6: MJPEG encoder performance numbers

Performance numbers are shown in Figure 6a through 6d. It can be seen that not only task parallelism increase performance (single stream only), but a combination of both as well. Best performance was achieved for two streams on four cores.

- Larger channel sizes continue to have a benevolent impact on performance.
- Release / acquire mechanism shows a slightly better performance for MJPEG encoding with a sufficient channel size. Token sizes are bigger, but the impact is only slight as CPU to CPU transfer rates are very high.

[4]Mars Exploration Rover Mission, 2005. URL: `http://apod.nasa.gov/apod/ap051114.html`

- In the 1-stream version of MJPEG there are only two processing nodes, so a maximum of two threads. However if we allow the OS to use all four cores and not restrict to two, we can observe higher execution times. This is in line with our Sobel findings. The OS needs help in mapping the cores.
- For some executions of the 1-stream version of MJPEG (MJPEG1; Figure 6a) the same "strange" issue happens as for the Sobel edge detection algorithm. Pointer reference channel access is slower on four cores on the minimal deadlock-free channel sizes. As MJPEG1 only has two threads and channels of size one (Figure 11 on page 25), blocking will happen very often. The many context switches as a result and the OS executing nodes on any core it wishes, even moving them from core to core at certain times adds much additional overhead on accessing global memory instead of just using the local copy which we believe is the reason for this strange behaviour [14].

Exploiting data parallelism as well as task-parallelism should be used as much as possible. Our expectation is that a desktop computer with more cores will achieve higher performance with each new additional stream. We must however not overload the processor with too many threads as this impacts performance. 10 threads on four cores is slower than 6 threads with less data-parallelism. For a shared-memory system the best mapping is when the number of cores is equal to the number of threads thereby reducing context switches.

7 Wavelet Correlation Matrices of fMRI Data

In cooperation with the Leids Universitair Medisch Centrum[5] we have used the *pn* tool, ESPAM and the presented HDPC framework as well as manual exploration to assist in their research, and in particular to accelerate the algorithms used. We have researched the possibility of execution on an FPGA device as well as our desktop machine through HDPC. Finally, we have explored the possibility of mapping the application on a GPU.

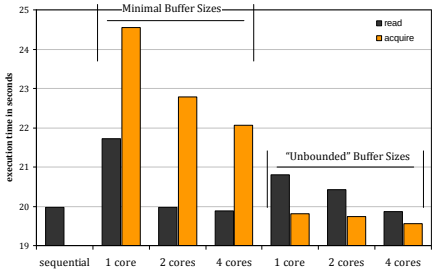
An exhaustive description, design choices and background information is presented in Appendix B on page 26 which is dedicated to functional magnetic resonance imaging (fMRI).

In this section, we present results of two different implementations of the application. The sequential source code as described in Algorithms 2 on page 27 and 3 on page 28 respectively. KPN graphs are also found in the Appendix. We will not give numbers of the FPGA platform here as it was completely unsuitable for this problem area taking hours to complete.

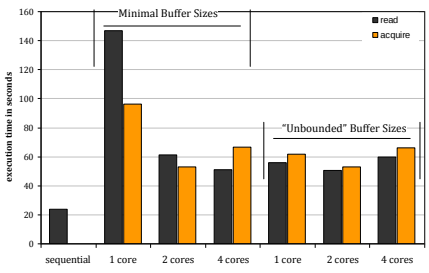
[5]Division of Image Processing (LKEB) Neuro-Image Processing Section

7.1 The Results

Performance numbers set against the sequential version are shown in Figure 7 depicting performance on the HDPC framework. The optimised version correlated a total of $n = 8000$ regions, the original version only $n = 1000$. Scaling up for comparison purposes would take the original version about 65 times longer, taking 26 minutes versus ~ 22 seconds, each new region exponentially increasing computational complexity.



(a) application as provided $n = 1000$ Note: time-scale starts at 19s!



(b) sequentially optimised $n = 8000$

Figure 7: fMRI performance

What do these numbers show us? Firstly, profiling data highlighted that the application has one very heavy processing node. This node accounts for about 97% percent of runtime. We cannot improve any further on this through pure task-level parallelism. The optimised version has the same issue, but now the critical path has shifted to another node. Where first this was the wavelet transformation, now it is the correlation function.

Still, especially in the unoptimised version our framework performs very closely, eventually even achieving a slight performance increase, 19.5 seconds versus 20 seconds. This means that truly our framework has very low overhead. The optimised version of fMRI introduced many self-loops (Figure 12b on page 29) all adding additional overhead for the critical node which will severely impact performance.

Next, we have performed a lot of manual optimisation, porting the application to the GPU. There is a lot of data-parallelism available for exploitation. Figure 8 shows the performance on the GPU versus the best we could get from any other method. 15455 regions were processed in less than 13 seconds. Note that in this experiment we have used a low-end

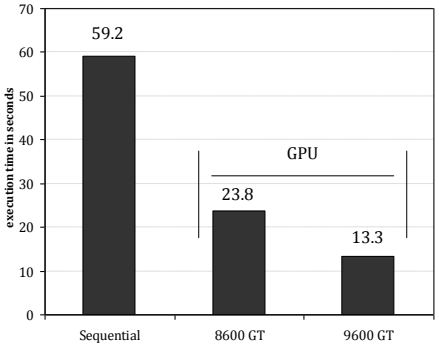


Figure 8: GPU performance for fMRI $n = 15455$

GPU. If a high-end graphics card is considered such as an NVIDIA GTX280, than the expected improvement is not 4.4x but more than 20x.

7.2 Summary

fMRI analysis is not really suitable for task-level parallelisation. Its non-streaming character and single-process heavy critical path will mean only meager performance improvements. At best, parts of the correlation calculation and writing to disk can be done without any overhead.

Since in this particular case we were interested in getting the best performance available, we have hand-tuned the application and executed its data-parallel parts on the GPU.

Part IV

Conclusion

Data-parallelism is relatively easy, both in identifying appropriate parts as well as implementing with many tools already available. There is no overhead, nor concurrency problems present. Task-parallelism however, is another matter entirely. Our tool-chain presented in this paper will find, identify, and correctly implement a concurrent version on a desktop computer.

Buffer sizes are very important. While the *pn* tool guarantees deadlock-free FIFO channel sizes, it does not take into account the number of context switches due to blocking on read or write. The smaller these buffers are, the more context switches happen. The more context switches, the more overhead.

The pointer-reference method of accessing channels has its advantages in pushing performance even further, but care must be taken to only use this method for tokens of a big enough size. Possible blocking situations only increase this threshold.

Mapping of processes by the OS or manually has a small performance impact. However, it is much more important to choose a static, compile-time mapping, that uses about as much threads as there are cores in the system to get the maximum performance (see MJPEG example). When exceeding this soft-limit or allowing automatic mapping of threads to cores by the OS, performance can even degrade.

Whether performance gains can be achieved through task-parallelism is highly dependent on the application at hand. Figure 9 shows a summary comparison of our case studies. The Sobel algorithm was known up front to be a bad candidate for task-parallelism on a shared memory system. We are delighted with the results of the fMRI case study, which shows that given circumstances the maximum theoretical performance gain can be achieved despite the overhead of our framework. For MJPEG the speedup of 2.6x is very good considering that maximum performance of a quad-core machine over a single-core is on average 3x [15, 16].

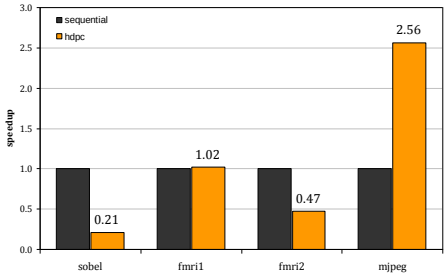


Figure 9: Case-studies summary speedup

We believe that with the future improvements of our tools as discussed, especially in the area of self-loops, the overhead of explicit communication channels can be brought down to a minimum achieving near-sequential performances in the worst case. Once solved, there is no reason anymore of *NOT* using HDPC to exploit any possible task-parallelism.

"Unfortunately", it still holds true that manual optimisation, intricate knowledge of the target platform, and lots of dedicated time achieve the best results... automation can only go so far.

Acknowledgments

I would like to thank Hristo Nikolov at LERC who was always ready to answer my questions and actively participating throughout my research with suggestions towards a better framework. I also want to thank Luca Ferrarini and Julien Milles for their wonderful introduction to the medical imaging world.

References

[1] Sutter H. The free lunch is over. URL: *http://www.gotw.ca/publications/concurrency-ddj.htm*, 2005.

[2] RapidMind. URL: `http://www.rapidmind.net/`.

[3] AccelerEyes. URL: `http://www.accelereyes.com/`.

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

[5] Kienhuis B., Rijpkema E., and Deprettere E. Compaan: Deriving process networks from matlab for embedded signal processing architectures. *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, 2000. May 3-5 2000, San Diego, CA, USA.

[6] Zissulescu C., Stefanov T., Kienhuis B., and Deprettere E. Laura: Leiden architecture research and exploration tool. *International Conference on FPL*, 2003. Sept 1-3 2003, Lisbon, Portugal.

[7] Verdoolaege S., Nikolov H., and Stefanov T. pn: A tool for improved derivation of process networks. *EURASIP journal on Embedded Systems*, 2007(75947), 2007.

[8] Nikolov H., Stefanov T., and Deprettere E. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transaction on Computer-aided Design of Integrated Circuits and Systems*, 27(3), 2008.

[9] Parks T. Bounded scheduling of process networks. *Tech. Rep. UCB/ERL*, 95:105, 1995.

[10] Buck J. and Lee E. Scheduling dynamic data flow graphs with bounded memory using the token flow model. *Proc. IEEE Conf. Acoust., Speech, Signal Process*, pages 429–432, 1993.

[11] Wrinn M. Is the free lunch really over? scalability in manycore systems part 2: Using locks efficiently. *Intel,* URL: *http://software.intel.com/file/7354*, 2008.

[12] Daedalus. URL: `http://daedalus.liacs.nl/`.

[13] de Kock E. A., Essink G., Smits W. J. M., van der Wolf P., Brunel J.-Y., and Kruijtzer W. M. Yapi: Application modeling for signal processing systems. *In ARTICLE of the 37th design automation conference (DAC'00)*, pages 402–405, June 2000.

[14] Multicore is Bad News For Supercomputers. URL: `http://www.spectrum.ieee.org/nov08/6912`.

[15] Intel Measuring Application Performance Figure 3. URL: http://software.intel.com/en-us/articles/measuring-application-performance-on-multi-core-hardware.

[16] Multicore Parallel Computing with OpenMP. URL: http://www.nsu.edu.sg/comcen/svu/publications/hpc_nus/may_2007/NAS-openmp.pdf.

[17] Compute Unified Device Architecture. URL: http://www.nvidia.com/object/cuda_home.html.

[18] NVIDIA Cuda Programming Guide 2.0. URL: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf.

[19] Achard S., Salvador R., Whitcher B., Suckling J., and Bullmore E. A resilient, low-frequency, small-world human brain functional network with highly connected association cortical hubs. *Journal of Neuroscience*, 26:63–72, 2006.

[20] Sporns O., Chialvo DR., Kaiser M., and Hilgetag CC. Organization, development and function of complex brain networks. *Trends Cognitive Science*, 8:418–425, 2004.

[21] Stationary Wavelet Transform. URL: http://en.wikipedia.org/wiki/Stationary_wavelet_transform, and URL: http://www.mathworks.com/access/helpdesk/help/toolbox/wavelet/ch06_ad6.html.

[22] Harris M. Optimizing parallel reduction in cuda. URL: *http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/reduction/doc/reduction.pdf*, 2007.

[23] Xilinx Inc. Xilinx platform studio and the embedded development kit. URL: *http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm*.

[24] Eker J. and Janneck JW. An introduction to the caltrop actor language. 2001. Berkeley, CA 94720-1770.

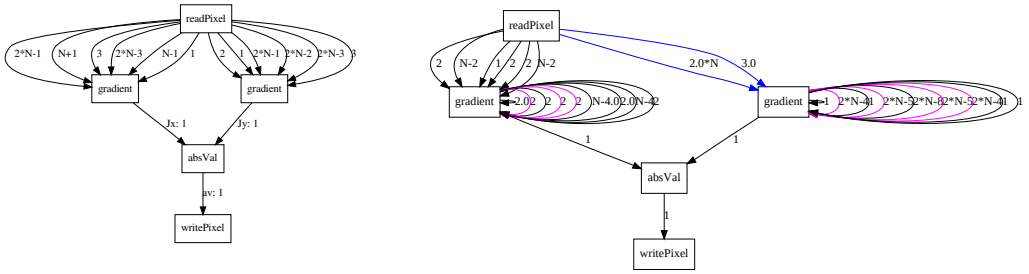
Part V

Appendixes

A Technical Data of Case Studies

A.1 Sobel

Figure 10a shows the KPN graph of the initial version of the Sobel edge detection algorithm, and Figure 10b after applying the data reuse approach in the *pn* tool - introducing self-loops. Each node has been mapped to a separate thread.



(a) channel multiplicity

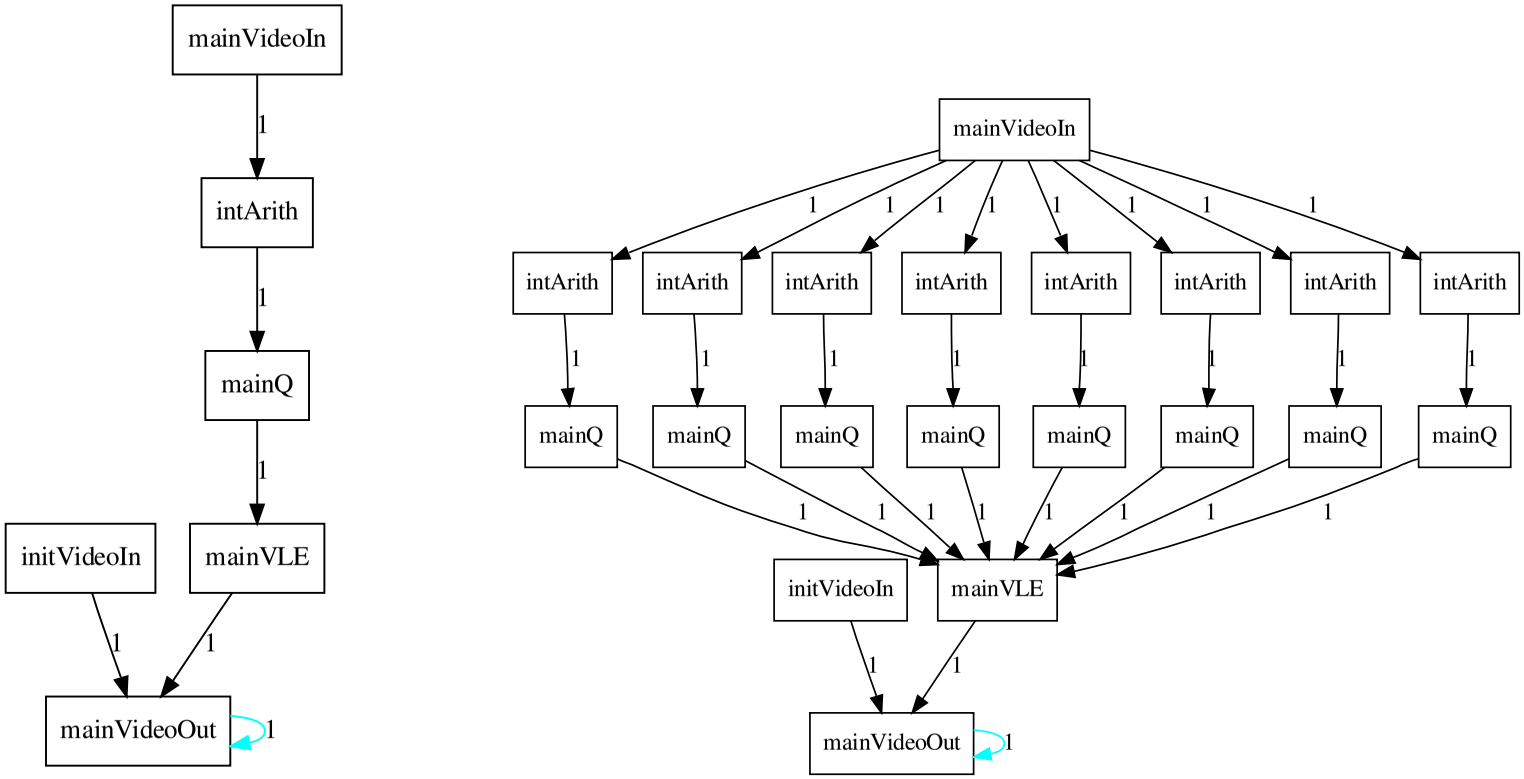
(b) self-loops

Figure 10: Kahn Process Network graphs of the Sobel edge detection algorithm

A.2 MJPEG

Figures 11a and 11b show the KPN graphs of the Motion JPEG encoder. We show the single stream and eight stream version. It is clearly visible that the $initArith \Longrightarrow mainQ$ processes can be executed independently block by block before passing all elements to the variable length encoder.

The eight stream MJPEG encoder groups every $initArith \Longrightarrow mainQ$ block separately together with the input and output, producing a total of ten threads. MJPEG with a single stream only has two threads combining $mainVideoIn \Longrightarrow initArith \Longrightarrow mainQ$ and $initVideoIn \Longrightarrow mainVLE \Longrightarrow mainVideoOut$.



(a) MJPEG using a single stream

(b) MJPEG using a 8 streams

Figure 11: Kahn Process Network graphs of the Motion JPEG encoder

B Extended Technical data of the fMRI case study

In the this section we discuss briefly the research background done at the Leids Universitair Medisch Centrum (LUMC), their problem and the different approaches we have taken. In Appendix D on page 38, we motivate the correctness of the obtained results.

B.1 The Small-World Human Brain Network

Research by Achard et al.[19] showed the human functional network to exhibit "small world" properties. Small-world networks are a type of graph where most nodes are not neighbours of each another, but most nodes can be reached from every other by a small number of steps. These networks contain a substantial core of highly connected hubs and long-distance connections to other regions. Such properties are attractive models for connectivity of nervous systems because they allow for both specialised or modular processing in local neighbourhoods and distributed or integrated processing over the entire network [20].

Achard et al. based their research on fMRI time series images of human volunteers from 45 anatomical regions of interest of the cerebral hemisphere. Wavelet transformation - more specifically Maximal Overlap Discrete Wavelet Transform (MODWT) - was then applied to the imaging data before estimating the pair-wise inter-regional correlation of the wavelet coefficients.

Dividing the cerebrum into 90 regions - 45 anatomical regions for both halves of the cerebral hemisphere - only provides us with a coarse overview of connectivity. LUMC research was to subdivide the anatomical regions to get a more detailed picture, thereby increasing the number of voxels to 15445. The signal was followed over 200 time-units, the so-called time-dimension.

B.2 Computational Background

The original research application at LUMC was written in Matlab®. We have reimplemented all code including Matlab-internal functions into C++. Algorithm 2 shows a simplified C/C++ version of the application to generate the inter-correlation maps. After data acquisition, a stationary wavelet transformation `swt()` is applied and the correlation coefficient `corrcoef(a, b)` is calculated for each voxel.

The Stationary Wavelet Transform (SWT) [21] is similar to the discrete wavelet transform except for the fact that the signal is never subsampled, instead the filters are upsampled at each level of decomposition. The correlation coefficient $\rho_{x,y}$ between two random vectors X and Y of length n is:

Algorithm 2 Generating inter-correlation maps of N voxels

```
typedef struct doubled {
  double data[TIME];
}

int main() {
  doubled data[N];
  for (int i = 0; i < N; i++) data[i] = readData(i);

  for (int i = 0; i < N; i++) {
    doubled voxel1 = swt(data[i]);
    for (int j = i + 1; j < N; j++) {
      doubled voxel2 = swt(data[j]);
      double r = corrcoef(voxel1, voxel2);
      writeData(r);
    }
  }
  return 0;
}
```

$$\rho_{x,y} = \frac{cov\,(X,Y)}{\sqrt{cov\,(X,X)\,cov\,(Y,Y)}} \qquad where \qquad cov = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{x})\,(Y_i - \bar{y}) \tag{1}$$

It should be noted that Algorithm 2 is already a slightly optimised version of the original code. The definition of $\rho_{x,y}$ above shows that $\rho_{x,x} = 1$. More importantly $\rho_{x,y} \equiv \rho_{y,x}$. Therefore calculating the full correlation matrix is highly redundant. Coefficients are mirrored across the main diagonal which only has values of one. Equation 2 shows this for a vector of length n. α is the correlation coefficient of voxels one and two.

$$\left\langle \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \right\rangle \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 1 & \alpha & \beta & \gamma & \delta \\ \alpha & 1 & \epsilon & \varepsilon & \zeta \\ \beta & \epsilon & 1 & \eta & \theta \\ \gamma & \varepsilon & \eta & 1 & \vartheta \\ \delta & \zeta & \theta & \vartheta & 1 \end{bmatrix} \end{array} \qquad \begin{bmatrix} 1 & \alpha & \beta & \gamma & \delta \\ & 1 & \epsilon & \varepsilon & \zeta \\ & & 1 & \eta & \theta \\ & & & 1 & \vartheta \\ & & & & 1 \end{bmatrix} \tag{2}$$

Therefore, the input vector of n voxels resulting in an $n \times n$ correlation coeffient matrix needs $\frac{(n\times(n-1))}{2}$ iterations. We call this matrix the upper unitriangular matrix.

We can see that the resulting matrix grows exponentially. While 90 regions only result in 4005 possible pairs between voxels, 15455 regions result in $\approx \frac{1}{2}15455^2$

pairs - over 100 million. Not only computation time explodes exponentially but also the resulting data set.

B.3 Platform Results

We have executed this application on a number of different platforms and programming approaches. The goal was twofold:

1. Beat the sequential program as shown in Algorithm 2 in terms of performance.
   This is the *original benchmark* where an application developer uses our toolset without any changes to the source code.
2. Beat an optimised version of the algorithm that an application developer could perform himself with a some time spent during development. We will call this the *optimised benchmark*.

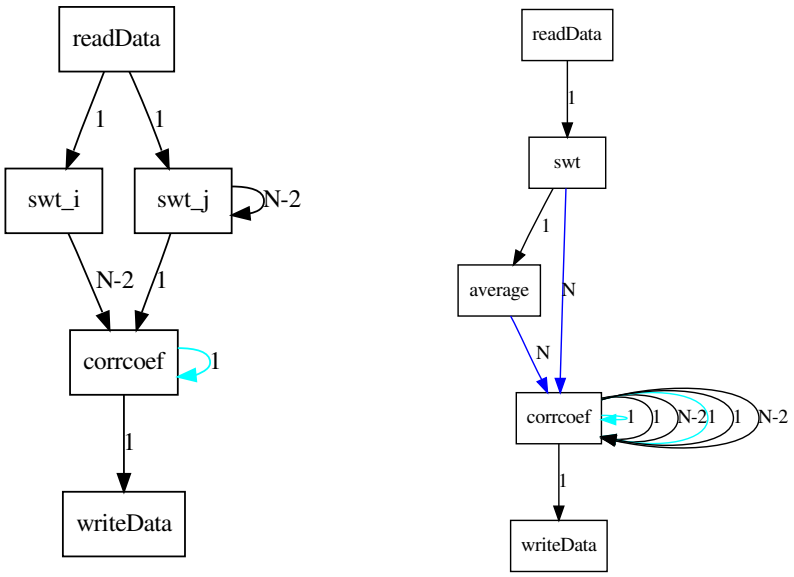
Algorithm 3 Optimised generation of intercorrelation maps of N voxels

```
typedef struct doubled {
  double data[TIME];
}

int main() {
  doubled data[N], swt_data[N];
  double avg[N];
  for (int i = 0; i < N; i++) {
    data[i] = readData(i);
    swt_data[i] = swt(data[i]);
    avg[i] = average(swt_data[i]);
  }

  for (int i = 0; i < N; i++) {
    for (int j = i + 1; j < N; j++) {
      double r = corrcoef(avg[i], avg[j], swt_data[i], swt_data[j]);
      writeData(r);
    }
  }
  return 0;
}
```

The optimised benchmark's algorithm is shown in Algorithm 3. We can see that the wavelet transform has been taken out of the loop as there is no feedback needed to recalculate the transformation every iteration. A pre-calculated average of a voxel is now passed to the correlation function because that is also a constant during execution.

B.3.1 KPN Mapping

Figure 12 shows the generated Kahn Process Networks of the fMRI scan analysis application for a given number of n regions. Figure 12a shows the unoptimised version, while Figure 12b is graph of Algorithm 3.



(a) original (b) optimised

Figure 12: Kahn Process Network graphs of fMRI analysis

Previous profiling data showed that the wavelet transformation function is the heaviest, about 60x slower per execution than the correlation function and taking 97% of runtime for Figure 12a. In any parallel program, the best possible performance cannot improve on that of the slowest component. In Figure 12a generated from Algorithm 2, the critical path is clearly node `swt_j` which is executed in every iteration. Any performance increase that will be achieved through pure task-parallelism will not exceed 3%.

B.3.2 FPGA Implementation

Mapping our application onto an FPGA board first meant working around some limitations of the platform, a *Virtex II* PCI card. This Xilinx board only had 6MB of internal memory.

For streaming applications this memory limit does not have to be a problem. However, in analysing data from the fMRI there can be no real talk of a streaming application.

Equation 3 below shows a simple matrix of five different regions and the corresponding correlation. Vectors of length two show which regions are correlated at that position.

$$\begin{bmatrix} . & \langle 1,2\rangle & \langle 1,3\rangle & \langle 1,4\rangle & \langle 1,5\rangle \\ . & . & \langle 2,3\rangle & \langle 2,4\rangle & \langle 2,5\rangle \\ . & . & . & \langle 3,4\rangle & \langle 3,5\rangle \\ . & . & . & . & \langle 4,5\rangle \\ . & . & . & . & . \end{bmatrix} \tag{3}$$

Assuming that we process elements row by row, we can see that for example region five is not only needed in the very beginning but also at the very end of the process.

Due to memory constrains on the FPGA device not all regions will be present in memory and thus need to be refetched from the host; possibly even multiple times. As the FPGA device is a PCI add-on card all data has to move through the PCI bus. Its throughput is much less than that of on-chip CPU to CPU transfer, therefore it is very important to minimise transfer overhead.

For any given number of regions the following mathematical formula describes the total memory requirements:

$$\tau\lambda x + \frac{1}{2}\lambda x\,(x-1) = \phi \qquad \begin{array}{ll} \tau & experiment\,time\,frames \\ \lambda & element\,size\,(4\,float \vee 8\,double) \\ x & elements\,to\,process\,simultaneously \\ \phi & total\,required/available\,memory \end{array} \tag{4}$$

$\tau\lambda x$ is data in and $\frac{1}{2}\lambda x\,(x-1)$ is data out. When correlating $x = 5$ regions, we will need to allocate $5\tau\lambda$ memory for input and 10λ memory units for output on the device.

Taking actual values from the case study we have values: $\lambda = 8$, $\phi = 6MB$ and $\tau = 200$. Solving this equation results in the ability to process $x = 1070$ regions simultaneously. In the case of a streaming application splitting the actual 15455 regions into fourteen blocks would be enough. But we can not do this.

Our solution was to transform Equation 4 into the pure polynomial form $2\tau\lambda x + \lambda x^2 = \phi$. We can solve this second degree polynomial equation easily and

implement the following calculation metric:

$$\prod_{i=1}^{i=n} \begin{cases} i = odd, & \prod_{j=i}^{j=n-1} x_i \times y_{j+1} \\ i = even, & \prod_{j=n-1}^{j=i} x_i \times y_{j+1} \end{cases} \qquad i, j \quad elements\, of\, \tau\, time frames\, each$$

So now the calculation order will be $1,2 \Rightarrow 1,3 \Rightarrow 1,4 \Rightarrow 1,5 \Rightarrow 2,5 \Rightarrow 2,4 \Rightarrow 2,3 \Rightarrow 3,4 \Rightarrow 3,5 \Rightarrow 4,5$ using the example from Equation 3. The only difference is now that the numbers now are no longer single regions, but blocks of $x = 1070$.

The reversion is done at the end of each row because otherwise two new elements need to be transferred through the PCI bus instead of one; e.g. transferring blocks $\langle 2,3 \rangle$ where memory contains $\langle 1,5 \rangle$ instead of only transferring block 2.

We will not show actual performance numbers here because the platform was very slow. Calculating a single correlation took on average 17200 cycles on the FPGA device. Running at 66MHz, the whole application will be busy for at least 7 hours. In theory, and after lots of manual tuning, performance could be improved six times by using six independent softcores (MicroBlazes) connected to the six memory banks on the device. Even then, parallelism will need to provide the other 78x speedup to even match the simple C++ version. Given these numbers, further research into this platform for fMRI analysis was abandoned.

Trying to implement the KPN network created above by ESPAM was not even attempted. Buffer sizes need to be a full N region sizes. Given only 288KB of on-chip memory, we would need to allocate memory from main memory, further reducing block sizes and increasing runtime.

B.3.3 Heterogeneous Desktop Parallel Computing

We manually copied the control flow into our framework. For results, we refer to Section 7 and Figure 7.

The wrapper code of our HDPC framework is in Algorithm 4. It shows the input and output disk processes, the five computing nodes and their interconnections. Self-loops are created on line 20. We also limit execution of node 1 (`ND_1`) to core 3 (0x4h is 100b which sets the third bit).

B.3.4 GPU Implementation

There is no support for data-parallelism in the ESPAM toolchain and that is neither the focus of our research nor our framework, but the cooperation with the medical faculty required the best available performance, so we have also implemented a GPU version.

Calculating correlation coefficients of many regions is a good test-case for data-parallelism, and therefore the GPU. We have used the Compute Unified Device

Algorithm 4 The fMRI application implemented in the Hdpc framework.

```
1  #include "hdpc/process.h"
2  #include "hdpc/platform/disk2.h"
3  #include "hdpc/platform/cpu.h"
4
5  #define N 15455
6  int main() {
7    StorageLineT<doubled> f1("input.bin", Storage::STREAM_IN);
8    StorageLineT<double>  f2("output.bin", Storage::STREAM_OUT);
9    CPU c1, c2, c3;
10
11   Process p_ND_0(0, 2);
12   Process p_ND_1(1, 1);
13   Process p_ND_2(2, 2);
14   Process p_ND_3(2, 1);
15   Process p_ND_4(1, 0);
16
17   p_ND_0.attachoutput<doubled>(0, p_ND_1.getInPort(0), 1);
18   p_ND_0.attachoutput<doubled>(1, p_ND_2.getInPort(0), 1);
19   p_ND_1.attachoutput<doubled>(0, p_ND_3.getInPort(0), N - 2);
20   p_ND_2.attachoutput<doubled>(0, p_ND_2.getInPort(1), N - 2);
21   p_ND_2.attachoutput<doubled>(1, p_ND_3.getInPort(1), 1);
22   p_ND_3.attachoutput<double>( 0, p_ND_4.getInPort(0), 1);
23
24   HANDLE h[5];
25   h[0] = p_ND_0.start(0x1, f1, fMRI_1ND_0);
26   h[1] = p_ND_1.start(0x4, c1, fMRI_1ND_1);
27   h[2] = p_ND_2.start(0x0, c2, fMRI_1ND_2);
28   h[3] = p_ND_3.start(0x0, c3, fMRI_1ND_3);
29   h[4] = p_ND_4.start(0x1, f2, fMRI_1ND_4);
30
31   WaitForMultipleObjects(lengthof(h), h, true, INFINITE);
32   return 0;
33 }
```

Architecture (CUDA) framework on NVIDIA based hardware for the implementation. Hardware model, and programming on a GPU are explained in Appendix C.1 on page 34.

In programming CUDA or any other data-parallel device, it is essential to limit the control flow which severely impacts performance. Therefore, we have padded the input data to the next power of two and used a heavily optimised parallel reduction [22]. A thread was created for every correlation point with only a very lightweight control command skipping the lower-triangular part of computation. Algorithm 3 on page 28 was used as a starting point. This saved us from implementing a difficult wavelet transform program with lots of control on the GPU and let us focusing only on the correlation.

While there is no such harsh memory limit as on the FPGA device requiring us to split calculation into multiple batches, there were performance issues. Again, we refer to Equation 2 on page 27. Doing larger blocks of regions at the same time means creating more "waste". As the lower triangle of the matrix is not computed, these threads do not execute anything and return after the initial check. We can see in Figure 13 on the next page how this affects performance.

With larger blocks there will be more waste, and this shows up in a higher execution time per unit.

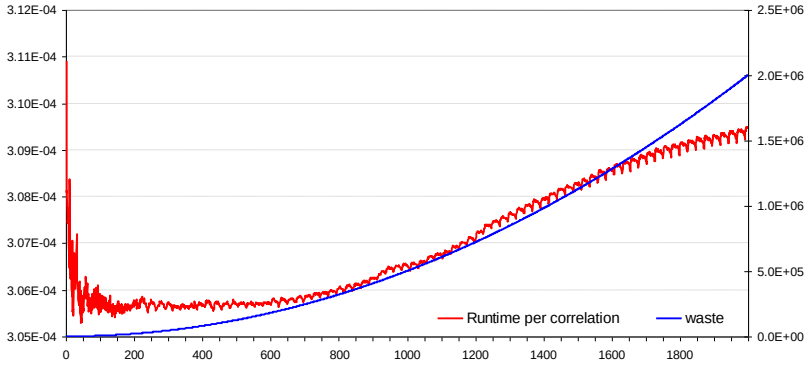


Figure 13: Average runtime per correlation on the GPU

Using this knowledge, we were able to achieve substantial speedups versus the optimised sequential version. Figure 14 shows this. C++ is the algorithm implemented on the host machine, 8600GT and 9600GT are the runtimes on the GPU.

The difference between these two cards are the number of streaming processors available, four versus eight. We can see an almost linear increase in performance for the 9600GT. Unfortunately, we had no access to more expensive professional cards - this one costs less than EUR 100 - but we can assume this linear increase holds for current top-of-the-line cards as well having 30 or more steam processors. In that case, performance can increase from 4.6x up to 20x or more.

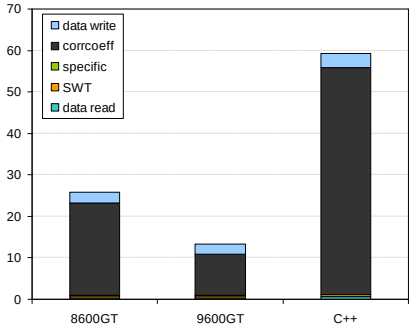


Figure 14: CUDA performance in seconds

We must stress however, that we have written manually optimised code, whose performance might be hard to match by tools. Also, our toolchain does not implement data-parallelism, only task-parallelism. However, we can write the GPU kernel ourselves and plug it into the corrcoef processing node of a HDPC implementation achieving comparable performance.

C Hardware Computing Devices

C.1 Graphics Processing Unit

C.1.1 General-Purpose Computation on GPUs

The Graphics Processing Unit (GPU) is a dedicated graphics rendering device embedded into a graphical card. These microprocessors handle the compute intensive manipulation of computer graphics traditionally performed by the CPU. The highly-parallel and compute-intensive nature of graphics rendering allow GPU chips to devote much more transistors to data processing rather than data caching and flow control as in a CPU.

Therefore, GPUs are well-suited to address problems which can be expressed as data-parallel computations - the same operation is executed on many data elements in parallel - with high arithmetic intensity - the number of computations performed on a single data element. The more computations are performed on the same data element, the higher this ratio becomes and the better performance will be achieved. Because the same algorithm is executed for each data element, there is less requirement for sophisticated flow control; and because of high arithmetic intensity, the memory access latency can be hidden with calculations instead of big caches.

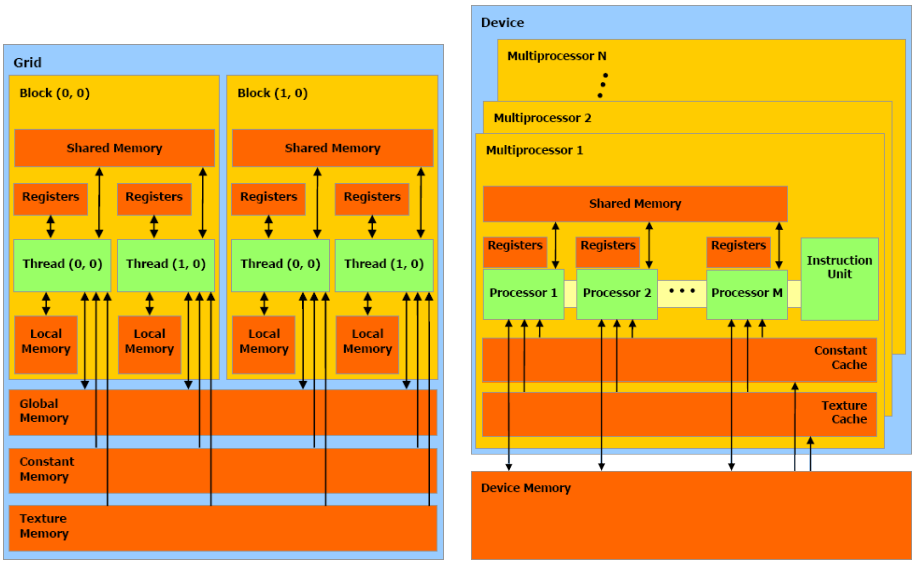
C.1.2 NVIDIA and CUDA Platform

NVIDIA, one of the biggest graphics manufacturers of today, has pushed General-Purpose computation on GPUs (GPGPU) to a new level starting with their Geforce8 series of products. When programmed through CUDA, algorithms that exhibit data-parallel, compute-intensive properties can be offloaded to the graphics device which acts as a co-processor to the main CPU. The device will then execute this part - called the *kernel* - as many different *threads* with each thread operating on a single data element.

Both the host and the device maintain their own memory, and data transfers between the two is achieved through calls that utilize the device's high-performance Direct Memory Access (DMA) engines.

C.1.3 Device Architecture

The CUDA architecture adds support for general Device Random Access Memory (DRAM) addressing (gather and scatter - read and write to any memory location, just like on the CPU). On-chip shared memory with very fast access times which threads can use to share data with each other is available as well. Applications



(a) memory model (b) hardware model

Figure 15: CUDA architecture

can use this cache to minimize round-trips to DRAM and become less dependent on DRAM memory bandwidth.

The batch of *threads* that is executed on a given kernel on the device is organised into a *block* of threads that can cooperate together by efficiently sharing data through this fast shared memory and synchronize their execution to coordinate memory accesses. Blocks of the same size can be batched together into a *grid* of blocks as shown in Figure 15a.

Organising blocks and grids in this way allows kernels to efficiently, and without recompilation scale up with newer or more powerful devices. A low-end device with only a few multiprocessors may run all the blocks of a grid sequentially, whereas a high-end device with a lot of parallel capabilities (lots of multiprocessors) in parallel - usually a combination of both.

A device is implemented as a set of multiprocessors with a Single Instruction Multiple Data (SIMD) architecture, see Figure 15b. Each multiprocessor consists of 8 generalised processors that always execute the same operation in a SIMD fashion on different data elements.

The NVIDIA CUDA Programming Guide [17, 18] on the NVIDIA website explains the CUDA architecture in much more detail.

It is worth noting however that NVIDIA is not the only graphics chip manufacturer that has a platform for GPGPU. AMD - which has recently acquired ATi - offers a similar solution through their *ATI Stream* technology[6].

C.1.4 The Compiler

NVIDIA supplies their own C compiler called *nvcc* to generate device code. Developing for this platform will entail a control part that runs on a general purpose computer - Windows, Linux - using one or more NVIDIA GPUs as coprocessors to execute SIMD parallel jobs. *nvcc* is smart enough to pass the *host* part of the compilation trajectory to the system-installed compiler - gcc or msvc for example - thereby allowing the programmer to exploit all compiler features and C++ intricacies in host mode.

For *device* mode several steps of splitting, compilation, preprocessing and merging are performed by nvcc to finally produce a binary code image embedded in the executable containing the job to execute on the device.

Most interesting feature of nvcc is actually being able to generate an intermediate assembler code format called Parallel Thread Execution (ptx) that can be compiled runtime for the proper architecture. In this way, the developer can create an executable using a "compile once run everywhere" approach much akin to the Java bytecode and Java Virtual Machine paradigm. However, the developer might also choose to pre-compile finetuned versions of ptx for particular architectures and allow the CUDA runtime system to choose the proper image.

C.2 Field-Programmable Gate Array

Traditionally, digital design was a manual process of designing circuits using schematic tools. This bottom-up method is both time-consuming and error prone. Application-Specific Integrated Circuits (ASICs) enable engineers to use a top-down approach by using hardware-description languages.

However, once these Integrated Circuits (ICs) have been programmed they are hardwired and changing the product, fixing bugs, or even prototype development is an expensive process. The FPGA presents a solution to this problem. A completely finished device whose programmable logic blocks and reconfigurable interconnects allow rapid prototyping and hardware/software integration testing.

Using tools such as the XILINX Platform Studio [23] even allows developers to specify their application in the high-level C language, targeting pre-fabricated

[6] `http://ati.amd.com/technology/streamcomputing/`

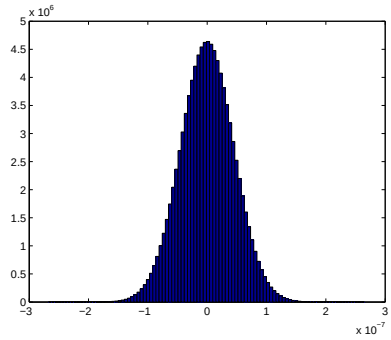
softcores such as the $\mu Blaze$. This flexibility allows developers to fully exploit the hardware capabilities.

At Liacs we use this platform for developing streaming parallel programs and achieve performance that will even beat desktop PC computers in certain areas.

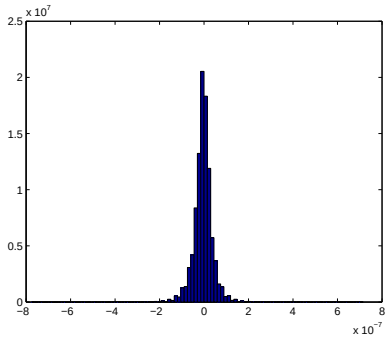
D Correctness of fMRI Implementation

Since the original research application at LUMC was written in Matlab® we have reimplemented both SWT and the correlation calculations in C++. With a sufficiently low number of voxels we were able to verify our results against the "reference" Matlab implementation.

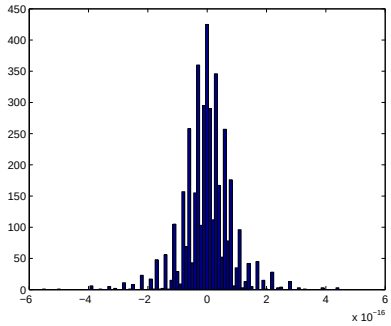
Testing on $n = 90$ regions, double-precision calculation has a maximum error rate of $1.3E^{-15}$, single-precision is $1.8E^{-5}$. These results were obtained by converting double-precision input to single-precision, doing the calculations in single-precision, converting results back to double-precision and comparing the results with the original double-precision results.



(a) C++ (double) and CUDA $n = 14000$



(b) C++ (single) and CUDA $n = 14000$



(c) Matlab and C++ (double) $n = 90$

Figure 16: Error rates

Figure 16 plots the error rates. All graphs were created with a 2nd level SWT decomposition and the Daubechies 6 filter. We can see that not only most of the error rates are zero, but are also clustered around that point. Figure 16c shows the difference for $n = 90$ of the reference Matlab implementation and our C++

version run on the host machine in double-precision mode. Figures 16b and 16a show the difference between host machine and the calculation performed on the GPU.

These slight error rates for floating-point computations are expected. In general, all you need to get different results for the same floating-point computation are slightly different compiler options, let alone different compilers, different instruction sets or different architectures.

E Hdpc API reference

The Heterogeneous Desktop Parallel Computing framework consists of three parts. A *Preprocessor* part allows for manipulation of the framework. These include debugging functionality, timers and the communication channel type.

The *Process Network Construction* phase is responsible for creating the processing nodes, identifying them with functionality and computing devices and connecting the whole network. Finally, *Process Functionality* handles the communication with the channels, the control flow and token passing.

Platforms in the platform backend allow execution to be directed to external computing devices or use the integrated and approach to IO. They are discussed in Section E.4. An Unified Modeling Language (UML) diagram of our framework is shown in Figure 17 (the fMRI specific read/write functions are included as the `StorageSWT` class).

E.1 Preprocessor Directives

Preprocessor directives fix certain behaviour during compile time. This way Hdpc retains the flexibility of available debugging facilities for example whilst not compromising performance. The defines below have to be passed to the compiler as a parameter, eg `-DSIGNAL_WAIT` for gcc.

E.1.1 Debugging

NAME

`DEBUG_PROCESS_NETWORK` - enable debug mode

DESCRIPTION

The framework is executed in debug mode. Not only will the network connections be checked for correctness up front but also during runtime type information is kept. When a process has finished all its input channels are checked for remaining data.

Performance will obviously drop in debugging mode but it is advisable to at least once execute the network in this mode to ensure the setup has been correctly done.

E.1.2 Signal / Semaphore Wait

NAME
`SIGNAL_WAIT / SPIN_WAIT` - select channel polling method

DESCRIPTION
Signaling will use windows semaphores for access yielding thread execution until such a semaphore is set. Spinning continuously tests the channel status only yielding its current timeslice and requesting immediate thread rescheduling using a shared (volatile) variable. Spinning usually gives higher performance but will keep the system at full load while just waiting. Signals on the other hand idle until reawoken by the Operating System.

E.1.3 Read / Write and Acquire / Release

NAME
`USE_ACQUIRE_RELEASE_MECHANISM` - use release/acquire semantics

DESCRIPTION
HDPC allows the developer to request pointers to the channel tokens and operate directly on those instead of making a local copy. Using acquire/release semantics is still possible without this preprocessor define, however certain restrictions apply, for example it is not allowed to acquire a pointer from the same channel consecutively in the same iteration without releasing the first. When `DEBUG_PROCESS_NETWORK` is defined a warning is given in this case.

E.1.4 Timing

NAME
`KEEP_TIMING_INFORMATION` - keeps a few timing statistics

DESCRIPTION
Enables some timers that will keep count of process execution and idle (blocked) time at some performance penalty. The timer statistics are automatically printed to the console when `DEBUG_PROCESS_NETWORK` is defined.

E.2 Process Network Construction

The `Process` class is the visible side of the framework. Through functions of this class the network is constructed, connected and behaviour defined.

E.2.1 Process

NAME
`Process` - main HDPC class

SYNOPSIS
```
Process(size_t inPortCount, size_t outPortCount);
```

DESCRIPTION
Constructs an HDPC class with the given number of in and outgoing ports.

E.2.2 Port Access

NAME
`getInPort / getOutPort` - get a pointer to a reference of a port

SYNOPSIS
```
ChannelBase *&getInPort(size_t port);
ChannelBase *&getOutPort(size_t port);
```

DESCRIPTION
Returns the pointer to a reference of the given port. Used in connecting the network in functions `attachinput()`, `attachoutput()`.

E.2.3 Connecting Processes

NAME
`attachinput / attachoutput` - make the connection between processes

SYNOPSIS
```
template <class T> bool attachinput(size_t port_in, ChannelBase
*&q, size_t queueSize);
template <class T> bool attachoutput(size_t port_out, ChannelBase
*&p, size_t queueSize);
```

DESCRIPTION

A process object attaches to the specified input/output port a channel of a certain size. The channel is returned by the `getInPort()` / `getOutPort()` functions of the connected process. The templatised variable will give the token type. The channels will be `queueSize` token-type long. When connecting two nodes it is sufficient to only attach the output of one to the input of the other; both will know of the connection.

E.2.4 Starting Execution

NAME

`start` - start the execution of a process

SYNOPSIS

```
HANDLE start(DWORD_PTR cpu_mask, Platform &a, process proc, bool
allocmem = false);
```

DESCRIPTION

This function associates a processing backend, `Platform`, with some functionality to the process which will execute including all the control flow. The `allocmem` parameter will allocate memory of the proper channel size on the processing backend. Use it if needed. `getDeviceInMem()` and `getDeviceOutMem()` will retrieve the pointers.

`cpu_mask` is usually 0x0, unless one wishes to restrict execution of the process on a given core of the host machine. Then it is a bitmask of enabled cores. For example 0x5 executes on cores 1 and 3 of a quad-core machine.

E.3 Process Functions

Each node has a reference to a `Process` class as input argument to the function. The functionality below can be accessed as member-functions of this class.

E.3.1 Physical Channel Communication

NAME

`readFromPort / WriteToPort` - transfer a token through a channel

SYNOPSIS

```
template <class T> bool readFromPort(size_t port, T &element)
throw(...);
template <class T> bool writeToPort(size_t port, const T
&element);
```

DESCRIPTION

Execute a blocking read or write operation on the given port. The token to be communicated is in the second argument. Template types are automatically derived from the token and are checked for consistency in `DEBUG_PROCESSING_NETWORK` mode. A true return value indicates success.

E.3.2 Acquire Channel Locks

NAME

`getReadPointer / getWritePointer` - acquire the pointer to token

SYNOPSIS

```
template <class T> const T &getReadPointer(size_t port)
throw(...);
template <class T> T &getWritePointer(size_t port);
```

DESCRIPTION

Returns a reference to the first available token in the given channel. Specifying the channel type in the template is mandatory. A wrong type will give an error in debugging mode.

E.3.3 Release Channel Locks

NAME

`releaseReadPointer / releaseWritePointer` - release the pointer to token

SYNOPSIS

```
template <class T> void releaseReadPointer(size_t port);
template <class T> void releaseWritePointer(size_t port);
```

DESCRIPTION

Releases the pointer for the channel, indicating that processing on that token is finished and is ready for other nodes to access. Release must be done in the same order observing the same control flow as acquire, otherwise problems can occur. `releasePorts()` can help in complicated situations.

E.3.4 Release All Channel Locks

NAME

`releasePorts` - release all acquired read and write tokens

SYNOPSIS

```
void releasePorts();
```

DESCRIPTION

Releases all acquired read and write tokens of the current process in the current control iteration. Adding this to the end of the control iteration is sufficient to ensure proper release operation.
Using the normal release functions, which only release one port at a time the whole control flow of acquire has to be copied for release mode. E.g. if only iteration two and three acquire a read lock for port zero, then only iterations two and three can release that lock. By using this function any acquired port in the current iteration is automatically released.

Of course this means locks are possibly released much later if used at the end of the current control iteration, so could come at some performance penalty. However a lot of additional control flow is avoided, so it is up to the designer to find the best solution.

E.3.5 Accessing the Computing Backend Memory

NAME

`getDeviceInMem / getDeviceOutMem` - return pointers of computing device

SYNOPSIS

```
void *getDeviceInMem(size_t port);
void *getDeviceOutMem(size_t port);
```

DESCRIPTION

Get pointers to the address space of the computing device that was allocated previously in the `start()` function. These pointers can be passed to the actual executing function already pointing to valid memory locations. Through this mechanism the framework can automatically transfer memory to and from the device.

E.3.6 Accessing the Computing Backend Platform

NAME

`getProcess` - return a reference to the computing backend

SYNOPSIS

```
Platform &getProcess();
```

DESCRIPTION
Returns a reference to the computing backend through with which its specific read and write operations can be performed.

E.4 Platform functions

The `Platform` class is the computing backend of our framework. This class is responsible for device-specific communication. To define a new computing library the following functions have to be declared and implemented.

NAME
`init / deinit` - (de)initialise the backend

SYNOPSIS
```
bool init();
bool deinit();
```

DESCRIPTION
Initialise the device, set up required resources, communication in such a way that subsequent calls to this backend's `read()` / `write()` functions will succeed. Deinitialisation should restore the device to a state as it was before initialisation was performed.

NAME
`write / read` - transfer data to and from the backend

SYNOPSIS
```
bool read(const void *src, void *element, size_t size);
bool write(void *dst, const void *element, size_t size);
```

DESCRIPTION
Writes, respectively reads to/from the memory location at `dst` / `src` the token of a given size. This is usually `sizeof(element)`. The return value indicates success or failure. The target and source memory address can be retrieved by a call to `getDeviceInMem()` and `getDeviceOutMem()`. Functionality of transfer is device-specific.

NAME
`name` - return name

SYNOPSIS

```
char *name();
```

DESCRIPTION

Used for debugging purposes and should be set to a string describing the backend, eg. “NVIDIA 8600GT” for the GPU. Can be left empty “” if no name is desired.

NAME

`allocmem / freemem` - (de)allocate memory on computing backend

SYNOPSIS

```
bool allocmem(size_t size, void *&buf);
bool freemem(void *&buf);
```

DESCRIPTION

Allocate memory of a given size on the device which HDPC can use for the `read()` / `write()` operations. `buf` is a pointer to a reference and will contain a pointer to the memory space of the device.

Deallocation frees any allocated memory on the device previously allocated. The parameter is a pointer returned by `allocmem()`.

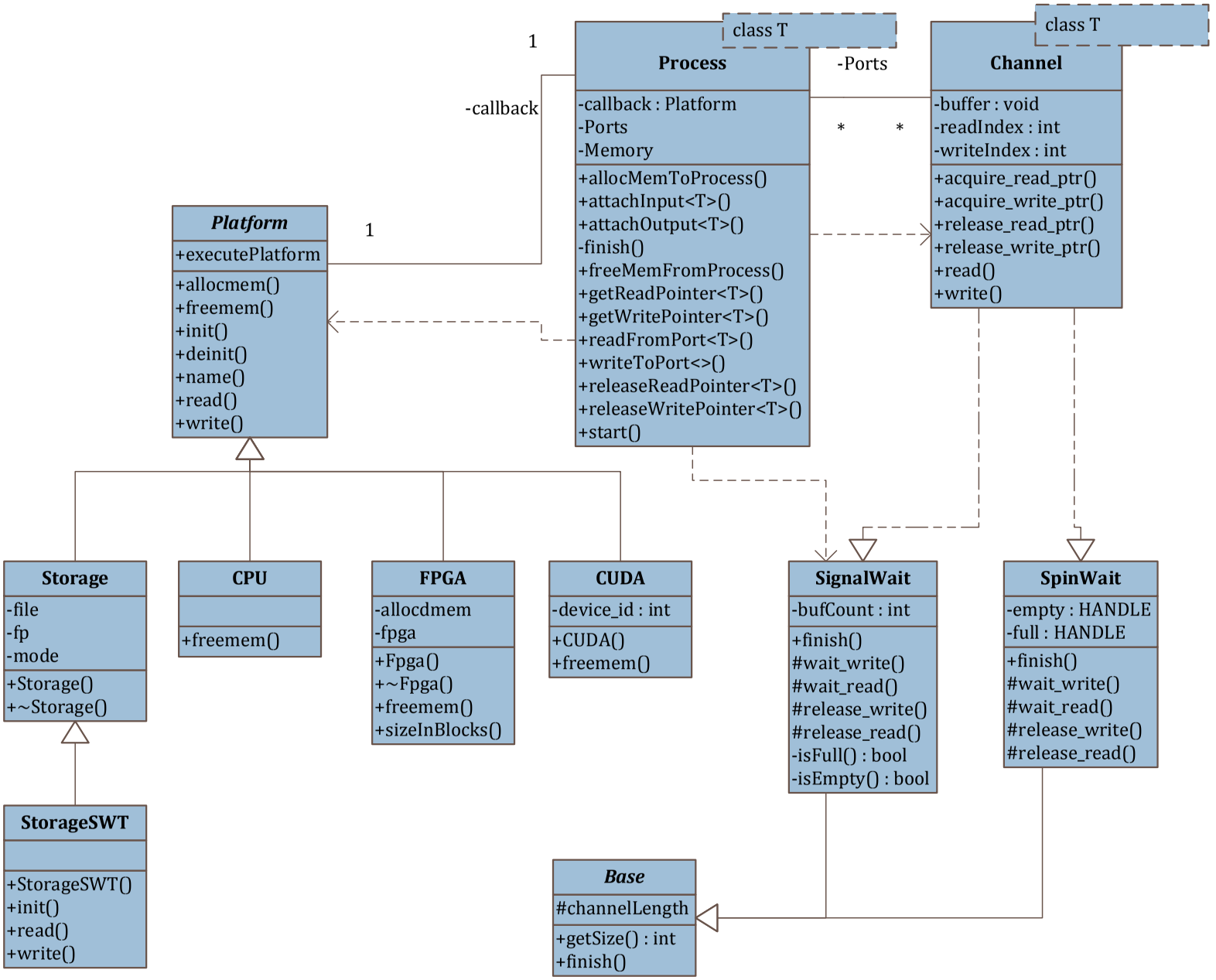


Figure 17: Unified Modeling Language (UML) diagram of HDPC framework

F List of Acronyms

API	Application Programming Interface 5
ASIC	Application-Specific Integrated Circuit 36
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture 31
DMA	Direct Memory Access 34
DRAM	Device Random Access Memory 34
Espam	Embedded System-level Platform Synthesis and Application Mapping 3
FIFO	First-In-First-Out
fMRI	functional magnetic resonance imaging 18
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit 3
GPGPU	General-Purpose computation on GPUs 34
Hdpc	Heterogeneous Desktop Parallel Computing 3
IC	Integrated Circuit 36
KPN	Kahn Process Network 2
Lerc	Leiden Embedded Research Center
Liacs	Leiden Institute of Advanced Computer Science
LUMC	Leids Universitair Medisch Centrum 26
MD5	Message-Digest Algorithm 5. URL: http://tools.ietf.org/html/rfc1321
MJPEG	Motion JPEG 15
MoC	Model of Computation 2
MODWT	Maximal Overlap Discrete Wavelet Transform 26
OS	Operating System 7
PCI	Peripheral Component Interconnect
ptx	Parallel Thread Execution 36
SANLP	static affine nested loop program
SIMD	Single Instruction Multiple Data 35
SWT	Stationary Wavelet Transform 26
UML	Unified Modeling Language 40
XML	eXtendible Markup Language
Yapi	Y-chart Application Programmer's Interface. URL: http://y-api.sourceforge.net/