Massively Parallel Computing as a Service

Designing a service for use of massively parallel computation in a service-oriented architecture

Frank Kusters

Radboud University Nijmegen & Logica
Research number 624

Massively Parallel Computing as a Service

Designing a service for use of massively parallel computation in a service-oriented architecture

Massively Parallel Computing as a Service

Designing a service for use of massively parallel computation in a service-oriented architecture

Frank Kusters

December 1, 2009

Radboud University Nijmegen & Logica
Research number 624

Versioning

Date	Changes
March 17, 2008	Initial version
May 27, 2008	Updated chapters on definitions and architectures, added chapter on software
June 18, 2008	Expanded Clean code, added chapter on Robotarm
July 2, 2008	Rewrite of introduction and chapter on definitions
October 19, 2008	Added chapter on software estimation
October 22, 2008	Revised chapter on software estimation
May 12, 2009	Rewrite of chapter on architectures
June 4, 2009	Rewrite of chapter on service-oriented architecture, revised chapter on architectures
August 7, 2009	Rewrite of chapter on Hough transformation
September 23, 2009	Revised chapters on architectures, SOA and estimation
October 23, 2009	Added discussion and conclusion, replaced introduction, abstract and preface, revised all other chapters and appendices
December 1, 2009	Revised all chapters - Final version

Distribution

Recipient	Organization
Frits Vaandrager	Radboud University
Reinald Minnaar	Radboud University
Theo Schouten	Radboud University
Lambert Mühlenberg	Logica
John Pleunis	Logica

Abstract

As nowadays increasingly larger computer applications are developed, there is a need for faster and cheaper computer systems and for software architectures to improve development and deployment of applications. Many massively parallel processors being developed from GPU's (Graphical Processing Units) become available now, promising teraflops on the desktop. Service-oriented architectures are becoming popular because of their promise to separate the services provided by large enterprise systems and to hide the implementation from the users. Only speed and scalability are implementation dependent. In this thesis I studied the question of how to offer massively parallel computing as a service.

To answer this question I took a largely experimental approach by designing, implementing and testing an application as a service on a system consisting of a standard PC with an Nvidia Tesla C870 board consisting of 16 vector processors each capable of operating on 8 elements concurrently. The application was a Hough transformation taken from an existing image processing application at Logica. The needed implementation effort was large due to inadequacy of the development environment; several suggestions for improving this are made. An application speedup of up to a factor 10 was obtained, depending on the size of the relevant data set used.

Based on the above experiment I showed that with some effort a generic service can be built on a system consisting of a PC and one or more GPU based massively parallel processing cards. Regarding the effort estimation methods used by Logica, I determined that one of them, work breakdown structure combined with expert estimation, is an easy and accurate approach.

The resulting conclusion of this thesis is that offering massive parallel computing as a service is feasible.

Acknowledgments

This thesis was created as part of my master research project for the Computer Science master of the Radboud University Nijmegen, in cooperation with Logica. The idea for the project originated from Martin van Amersfoorth.

I would like to thank Lambert Mühlenberg, Frits Vaandrager, John Pleunis and Reinald Minnaar for their extended support during my research. Also, a profound thanks goes to all students participating in the Working Tomorrow program at Logica in 2008.

Venray, december 2009

Contents

1. Introduction

Massively parallel processors being developed from GPU's (Graphical Processing Units) become available now, promising teraflops on the desktop. Service-oriented architectures are becoming popular because of their promise to separate the services provided by large enterprise systems and to hide the implementation from the users. I will study the question of how to offer massively parallel computing as a service. Furthermore I will look at the software effort estimation methods used by Logica and determine whether they are the best fit.

1.1 The fields

In this section the current states of the fields of massively parallel computing, service-oriented architecture and software effort estimation are described.

1.1.1 Massively parallel computing

As larger and larger computer applications are developed, faster and faster computers and clusters are needed to execute these applications. Parallel computing divides processing and/or information of a computing task into partitions (called subtasks). Each subtask is then executed simultaneously, speeding up the execution measured in run time by approximately the number of partitions. However, due to fact that some parts of an application cannot be parallelized, the speedup is limited to relatively small amounts of processors. This is also known as Amdahl's law [3].

Gustafson showed that when the dataset is large enough, massively parallel computing (MPC) allows division of the information into arbitrarily many partitions or subtasks, therefore limiting the speedup only by the number of processors executing the instructions [21]. The division into arbitrarily many partitions is possible because of the limited dependencies between different partitions. Therefore MPC is only suitable for specific algorithms.

In hardware, massively parallel computing has recently gotten attention [31] partly due to the availability of the Nvidia Tesla, a processor which enables high performance computing on the desktop [13] and allows for relatively cheap high performance servers. The high performance is achieved by using multiple processors, which also perform instructions on multiple data elements simultaneously [29]. The hardware design necessitates software to be rewritten to fit the architecture.

1.1.2 Service-oriented architecture

On the software side, service-oriented computing is becoming increasingly popular [37, 11]. Service-oriented architecture keeps every subsystem (a service) of a large enterprise application separate, thereby allowing for (among others) more reuse of these services.

Essentially (and ideally), the implementation of a service is hidden from the outside world. Whether or not a service utilizes massively parallel computing is therefore an implementation detail and should manifest itself only in fast response times. However, there are many algorithms and applications that could be enhanced using massively parallel computing, which would cause similar code to appear several times in a project (or enterprise). SOA strives to prevent the last notion by creating a service that can then be used throughout the enterprise.

1.1.3 Software effort estimation

Software effort estimation focuses on the amount of time (the effort) it takes to complete a software project. Software effort estimation is hard, as shown by Moløkken and Jørgensen [28]. Most software projects exceed their allocated time and/or budget. To increase the estimation accuracy (the deviation of the actual effort from the original estimate), many estimation approaches have been developed, with varying success.

1.2 Research questions

The aim of my research is to design and build a 'massively parallel computing' service: it should offer massively parallel computation while hiding the implementation details of the underlying hardware architecture. Ultimately, developers and architects should be able to use massively parallel computing in their services without in-depth knowledge of the hardware architecture, which is very necessary at this time [33, 5]. I do not aim at automatic parallelization of existing services, as that is a different subject altogether.

An implementation of the aforementioned service has a significant impact on the cost of development of new software that uses massively parallel computing. It is therefore useful to quantify this impact, i.e. find out how much it will cost to build a new service on top of the designed service.

To this end, I have devised the main research question as follows:

> *How can massively parallel computing be offered as a (business) service?*

To be able to answer the main question, I developed ten subquestions for my research proposal. These are divided in specific questions and generalizations.

1.2.1 Specific questions

1. How are 'massively parallel computing', 'service-oriented architecture' and related terms defined in literature?

2. How can a service be designed and constructed such that the specific case of 3D image rendering can be performed on a general purpose massively parallel processor?

3. How can the service be adapted into a demonstrator suitable for display on events and to clients?

4. How are methods for cost estimation of software projects outlined in literature? Which of those methods, if applicable, does Logica use and how does it compare to other methods?
5. What are the costs of a business service based on 3D image rendering, based on the cost estimation method of Logica? Are there clients of Logica for which such a business service would be attractive (i.e. cost-effective)?

1.2.2 Generalizations

6. What are the criteria to determine if an algorithm is suitable for massively parallel computation on architectures like those of the Nvidia Tesla? Which classes of parallel algorithms satisfy these criteria?
7. How can a generic service be designed such that the implementation of the applicable classes of algorithms on a general purpose massively parallel processor can easily be carried out?
8. What are the advantages and disadvantages of using the aforementioned design over existing systems?
9. What kind of business services can be offered using the developed design?
10. What are the business benefits (if any) to Logica for offering this business service?

1.3 Related research

I am not the first to combine the concepts of massively parallel computing and service-oriented architecture. Some researchers have implemented a highly parallel computing infrastructure *using* a service-oriented architecture [16, 20]. Although it is an interesting way to achieve distributed computing, it does not solve the problem of making integration into an existing SOA easier. Furthermore, massively parallel computing and distributed computing are not the same architectures, which will be explained in section 2.3.2.

Some have used massively parallel computing in a SOA, but because it wasn't the main research goal, no details are given on the implementation [34]. This is not the case with Hawick et al., who designed and implemented a service for massively parallel computing in 1998 [22, 35]. The concept of a "service-oriented architecture" didn't exist, but they did touch many of the issues that arise in designing a service. They also used a massively parallel computer, a "Connection Machine" (which is a supercomputer design from the early nineties). However, my research focuses on the more modern Nvidia Tesla instead.

1.4 Structure

The report is divided into four themes. The first theme, "architectures for parallel computing", is discussed in chapter 2. Chapter 3 is about "software design in a service-oriented architecture". In chapter 4, the implementation of a real service is handled. Then software effort estimation methods are described in chapter 5. Chapter 6 discusses the results. Lastly in chapter 7 a conclusion is drawn from the research.

2. Architectures for parallel computing

2.1 Introduction

As more and more massive computer simulations are developed, colossal datasets are analyzed, and gigantic programs evolve, a need arises for an enormous amount of processing power to deal with these programs. Traditionally, this problem had been solved by building either special processors (an expensive undertaking), building computer clusters from commodity parts (also relatively expensive), or computer clusters from special processors, the highest priced possibility.

Because of the price, such equipment was mainly reserved for large research departments within universities and enterprises. However, there are plenty of companies who would like to have a decent amount of processing power without the steep bill, and until recently they had to make do with tiny versions of those computer clusters with commodity parts. In November 2006, this changed. Three similar but different products were introduced independently by three companies. These are:

- the Tesla family of processors, by Nvidia,
- the FireStream family of processors by Ati (now AMD), and
- the Cell processor by STI, a joint venture of Sony, Toshiba and IBM.

These products all have one thing in common: they are special processors available for the price of a common processor. The rationale behind the low price is fairly straightforward. The processors are the same as those found in popular consumer products (video cards and game consoles), which means millions of them are produced (so actually, they *are* common processors). This allows the manufacturers to spread the R&D costs, thereby lowering the price of the final product substantially.

The marketing brochures of the above-mentioned processors say they can run programs up to 200 times faster compared to Intel and AMD processors, depending on the application. This has to do with a radically different architecture being employed. This is the subject I will cover in this chapter. In the next section definitions are given for much-used terminology. Section three explains different kinds of applications and the computer architectures that exist to handle these applications. In the fourth and fifth section, the actual architectures of the most common processors are explained: the x86 architecture and the new GPGPU architecture. Finally, I will discuss the differences.

2.2 Definitions

There are many terms used in the world of parallel computing, often interchanged. For clarity, I will give some strict definitions for terms used in this chapter. I divided the terminology between conceptual terminology, which is about the mathematical part of algorithms, and implementation terminology, which is about the technical implementation of an algorithm (actual code and execution by a computer).

Conceptual

job All calculations together; the entire computation, the undivided application.

task Part of a job [17, page 86]. A job is split into one or more tasks. Tasks can be run in parallel to speed things up (see also 'process'). Tasks can themselves have subtasks.

granularity The average size of tasks in a job [2, page 15]. The more fine-grained (i.e. smaller) tasks are, the more parallelism is possible, and the more overhead is induced.

overhead Processing time that is not spent on the job, but used for communication and synchronization. (Of course, overhead should be reduced to a minimum, which means that a program must be designed so that a minimal amount of communication and synchronization between tasks takes place.)

embarassingly parallel An algorithm that is so easy to parallelize, with so little overhead, that it's 'embarrassing'. For example, the well-known brute-force search is embarrassingly parallel. (When finding a password that decrypts a certain encrypted file, each password-check is independent from every other password-check.) It is especially suited to grid computing, but actually really useful on all highly parallel architectures (more on that later on, in section 2.3.2).

Implementation

program The actual implementation of a job.

process The actual execution of a task [17, page 203]. Think of it as the difference between a mathematical formula and the calculation of the result the formula. A process is more like the latter.

thread Also, the actual execution of a task. In hardware, there isn't much of a difference to a process: a common processor couldn't care less if it is executing a thread or a process. The operating system does care: threads share memory with each other, while processes explicitly do not. Since I am talking about hardware architectures, I will use both thread and process; whichever suits the text best.

processor To the outside, the processor is the executor of a thread or process. In a single computer, the processor is the CPU. In a distributed computing environment, which consists of many computers (see page 26), there are many processors; I will refer to them as processing elements. The load balancer of such an architecture resembles the processor, as it delegates the task at hand to one or more processing elements.

processing element The executor of a thread or process, as part of a bigger system like a CPU or a distributed computing system.

2.3 Parallel models and architectures

Parallel models illustrate the way a program is structured. Parallel architectures specify the way the computer(s) is (are) organized to execute such a program.

2.3.1 Parallel models

Grama et al. describe several commonly used parallel algorithm models. These models are “a way of structuring a parallel algorithm [..] and applying the appropriate strategy to minimize interactions” [17]. And by ‘interactions’, they of course mean overhead. The following models are described (they are shown in figure 2.1 on page 27):

pipeline model Also known as *producer-consumer model*, this model allows a stream of data to flow through different processes, where each process performs a task on the data. Each process consumes the data that the previous process has produced, hence the name producer-consumer model. A course-grained granularity causes the pipeline to take a long time to fill up, whereas a small granularity causes more overhead to transfer data between processes, necessitating a trade-off decision. (a)

data-parallel model Data is divided into equal-sized partitions. Similar or identical tasks are then performed by the processes on the different partitions. Interprocess communication is usually performed via some kind of synchronization point. This type of parallelism is also called *data parallelism*. (b)

work pool model Any task with the accompanying data may be performed by any process. A central data structure is used where each process gets and returns its task and data. This model is very suitable when the amount of data is small but the computation of the task is large. (c)

master-slave model A *master* or *manager* process generates and assigns work to different *slave* or *worker* processes. This is useful in case the work must be done in phases. The master process can preserve the order of the subtasks, and force worker processes to synchronize. (d)

task graph model Knowledge about tasks is used to map them onto processes in a way that minimizes communication overhead. This is especially useful in cases where different tasks with little computation but large data sets are used. The mappings of the tasks can be naturally expressed in a dependency graph. It is also called *task parallelism*. (e)

hybrid model Any hierarchical or sequential combination of any of the above models. For example, each process in a pipeline model may itself employ the data parallel model. Or, when a work pool task has finished, it may be post-processed using a task graph model. (f)

2.3.2 Parallel architectures

As stated before, parallel architectures define the way computers are organized to execute a parallel program. Often, an architecture is very suitable for execution of one specific type of parallel model. Figure 2.2 displays the relations betweens the parallel architectures and I will explain them here.

parallel computing Executing multiple tasks simultaneously. This involves the use of multiple processing elements (each executing one task), because parallelism is not possible under the Von Neumann architecture (this is explained in more detail in section 2.4, ‘x86 architecture’). *Parallel computing* is a generalization of all architectures.

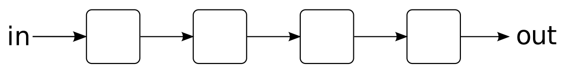
multiprocessing The use of multiple processors inside one computer, operating independently of one another. These can be integrated in one chip, or communicating on the same bus. They share all I/O and share all memory. Typically two threads can be executed simultaneously (the Intel Core Duo processor comes to mind), but more threads is possible (e.g. with multiple Core Duo processors). This architecture is suited to the task graph model, because of the all-round capabilities of the processing elements. It is expected by some that this parallel architecture will lose its importance, due to the increasing number of cores on consumer processors.

highly parallel computing “Using a large collection of processing elements that can communicate and cooperate to solve large problems fast” [2]. This is a generalization of several types of parallelism, as detailed below. It is also called *high performance computing*. Each of these architectures matches a different parallel model.

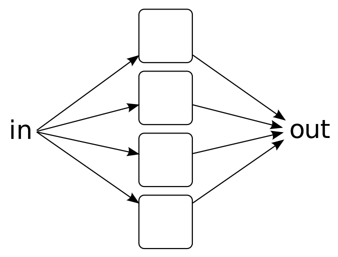
massively parallel computing Making multiple processing elements appear as one computer. All I/O goes through one interface, as there is virtually no communication between the processing elements. This allows massive scaling, because overhead is usually the bottleneck when scaling a system. The data-parallel model fits this architecture very well, since the data is not interrelated.

distributed computing Using multiple computers (i.e. separate computers connected via a network) for parallel computing. The speed of the network is a major factor in the response time achieved by the system, because of the overhead of network communication. This too is a generalization, of the following two architectures:

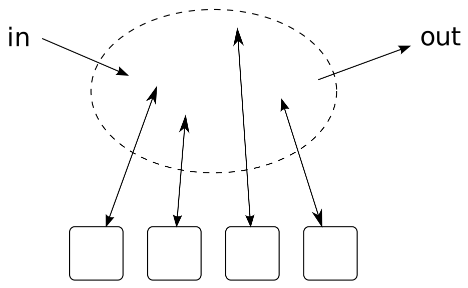
cluster computing A distributed computing setup with a known and controlled network, usually at the same geographic location. The hardware and software used on computers in the system is very similar or the same. The response time is usually several orders of magnitude shorter than that of a grid computing network. This works really well with the master-slave model, as the master knows exactly what the slaves are capable of and can distribute the work accordingly.



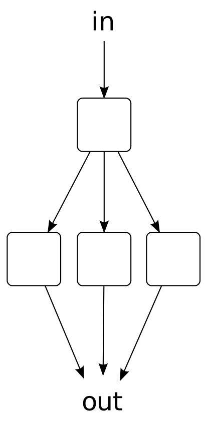
(a) The pipeline model



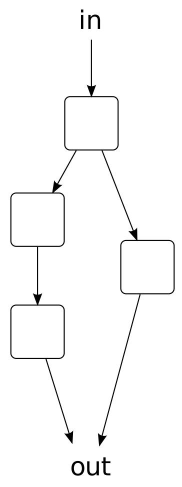
(b) The data-parallel model



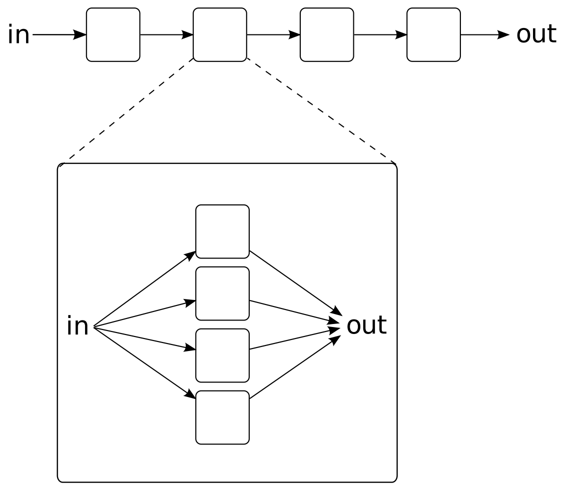
(c) The workpool model



(d) The master-slave model



(e) The task graph model



(f) A hybrid model

Fig. 2.1: The commonly used models for structuring parallel algorithms

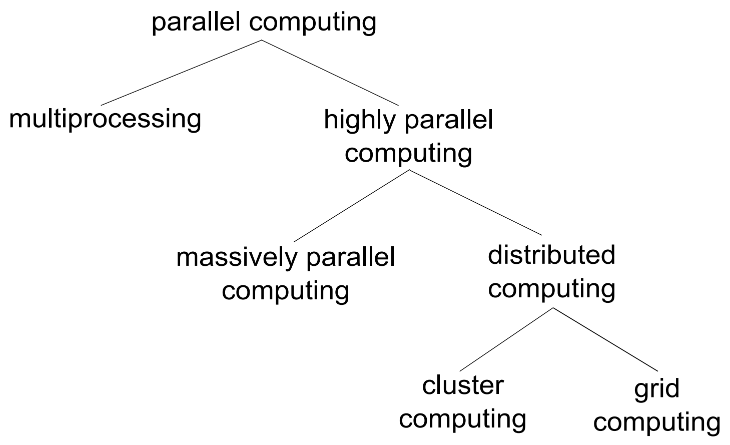


Fig. 2.2: The relationships between parallel computing architectures

grid computing Grid computing is the opposite of cluster computing. Multiple computers are joined in a network that is uncontrolled and potentially hostile. Any type of hardware and software may be combined. This naturally fits the work pool model, where processing elements pull the data from the pool. The processing elements are not under any type of control, so pushing data is out of the question.

cloud computing Cloud computing pops up every once in a while in the context of distributed computing. However, it is entirely different. In cloud computing, there is a grid or cluster running the software or environment needed, so all one has to do is rent time and space on it. This is more of a business concept than a hardware concept.

You may have noticed that the pipeline model is not matched to any of the parallel architectures. That is because it is often used in embedded hardware or integrated circuits where there are a lot of specialized processing elements. Pipelining in the Unix-world is not as often used to parallelize and speed things up. It can be though, and in that case usually employs the multiprocessing architecture.

2.4 x86 architecture

In this section, I will discuss the x86 architecture, while the GPGPU architecture is discussed in the next section.

Both types of processors use the Von Neumann architecture, which is a way in which programs and data are handled by an execution unit. It is the only mainstream architecture for general purpose computing; others, e.g. Harvard architecture, are only used in embedded or exotic hardware. John von Neumann, who designed the architecture in 1945, put the program in the same part of the computer as the data, allowing the program to be data, effectively copying the universal Turing machine. To save on costs, no mechanism for communication or synchronization for parallel computing was provisioned: the program is executed instruction by instruction, in sequence [2, page 109].

The x86 architecture implements the Von Neumann model. It originated from Intel with the advent of the Intel 8086 chip (of which the 'x86' name stems) in 1978 and built upon an architecture designed by Computer Terminal Corporation in 1970. It is the most popular architecture today in consumer and enterprise computers.

Because the Von Neumann architecture does not take parallel computing into account, Intel devised a number of innovations which allow for parallelism (and thus speedup) but leave the appearance of single-threadedness intact (implicit parallelism). With the 80486 processor, Intel introduced *pipelining*. The consecutive Pentium processor was a *superscalar* processor. The Pentium MMX processor added *SIMD* capabilities. *Hyperthreading* was introduced on the Pentium 4, although it breaks with the paradigm of one instruction at a time (making the parallelism explicit). And finally the Intel Core Duo processor was the first Intel processor for consumers with *multiple cores* (essentially two processors on one chip). I will now discuss these five methods of parallelization in detail, using a car trim shop as illustration.

2.4.1 Pipelining

Pipelining (which was discussed earlier in section 2.3.1) is very similar to an assembly line. Data passes by several execution units, which modify that data in succession [17, page 12]. This is much like the assembly line, where a car passes by several workers who each work on a seperate part of the car. So the painter works on the car first, the car rolls to the upholsterer, and so forth. The assembly line has the advantage that the car does not have to be carried back and forth to each worker. (It would be really inefficient if you would put the car in storage between each worker, which is exactly what a processor without pipelining does.) Similarly, data doesn't have to be stored and fetched after every instruction or operation.

2.4.2 Superscalar

A superscalar processor utilizes the same principle as with pipelining, but it is a bit different. The processor accepts instructions faster than it could execute them normally. These instructions are then dispatched to different available execution units in the processor [17, page 12]. Of course, this is often combined with pipelining.

Suppose there is a painter and an upholsterer in the trim shop. Several cars are ordered: two need to be upholstered and one just needs painting. A traditional trim shop only handles one car at a time: one car is taken in and upholstered, another car is taken in and upholstered, and the final car is taken in and painted. It would be more efficient if, after the first car for the upholsterer is taken in, a *second* car would be taken in so one of the painters can get to work.

2.4.3 SIMD

The last optimization added to the x86 processor is the SIMD capability. SIMD stands for 'Single Instruction, Multiple Data' and simply means that the processor can modify more than one data element at a time. (Like having two painters in the trim shop to paint two cars at the same time.) This is a very effective speedup because only one instruction needs to be fetched and decoded for every two, four or eight elements. The (small) drawback is that the compiler needs to be adapted to profit from it, because special instructions are used.

2.4.4 Hyper-threading

Hyper-threading allows the processor to advertise itself to the operating system as two processors, although it is only one. The principle is simple: it is similar to a superscalar processor. Only, the superscalar processor accepts instructions faster than it can execute them, while the hyper-threading processor accepts the instructions in parallel. The trim shop now has two doors where cars can be taken in instead of one. For software to take advantage of hyper-threading, it has to be multi-threaded (see the next section).

2.4.5 Multicore

Another method to achieve parallelism is just using more than one processor on the same chip. They are physically seperate circuits, but they share the same surrounding hardware like memory. There is no longer just one trim shop: it is more like two trim shops in the same building. Using a multicore processor is a form of multiprocessing because only a few processing elements are used.

Now communication and synchronization come in to play. There are two ways to solve this:

Multiple programs

The first way is to execute entirely different programs at the same time. These are different processes, each with their own instruction code and their own memory space. The operating system then assigns the programs to their own processors, so the programs won't even know the other program is running. In this case, the advantage over the use of a single-processor system is that the processor doesn't have to keep switching tasks, which incurs a lot of overhead. The execution speed has effectively doubled (provided that there are no I/O bottlenecks like a harddisk).

Programs running on such a system are very easy to implement. Any program that does not make assumptions on the state of the system during execution will work. An example of such an assumption is that a specific file will keep existing during the lifetime of the program. This is not necessarily true because another program might delete the file. (Technically, this could also happen in a single-processor system, although the condition is less likely to occur.)

Divide programs

The second way is to divide one program into subtasks. Each subtask is assigned to a thread. Each thread is run on its own processor (again, the operating system takes care of which thread runs on which processor). The difference is that threads share memory with other threads, and possibly share code. Now it starts to get a bit difficult, because the programmer himself has to take care of communication and synchronization. He has to provide the threads with ways of knowing what the state of other threads is, he has to divide the work into subtasks, etc. A compiler can help with this job (for example in Java there is the `Thread` interface), but the majority of the work has to be done by the programmer.

2.5 GPGPU architecture

We have seen how the x86 architecture works with regard to parallelism. Good advances have been made, and especially the incorporation of more than one processor in the same chip has made the technology of parallel computing available for the masses. However, to achieve *massively* parallel computing, the number of processing elements has to be scaled way up. That is hardly possible with the traditional x86 architecture because the level of overhead rises too high. The amount of communication in typical multiprocessing applications increases polynomially with the number of processing elements; this also depends on the parallel model used.

Moreover, adding more chips means adding additional unnecessary circuitry: x86 chips have circuitry for handling all kinds of I/O and instruction scheduling (which is a substantial portion of the chip). A lot of this does not get used in a multiprocessor system. But producing a chip without this circuitry is too expensive: the design has to be thoroughly tested and fabricating a different kind of chip incurs a lot of extra costs. All this for a relatively small market.

This is where GPGPU - which stands for 'General Purpose Graphical Processing Unit' - comes in[1]. Powerful graphical processing units have been common in consumer computers since the advent of the 3Dfx Voodoo processor in the mid-nineties. They accompanied the rise of 3D games, which require a lot of identical computations: the rendering of thousands of 3D polygons to a 2D screen. These processors are nowadays common in all mid- to high-end PC's, which means they are produced in enormous quantities, which equals a low price per unit. These processors are not encumbered with the I/O capabilities and instruction schedulers of x86 processors. The former are plain unnecessary, while the latter is handled by the compiler at compile time.

But, we do need to be able to write something other than games for such a processor. Several manufacturers created processors which allow this, namely Nvidia, AMD, and STI, which is a joint venture of Sony, Toshiba and IBM. The basis of each of their products is the same: 'Multiple Instruction, Multiple Data', or MIMD. To implement this, two techniques are used that are not present in the x86 architecture: vector processor and stream processing. I will explain those techniques first, and then elaborate on the different products.

2.5.1 Vector processing

Vector processing is actually very similar to SIMD that is used by the x86 architecture. It employs the data-parallel model: an operation is fetched and

[1] There have been other attempts at solving the problem of using the same chip for both normal and highly parallel processing. In 1987 there was a company called INMOS that built 'transputers', (then) cheap processors which, when combined, could build both small desk computers and super computers. It was more like a cross-over between massively parallel computing and cluster computing. Unfortunately the company failed, but their ideas are used in many of the concepts mentioned in this chapter.

decoded once, and then executed on different operands, thereby saving fetch and decode cycles for those operations. However, unlike SIMD in x86, which only works with a few data elements (four, maybe eight) and with a few specific instructions, vector processors do this on a lot of elements: 16 or even more, and with every instruction. (The trim shop has grown to employ 16 painters.) A significant speedup for the vector processor is achieved because it does the fetch and decode only once for all 16 data elements, and then executes on each data element, yielding 1 fetch + 16 execute = 17 cycles, saving 15 fetch and decode cycles.

2.5.2 Stream processing

Because GPGPU's can work on hundreds of thousands of data elements at a time, it would be kind of troublesome for the programmer to instruct each processing element seperately. It would be a lot more efficient to automate this; the work can be more efficiently distributed over the processing elements.

So, the programmer divides his set of data into subsets. The compiler builds the program using thousands of threads. Then, the same group of instructions is executed on each subset. The compiler takes care of distribution of the data, communication and synchronization, with some aid of the programmer. So instead of a corporate customer (who needs several hundred cars trimmed) outside the trim shop deciding what each worker must do, the trim shop has instead hired a manager that tells the workers what to do. The master-slave model fits nicely here.

Also, most compilers for GPGPU's are built smart enough that they can utilize the multiple vector processors which are available in a pipeline fashion, further speeding things up.

2.5.3 Implementations

As mentioned in the introduction, there are three products available that implement this architecture. They are very similar in design, so I will mainly describe their hardware configuration and performance.

Nvidia Tesla

The Nvidia Tesla is the name of a family of products made by Nvidia. The first members of the family were based on the Nvidia G80 processor. One of these processors is put on the Tesla C870 card, which is the card I used for my research. It has 16 vector processors, each capable of operating on 8 elements at the same time, for a total of 128 threads. It has 1.5GB of memory. Such a card can then be bought separately as a PCI Express card (model C870), in duo in a desktop casing (model D870) or in quadruple in a server casing (model S870). The C870 is capable of 519 billion floating point operations per second (GigaFLOPS).

In June 2008 Nvidia introduced the successor to the G80: the G200. With 30 vector processors instead of 16, the number of threads that can be handled simultaneously leaps to 240. Also the clock speed increased a bit, so the processing power is 936 million floating point operations per second (almost one TeraFLOP). This is available as a PCI Express card with 4GB of memory (model C1060), or, like the S870, in a server case (model S1070). The latter has an even higher clock speed, and can perform 4.3 TeraFLOPS. The Intel Core i7 (an x86 processor), which was introduced November 2008, is capable of only 51 GigaFLOPS, for a price about equal to the C1060.

The G80 and G200 processors are a coprocessor to the main processor, typically an x86 processor as mentioned in the previous section. The main processor (or host processor) does all the 'dirty' work of running the operating system, performing I/O, etc. To limit the amount of thread switching and the amount of data that needs to be downloaded into the device's memory, it is most efficient to use it for applications with a high arithmetic intensity, i.e. with a lot of (mathematical) operations per data element.

AMD FireStream

The AMD FireStream series is similar to the Nvidia Tesla family. Since I have not used any of the older specimen, I will just outline the specifications of the newest board, the AMD FireStream 9270, containing the Radeon R700 chip, which was released in November 2008. It has 160 vector processors, arranged in 10 groups of 16 processors. Each vector processor is capable of processing 5 data elements at a time, allowing for 800 threads to execute concurrently. The board has 2GB of memory, and does 1.2 billion floating point operations per second, slightly more than the Nvidia Tesla C1060, at roughly the same price.

STI Cell processor

The STI Cell processor is a processor jointly developed by Sony, Toshiba and IBM (it is also called the IBM Cell processor). It is best known for its use in the Sony PlayStation 3 game console. The Cell processor is the odd man out in this lineup of GPGPU's, because it is a hybrid design: it combines a number of vector processors with a more all-round processor.

For business/development use it is available on the Mercury Cell Accelerator Board 2 and the Fixstars GigaAccel 180. Both are PCI Express boards including respectively 5 and 4GB of memory, and are capable of running an entire operating system by themselves (due to the all-round processor). These boards are expensive: they cost around 4 to 5 times as much as the Tesla or the FireStream. The world's fastest supercomputer (since June 2008) encompasses 6,480 AMD Opteron processors and 12,960 IBM PowerXCell 8i processors (that model is a slight variation on the 'normal' Cell processor with better double precision floating point performance).

As said before, the Cell is essentially a combination of a coprocessor and host processor. The 'host processor' is the Power Processor Element, which is a PowerPC[2] processor in itself, capable of executing entire applications and even an operating system. It can, at the will of the programmer, give highly parallel arithmetic tasks to the Synergistic Processing Elements (SPE). An SPE is actually just a vector processor capable of performing instructions on 16 data elements at the same time. Thus, 8 SPE's execute 16 threads simultaneously, for a total of 128 threads. It can achieve 218 GigaFLOPS with this method.

[2] The PowerPC architecture is a competitor of the x86 architecture.

2.6 Discussion

In this chapter, I have given an overview of parallel models and architectures, and shown how the x86 and GPGPU architectures differ from each other. The basis of executing a program step by step from memory is the same; the distinction lies in the capability to execute an instruction on a lot of data elements at the same time, using techniques like vector processing and stream processing.

The speedup of 200 times when compared to conventional x86 processors can only be gained when the program fits in the data-parallel model and the master-slave model at the same time, because only in this specific case can advantage be taken of all processing elements.

Due to the widespread availability of GPGPU's, massively parallel computing is now reachable for the masses. Already a great number of applications are being developed that were deemed inconceivable before. It also seems that, with some effort, a lot of programs can be squeezed into the data-parallel/master-slave hybrid model[21].

I have chosen the Nvidia Tesla C870 board for my research because at the beginning of my research, it was the cheapest board of the ones available and it had the most documentation. Also, AMD FireStream products were not available when I started my research. When my research was finished the differences between the boards had diminished; there are even efforts to integrate the software for the Tesla and FireStream products (called OpenCL).

3. Software design in a service-oriented architecture

3.1 Introduction

Service-oriented architecture addresses the problem of software being so interconnected and so large that no-one alone has oversight and it is almost impossible to change a subsystem without breaking something else. It does this by, so to speak, putting every subsystem on its own island, with defined shipping routes (swimming is not allowed).

I want to build my software as a service because I'm trying to make massively parallel computing available for the entire enterprise. If I can put massively parallel computing into a service, each new service that is built can use it with relative ease, thereby leveraging the enormous power of MPC without the drawback of complicated software design. In this chapter, I will show what is needed to build such a service.

In the next section I define SOA and the related terms. In the third section, different kinds of services are described. The fourth section enumerates all the design ideas taken into account and the final section shows what kind of service I will build and how the design principles apply to my design.

3.2 Definitions

The field of service-oriented architecture has been extensively described by Thomas Erl [11, 12]. Because many researchers refer to his work, this chapter will contain many of his ideas.

Erl takes several chapters to define and describe concepts and terms related to SOA. Some terminology and concepts are shown in figure 3.1 to make their relationships clearer. I have summarized Erl's definitions and descriptions of the concepts as follows (quoted parts are by Erl [12, pages 37-42]):

service "A physically independent software program with distinct design characteristics". These design characteristics are related to standardization, coupling, abstraction, reusability, autonomy, statelessness, discoverability and composability. Each of these is described a few sections further on. Services are often referred to as *web services* in literature [4].

A service is usually built from three parts: a service contract, the actual program that executes and processes the data, and a message processing system that captures and translates messages for the actual program.

service-oriented design paradigm A programming paradigm that is centered around services, much like object-orientation is centered around objects.

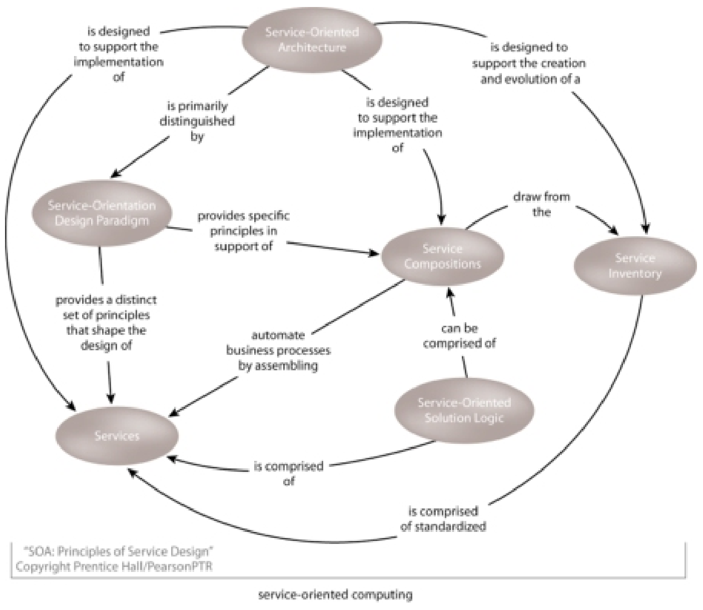

Service-Oriented Architecture
is designed to support the implementation of
is primarily distinguished by
is designed to support the implementation of
is designed to support the creation and evolution of a
Service-Orientation Design Paradigm
provides specific principles in support of
Service Compositions
draw from the
Service Inventory
provides a distinct set of principles that shape the design of
automate business processes by assembling
can be comprised of
Service-Oriented Solution Logic
Services
is comprised of
is comprised of standardized
"SOA: Principles of Service Design" Copyright Prentice Hall/PearsonPTR
service-oriented computing

Fig. 3.1: The service-oriented computing concepts in relation to each other (diagram by Erl [12, page 41])

The paradigm comprises the principles which I mentioned in the previous paragraph.

service-oriented solution logic Software that has been built according to the service-oriented programming paradigm.

service-oriented architecture An architectural model in which "services are the primary means of representing solution logic".

service inventory A collection of services that form a coherent system. Usually (entire) business processes of a company can be expressed in services from the service inventory.

service composition As the name implies, this is a composition of services, automating (part of) an entire business process. The difference between a service inventory and a service composition is that the inventory is just a collection of services, whereas a composition is an actual software application that does something (hopefully) useful.

orchestration The combination of services to perform a business process. The difference between a service composition and orchestration is that a composition is a hierarchical composition of services, whereas orchestration is a sequential combination.

service contract This is the technical contract (much like an API), which can be read by a computer. A possible implementation of this is through a *WSDL definition*. WSDL stands for Web Services Description Language, which is the XML way of describing the interface of web services.

choreography Any protocol that defines the way in which services must interact with each other [32]. In other words, a specification of the sequence of messages that a service can receive and send. This is an extension of the contract.

Finally, I use the term **business service** for “a business activity that often results in intangible outcomes or benefits” [4]. So in this thesis *service* is used in the technical (SOA) sense of the word and *business service* is used in the business sense of the word.

3.3 Service models

Erl has identified three different service models [12, page 43]), which he defines as follows:

entity service Much like an object in object orientation, an entity service defines an entity that is part of the business process (but does not know about that business process) with possible actions related to or on that entity. An example of such an entity could be an invoice entity, which can have the standard ‘create, read, update, delete’ actions and actions like ‘send’ or ‘pay’. The actions are only related to this entity. It can still be composed out of other services.

task service A task service encapsulates (part of) a business process. That is, it carries out a sequence of actions, involving multiple entity services. A task service is a separate category because it “spans multiple entity domains and does not fit cleanly within a functional context associated with a business entity”.

utility service Utility services provide functionality that is not tied to any business process and is often technology oriented. This functionality can be anything, from exception handling to complex mathematical formulae.

3.4 Design principles

Erl has devoted a book specifically to explaining how services should be designed: *SOA Principles of Service Design*[12]. Since I am designing a prototype service for use in a service-oriented architecture, I will follow his guidelines (‘design principles’) This section shows how his guidelines apply to the service I am designing.

3.4.1 Contracts

The first design principle Erl talks about is the standardization of ‘contracts’. The contract states what a service does. This section is about how to design the contract, which is not the same as designing what the service does. Erl mainly suggests two things: 1. standardize the commands accepted by the service and 2. standardize the data representation.

Standardization of commands accepted by a service reduces the possibility of confusion of service customers. If you build a `Cat` service with a ‘capability’ `meow()` and another service `Dog` with a capability `woof()`, it will be hard to find the right capability for the job. They are easier to find when both capabilities are called `makeSound()`.

Erl states that it is a good idea to standardize your data representation (or data model). This saves both development time and processing time, because you don't have to transform or convert your data for every service before the service can work with the data. If you transmit a full date ('`May 22, 2009`') with one service, and the other service expects a simple date ('`2009-05-22`'), you need a conversion layer which adds complexity and thus increases the probability of errors.

Erl also talks about standardization of service policies. I do not discuss these, as they are not related to my research on massively parallel computing.

3.4.2 Coupling

Coupling is the extent to which entities are connected to each other. I use 'entities' because it is not necessarily about connections between services; one can for example speak of coupling between a service and its contract or between a service and a data representation. Coupling is a tricky design principle. It is tricky because you need it (how are services going to communicate without it?), but you would like it absent as much as possible (to avoid too much dependence). So there is a balance to be struck.

Erl lists five types of coupling within a service, which don't exclude each other:

1. Logic-to-Contract coupling. This is a preferred type of coupling, where the contract is designed first and the 'solution logic' second. This should result in logic that actually does what the contract says, instead of the other way around:

2. Contract-to-Logic coupling. The typical developer's approach: build it first, then make a contract for it. Which of course has the disadvantage that no actual design principles are applied to the contract at all.

3. Contract-to-Technology coupling. The same as contract-to-logic coupling, only the logic already exists, leaving the developer no option but to add a contract to it. (Alternatively the developer can make a contract, then add some conversion layer. This of course initially adds development time.)

4. Contract-to-Implementation coupling. This kind of coupling often exists when interacting with the outside world in the form of databases, files, user accounts, etcetera. When dealing with these connections, it is preferable to minimize the number of services that use them. Also beware that the database schema doesn't sneak into the XML data representation, creating a dependency on the database.

5. Contract-to-Functional coupling. This happens when service A comes to rely on the way service B works, while B uses A to perform some function for B. It is now effectively impossible for A to also perform a function for C, making it tightly coupled.

3.4.3 Abstraction

When talking about abstraction of a service, Erl actually talks about the abstraction of meta-information of that service. He speaks of four types of meta-information: technology, functionality, programmatic logic and quality of service information.

The abstraction of technology is fairly straightforward: why would I, as a customer, care about what technology the service uses to accomplish its task? If it does what it advertises (i.e. does what is in the contract), the technology used is irrelevant. That also means there should be nothing in the contract that is specific for the technology. The exact same reasoning can be used for programmatic logic.

Quality of service (QoS) information is a bit different. Here, the QoS offered is abstracted from the reasons for this specific QoS to be offered. So, if the service is unavailable on sunday night, that is what should be stated in the contract. The reason why (maintainance) is abstracted away, because it is not of interest to the customer.

The abstraction of functionality is the odd man out. Erl says this (abstraction of functionality) happens when the service actually has more capabilities than advertised in the contract. This seems only applicable in case of contract-to-technology coupling. A program is available that does all kinds of things, but only one of these things is needed for the service to do its duty. So, a contract is added to the program that only advertises the single capability. Ideally the logic is built after the contract is designed, and therefore there is no more logic than advertised.

3.4.4 Reusability

The chapter on reusability boils down to two things: concurrent accessibility and genericity. Concurrent accessibility is easy to explain, but less easy to actually do: if a service is reusable it is might be used by multiple customers at the same time. If the service is not prepared to handle this it will malfunction, and is therefore not reusable. However, if the service manages a single resource like a printer or a massively parallel processor, tasks must be handled appropriately (a task queue comes to mind). More handling logic equals more complexity.

Genericity is best explained by the use of an example. Let us consider entity services. An `Employee` service should be able to be used in a composition with an `Invoice`, `Complaint` and `Pay slip` services. The generic design of the `Employee` service seems reasonably straightforward (most likely it contains an employee ID, a name, role, salary, etc. and their modification capabilities). However, a `Form` service is needed, that should also be used in composition with `Invoice`, `Complaint` and `Pay slip`. Now it gets tougher, because whereas `Employee` is used by the other services, `Form` uses the other services itself. Still there are advantages to making a generic `Form` service, because they could be used everywhere in your service-oriented architecture, making other services (for example a search service) more generic.

Some risks emerge (which are of course described in Erl's book), and I'll highlight two of them:

- Reliability concerns. Suppose a service is built which the entire company uses in its service compositions. If the service becomes unavailable, all the other services would fail too. There are methods to fail gracefully, and it is important to take those methods into account when designing a service. Example methods are having a backup service standby, or splitting message processing from the programming logic, so no messages are lost.

- 'Agile delivery' concerns. Making a service more generic means more work. If the service is not reused, all the time spent making the service generic was wasted. So one needs to put serious thought into making a service generic.

3.4.5 Autonomy

An autonomous service means that a service is in control over its environment and especially its resources. Preferably, this control is exclusive, so the service can guarantee consistent, scalable performance. Naturally there is a trade-off, because as more and more consumers use the service, the less autonomous it becomes. A completely autonomous service would be able to be deleted without causing trouble. It is obvious that this ideal cannot be achieved. This part of autonomy is already covered in the Coupling section. Erl has a nice definition for a really autonomous service. It is called a service with Pure Autonomy, where "the underlying logic and data resources are isolated and dedicated to the service".

This design principle also poses some risks of which one is worth noting: overestimating service demand. Building a massively parallel computing service which runs a simple calculator is overkill. This adds cost to implementation and infrastructure. This can be solved by thorough examination of how big the service needs to be, and making it easy to upgrade should the need arise. In my project this problem is not relevant as the hardware available to me is fixed.

3.4.6 Statelessness

A service that keeps data when it is not actively processing that data is stateful. A service that does not do this is stateless. Erl goes on to discuss a handful of state types, types of data, and so on, which are not relevant for my research.

A stateless service needs to receive and send its data before and after each activity. This has an impact on its performance. To make statelessness more feasible Erl advises focusing on start-up times: "... transition from an idle state to an active processing state in a highly efficient manner". A way to accomplish this is parsing XML messages using 'high-performance parsers' and 'hardware accelerators'. Erl recommends doing a 'performance assessment', in other words a benchmark; just try different ways of doing things and see if the speed difference is significant.

3.4.7 Discoverability

Whereas the standardization design principle is geared towards contracts that are read by machines, discoverability is really about humans. We still don't have computers that program themselves, so we need humans to connect services to each other into compositions. If you work in a Fortune 500 company, chances are someone else has already created exactly the solution logic you require; the challenge is to find it.

To be able to search for a service with any chance of success, one needs a 'service registry'. A 'service profile', which is a templated document containing everything there is to know about a service (in other words: the documentation), is stored in this service registry.

Erl has an example for a service profile template in his book (chapter 15, section 1). Some elements from this template are: purpose description, capabilities, QoS requirements, version, status and custodian. Each capability gets its own template, containing many of the same elements as the service template itself.

On a sidenote, Erl considers the "application of this principle by non-communicative resources" a risk [12, page 381]:

> *Often the definition of service contracts is left to the same team responsible for building the service itself. [...] this is generally desirable. However, while these individuals may be the most qualified, they may not be skilled communicators.*

In other words, programmers shouldn't be writing documentation, because they are probably not good at it. This can be solved by "subjecting [the discoverability meta information] to a review and revision by technical resources trained in the required communication skills".

3.4.8 Composability

Composability shows similarities with reusability, in that one tries to use a service on more than one occasion. However, since services are used in a hierarchy, a composition controller isn't necessarily reusable but does need to be composable. This is in contrast to a composition member, which needs to be both. Utility services are almost always composition members, seldomly controllers. This is inherent to them being on the bottom of the service hierarchy.

Whereas reusability focuses on being able to work with several different services, composability focuses on working with several services at the same time. Ideally the output of a service A can be streamed by another service B to a third service C, even though service A has no knowledge of, and no connection to, service C. Erl defines that a point-to-point exchange, meaning only two services are involved that communicate just with each other, is not a service composition. At least three services need to be involved.

For this kind of composition to emerge, one needs a solid service inventory. According to Erl, service inventories go through three stages before reaching such a state. These are (each stage is explained by its name):

1. Initial service delivery projects,
2. Hybrid applications and a growing service inventory, and
3. A service inventory is established.

One of the risks of service composition identified by Erl is that a composition member may become a performance bottleneck. He does not provide a solution; he just states it to create awareness. I address this point in the discussion (section 6.2).

4. Implementation of a 'Hough transformation' service

4.1 Introduction

To show that an algorithm can be computed using a massively parallel processor in a service-oriented architecture, I have taken the Hough transformation [10] as an example. The Hough transformation is part of a set of algorithms used in a prototype application within Logica. The application uses a camera and a robot arm. The camera sends images to the software, which tries to recognize a business card held in front of the camera, and then grasps the card using the robot arm. The image recognition is split up in multiple steps:

1. Apply the following filters:
   (a) Grayscale (color is unneccesary for the following steps)
   (b) Subtraction of the initial background (it is assumed the background is static, therefore it must not be processed)
   (c) High-pass filter (to create a binary image)
2. Detect blobs (pixels that are contiguous)
3. Find blob to process (select the largest blob)
4. Detect edges of blob (needed for Hough transform, as the transform is used to detect lines, not edges)
5. Perform Hough transformation (transform the image into a space graph with votes for the most prominent lines in the image)
6. Find hotspots (corresponding to lines) in Hough space graph
7. Find lines looking like a rectangle (i.e. a business card)

As the Hough transform is a very general algorithm and it takes most of the CPU time, it was decided to speed it up.

4.1.1 Hough transformation

The Hough technique transforms a binary image (an image with just black and white pixels) into a Hough space graph. For each data point in the image (shown in figure 4.1[1]), lines at arbitrary angles are drawn through them (for example

[1] `http://en.wikipedia.org/wiki/Hough_transform`, viewed on 23 october 2009.

for each degree on a 180 degree plane). The line perpendicular to the drawn line that intersects the origin is then measured for its angle and its radius. The corresponding point in the transformation (image on the right) is brightened by one unit. The brightest spots thus define the radius and angle of those lines through the origin. From them, the original lines in the image can be deduced (but not their location or length).

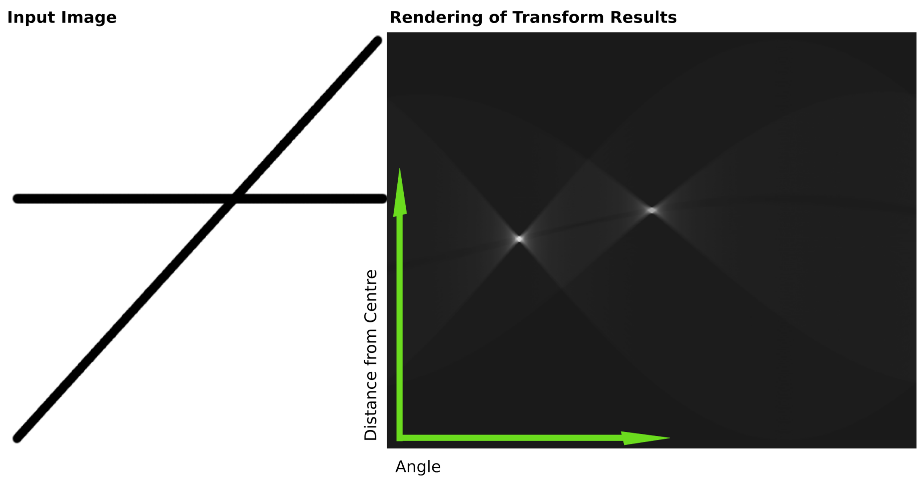


Fig. 4.1: An example of a Hough transformation applied to an image (left) and the rendering of the results (right). Image courtesy of Wikipedia.

The complexity of this algorithm is trivial: one draws all 180 (for example) lines through each pixel of the original image. For a 320x240 image, this yields 14 million identical operations. However, when processing High Definition images of 1920x1080 pixels, at 5 times the precision, thus drawing 900 lines through each pixel, then the number of operations jumps to 1.9 billion. The calculations are independent: they don't rely on other pixels or other angles. The memory access for storage is not entirely independent, as the same angle/radius combination can be accessed for multiple input image pixels.

4.2 Programming for the Tesla processor

As mentioned in chapter 2, I chose to work with the Nvidia Tesla processor. One cannot run software that has been written for an x86 processor on a Tesla processor, as the paradigm is entirely different. So, the part one wants to run on the Tesla has to be rebuilt.

There are several ways in which one can build software that runs on the Tesla: through CUDA libraries (in C/C++ or FORTRAN), by building ones own CUDA kernel (in C), or by using third-party Java and C# interfaces to the CUDA libraries and/or kernel.

4.2.1 Libraries

Nvidia supplies two libraries for use on the Cuda platform: 'CUBLAS' and 'CUFFT'. The libraries are self-contained: one does not need to write kernel code to take advantage of them. The way to use the libraries is best explained by the programming guide [30, page 1]:

> *The basic model by which applications use the CUBLAS library is to create matrix and vector objects in GPU memory space, fill them with data, call a sequence of CUBLAS functions, and, finally, upload the results from GPU memory space back to the host.*

Also, it is possible to combine kernel code and library code to achieve optimal results. The CUFFT library provides a simple interface for computing parallel Fast Fourier Transformations (FFT's) on the Tesla. Similarly, CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms), a standard for linear algebra computations such as matrix multiplication. For code examples, I will refer to the programming guides [30].

4.2.2 Kernel

If the functionality provided by the CUBLAS and CUFFT libraries is not sufficient (which is probably quite often the case), one can write their own custom programs for the Nvidia Tesla by creating a 'kernel'. A kernel is an ordinary C function, with the modifier `__global__`, like this:

```
__global__ void testKernel(float* g_idata, float* g_odata) {
    // code goes here
}
```

It is probably useful to specify some data structure or variable to get the input data from and to store the output data in (respectively `g_idata` and `g_odata` in this case). The function can then be called as follows:

```
testKernel<<< grid, threads, mem_size >>>(d_idata, d_odata);
```

The part between the angle brackets defines the *execution configuration*. The execution configuration starts the kernel on the device with the amount of blocks in `grid`, the amount of threads per block in `threads`, and the amount of memory available per block in `mem_size`. One block executes on one multiprocessor, and one grid executes on one CUDA device. If one would execute the function on a normal processor in a for-loop, it would be roughly equivalent to this:

```
int grid;
int threads;
for (int gridIdx = 0; gridIdx < grid; gridIdx++) {
    for (int threadIdx = 0; threadIdx < threads; threadIdx++ ) {
        testKernel(d_idata, d_odata);
    }
}
```

This method gives the programmer the freedom to do anything he or she wants. However, it is not trivial to rewrite algorithms to fit this paradigm, especially considering the hardware limitations (see section 4.2.4).

The kernel is the code that is run by the Tesla. Paralellism is created by the compiler which makes sure the kernel code runs once per thread. The thread itself must then decide what work to do. Usually this means a thread retrieves a specific part of the data set that is being processed, processes it, and stores the result (optionally somewhere else). A simple example is listed as follows:

```
__global__ void add(int *data, int delta)
{
        const int index = threadIdx.x + (blockIdx.x * blockDim.x);
        data[index] += delta;
}
```

`threadIdx.x` is the id (within the block) of the thread that is currently running. `blockIdx.x` is the id of the block that is executed, and `blockDim.x` is the total number of blocks that will be executed. Thus, to get a unique identifier (`index`) for the current thread, those variables are combined as shown. The identifier is then used to retrieve a data element from an array. This data element is not used by any other thread, therefore it is safe to modify the data element (in this case add a value to it).

4.2.3 Java / C# interface

The Israeli company GASS (which stands for GPU based Advanced Supercomputing Solutions) created the jCUDA and CUDA.NET libraries, for use with the respective languages. jCUDA employs JNI (Java Native Interface) to interface with both the CUDA libraries and the CUDA kernel system. It is a rudimentary interface which mimicks the behaviour of the C procedures used to run a kernel. jCUDA can be retrieved from the GASS website[2], along with its documentation and examples.

Because the jCUDA API is so basic, I created a wrapper class to deal with the specifics of loading a kernel function, setting its parameters and launching the kernel. Using the wrapper, the example kernel given earlier can be run as follows:

```
// initialize data
int[] array = {1,2,3,4,5,6,7,8};
int delta = 5;

// initialize device
CUDAWrapper cuda = new CUDAWrapper(new File("test.cubin"));
cuda.getModuleFunction("add");

// set parameters
CUdeviceptr arrayDevicePtr = cuda.setParameter(array);
cuda.setParameter(delta);

// launch!
cuda.launch(2, 1, 4, 1, 1);

// retrieve the result
array = cuda.getParameterIntArr(arrayDevicePtr);
```

The kernel still has to be programmed in C, but since this is only a small part of the entire application, this is less of a problem. The kernel then has to be compiled (using Nvidia's CUDA compiler, *nvcc*) into a library which is dynamically linked.

A quick glance at the documentation of CUDA.NET reveals that it works very similar to jCUDA. I have not looked at in detail, as it was not directly relevant for my research.

4.2.4 Hardware limitations

The CUDA architecture has its limitations. Some are resolved by Nvidia with new hardware models, some are not. The hardware models are defined by their

[2] The URL of jCUDA is `http://www.gass-ltd.co.il/en/products/jcuda/`.

'compute capability', currently ranging from version 1.0 (which is supported by the Tesla C870 I'm using) to 1.3 (for the newest boards). I will list the limitations I have encountered during development:

- The Tesla I'm using is not capable of performing atomic operations; if two threads write to the same memory location, one succeeds, but it is not known which one. Newer versions do support this.
- Section 2.5.3 already remarks the fact that the C870 board has 16 multiprocessors. This is important for the software, because there must be at least as many blocks as there are multiprocessors. Otherwise, some of the multiprocessors will be doing nothing. Likewise for the number of threads: they should be a multiple of 32 to prevent parts of a multiprocessor idling, or even a multiple of 64 to allow more freedom for the scheduler.
- Debugging is hard. There is an emulator which allows you to step through the code in a normal debugger. The issue with the emulator is that it does not differentiate between pointers to host memory and to device memory, which was a big source of errors in my code. Debugging therefore happens using old-fashioned temporary variables and arrays.
- Support for 64-bit floating point arithmetic (e.g. using the `double` datatype) hasn't been added until compute capability 1.3.
- The sine and cosine operators (`sin()` and `cos()`) operate with a deviation of 2 *ulps* (unit of least precision). This means that for the `float` datatype, inaccuracies of about 10^{-8} occur. This isn't much, but it makes it impossible to compare end results with those calculated by Java, which operates with a 0.5 ulp margin. This problem is described in more detail in section 4.3.2.
- There are two kinds of memory on a Tesla: global memory and shared memory. The latter is way faster, but it only exists during the execution of a block, meaning you need to keep it synchronized with global memory.

4.3 Implementation

As stated earlier, I will focus on programming the Hough transformation algorithm for the Tesla processor. The original business card recognition software is built in the programming language C#.NET, using the open source AForge.NET library.

> AForge.NET is a C# framework designed for developers and researchers in the fields of Computer Vision and Artificial Intelligence - image processing, neural networks, genetic algorithms, machine learning, robotics, etc.[3]

The Hough transformation is implemented in AForge.NET in the `HoughLineTransformation` class of the `AForge.Imaging` namespace. It is called Hough *line* transformation because the general Hough transformation can be

[3] As found on the homepage of AForge.NET: `http://www.aforgenet.com/framework/`.

used to detect any shape that can be defined by a limited number of parameters; AForge.NET also supplies `HoughCircleTransformation` which, obviously, detects circles.

4.3.1 First implementation in C

After building a few trial programs (listed in appendix 7.1), I began my first attempt at getting the Hough transformation algorithm to work in C. I did not succeed at this, due to various problems. The problems I encountered fall into two categories: the first are due to inadequate documentation and errors that are tough to solve without documentation, and the second are due to my lack of experience in the C programming language. Or sometimes both.

- First of all, at the start of my research, the only documentation available was a programming guide, a reference manual, some examples and a not very active forum. This made it very hard to find solutions to error messages. 'Howto's' can be immensely helpful, but they also were nowhere to be found. Logica had no experts on CUDA programming, as I was the first to use the technology.

- To just get the examples to work meant wading through compiler and runtime errors which were caused by:

  – the absence of third-party libraries (like GLUT, the OpenGL Utility Toolkit),

  – the wrong naming of shared libraries belonging to the Nvidia graphics driver, which also interacts with the Tesla,

  – incorrect file paths (shared libraries cannot be found by the compiler or at runtime),

  – the specific version of the GNU C compiler rejects certain syntax, therefore necessitating the use of an older version of the compiler,

  – the specific version of the graphics driver being incompatible with the CUDA libraries,

  – too strict settings of Security-Enhanced Linux (which is standard in Fedora Linux and should be supported according to the documentation).

- After I got the examples running and I started writing my own software, I ran into difficulty debugging C programs. The Eclipse debugger didn't work with the CUDA software development kit out of the box, and it was hard to figure out why. The only way for me to debug was by `printf`-ing variables to the console, which is a hassle. After a while I got DDD (Display Data Debugger) to work, which is still rather primitive. And even later a howto was published on the internet describing how to get CUDA to work in Eclipse[4], which solved this problem.

[4] *Quickstart for CUDA & Eclipse CDT*, from the Nvidia forums: `http://forums.nvidia.com/index.php?showtopic=71535`, viewed on 23 october 2009.

- Programming your own kernel also means doing your own memory management. In C this works with pointers, and there is no difference between pointers to host memory and pointers to device memory. At least not as far as the compiler is concerned. Errors of this kind are very hard to debug, as the emulator will run your application perfectly fine. The emulator converts all device pointers to host pointers, since all code runs on the host in emulation mode. The matter was further complicated by the fact that I was trying to send a `struct` containing pointers to the device; one needs to be very careful with allocating memory, copying data and assigning pointers when using a struct within CUDA. Eventually I gave up on the use of structs, and just used arrays (which is less elegant).

- The Nvidia compiler does not support the use of multidimensional arrays. Since the Hough transformation deals with images, I needed to calculate indexes into arrays representing images myself. This very easily leads to confusion and hard to find bugs (because of mixing up x and y coordinates and image width and height). The fact that image formats and the AForge.NET code do not use the same coordinate system made things even more complex.

In March of 2009, jCUDA was made available. Since I have more experience in Java than in C, I switched to Java development.

4.3.2 Second implementation in Java

Getting jCUDA to work was relatively uncomplicated, although I'm sure my previous experiences in C have helped. I have taken the Hough transformation algorithm and ported it to Java. The prototype software should be adapted to send the image to be analysed for the Hough transformation to the service, on which the transformation is done using the Nvidia Tesla. The resulting graph is then transferred back to the application to be processed further.

The application has just one interface to the outside: it is a function called `transform` with two parameters: `image` (in the case of Java, of type `BufferedImage` and `precision` of type `int`. The Java code of the specific method is displayed in listing 4.1 on page 52.

I'll walk through the code line by line to explain what is happening.

1-3 Class and method declaration.

4 A check is performed to see if the supplied image is a 256-color image and if a valid precision is given.

6-15 The parameters needed for the calculation are calculated and put in an array, as this is easier to send to the Tesla than sending seven seperate integers.

17 The CUDA system is loaded, selecting the appropriate library file (“`hough.cubin`”).

18 The necessary function (“`transform`”) from the library is chosen. There could be multiple functions in a library, therefore the function has to be specified.

20 The Java image is converted to a simple array and sent to the Tesla. The method `setParameter` takes care of allocating the correct amount of memory on the Tesla, copying the data to the device, and sending the pointer to the function.

21-22 A new array initialized with zeroes is sent to the Tesla. It uses the same `setParameter` method as on the previous line. This time the return value is stored (in `houghMapDevicePtr`); it is a pointer to the array in device memory. We need to store this to be able to retrieve the contents of the memory later. This wasn't necessary before as that was input data.

23 The last data to be sent to the Tesla are the calculation parameters.

25 This method launches the kernel. The program code is loaded into the memory of the Tesla and execution starts. This is actually an asynchronous operation (meaning the execution of the Java application will continue while the kernel is running), however there is code inside the `launch()` method which synchronizes the application with the kernel, as in this application there is nothing for the CPU to be done in the meantime.
The first two parameters define the block size, and the last three define the grid size. I only use one-dimensional grids and blocks in this case, therefore the second, fourth and fifth parameters are of the value `1`. The number of blocks should be maximized, and the number of threads should be at least 32. Therefore I divided the number of degrees by 4, resulting in 45 threads being executed. The precision is multiplied by 4, maintaining the 180 degrees to calculate.

27 When the launch has succeeded, the resulting data (the 'Hough map') is copied from the Tesla back to the host computer to be processed further.

28-30 The array is 'scaled' first; this means the largest value is decreased to 255, and all other values in the array are decreased by the same magnitude. This is done to be able to convert the array to a grayscale image, which only has 256 possible values ('`0` is completely black). Then the array is converted to a `BufferedImage` and returned to the caller of the method.

Implementation notes

I compared the result of the Hough transformation on the Tesla with the result of the same calculation on an x86 cpu, and found there were small differences. These differences occur where the calculated `radius` is very close to an integer number, e.g. a value of 0.9999999. The result is a `float`, which is casted to an `int`. In C, a type cast to an integer means the floating point number is truncated; in this case 0.9999999 is 'rounded' to 0. However, the same radius calculation on the Tesla yields 1.0000000, and when this value is truncated, the result is 1.

Using the `round()` function instead of truncating via a type cast does not solve this problem; it merely moves the problem to the values 0.4999999 and 0.5000000. After digging through the documentation, it appears that the Tesla has a larger margin for error than the x86 architecture. The documentation states that the Tesla complies with the IEEE-754 standard (a standard for single-precision binary floating-point arithmetic), although its behaviour deviates from what the x86 architecture does. The exact difference lies in the fact that the computation of a sine, cosine and division on the CUDA architecture have a 'maximum ulp error' of 2, whereas those computations on the x86 architecture have a maximum ulp error of 0.5.

'Ulp' stands for *unit in the last place*, and specifies the smallest gap between two values stored in a binary floating point format. Since a single precision float can store only 7 significant digits, this gap ('1 ulp') is 0.0000001 in the aforementioned cases. This means the Tesla can have a discrepancy of 0.0000002, which is enough to cause the issues described earlier. I have not found a way to emulate this behaviour in the Java environment. Another solution would be using double precision floating point numbers, except that 'doubles' are not supported in compute capability 1.0 (also, using doubles would incur a severe performance penalty).

After visualization of the Hough space graph, visual inspection revealed no significant differences between the graphs generated on CUDA and graphs generated by an x86 processor.

4.3.3 Kernel

The kernel, which must be programmed in the Nvidia adaptation of the C language, is shown in listing 4.2.

Again, I'll do a walkthrough of the code.

1 The function declaration. `extern ''C''` is used because the code will be dynamically linked. `__global__` was explained in section 4.2.2. The parameters are all arrays and speak for themselves.

2-8 The calculation parameters which were sent to the Tesla are extracted from the array. This is done for code readability, but it has an additional benefit: the array resides in 'global' memory, whereas the parameters are now transferred to 'shared' memory. Shared memory is orders of magnitude faster than global memory, so the algorithm should execute faster.

9 Another calculation parameter is computed from the previous parameters.

11 The `theta` variable is determined from the id's of the thread and the block. Since each thread executes the algorithm with its own theta value, the possibility of concurrent memory writes is eliminated.

13-14 The thread loops through all pixels of the image.

15 It is determined whether the current pixel is non-black (i.e. white, since the input is a binary image). If that is the case, the Hough space graph must be adjusted.

16-18 As explained at the beginning of this chapter (section 4.1.1), for each pixel different lines are drawn through it. The angle of the line is determined by `theta * thetaStep`. Then the distance of the line to the origin is calculated; this distance (in pixels) is stored in `radius`.

Listing 4.1: The Java method that calls the Hough transformation on the CUDA architecture

```java
public class HoughTransformation {
  public static BufferedImage transform(
                 BufferedImage image, int precision) {
    checkInput(image, precision);

    int[] params = new int[7];
    int width        = params[0] = image.getWidth();
    int height       = params[1] = image.getHeight();
    int halfWidth    = params[2] = width / 2;
    int halfHeight   = params[3] = height / 2;
    int halfHoughWidth  = params[4] = (int)Math.sqrt(
            halfWidth * halfWidth + halfHeight * halfHeight);
    int houghWidth   = params[5] = halfHoughWidth * 2;
    final int DEGREES = 180;
    int houghHeight  = params[6] = DEGREES * precision;

    CUDAWrapper cuda = new CUDAWrapper(new File("hough.cubin"));
    cuda.getModuleFunction("transform");

    cuda.setParameter(imageToArray(image));
    CUdeviceptr houghMapDevicePtr =
      cuda.setParameter(new int[houghWidth * houghHeight]);
    cuda.setParameter(params);

    cuda.launchGrid(precision * 4, 1, DEGREES / 4, 1, 1);

    int[] houghMap = cuda.getParameterIntArr(houghMapDevicePtr);

    return arrayToImage(houghWidth, houghHeight, scaleArray(houghMap));
  }
}
```

Listing 4.2: The CUDA kernel that performs the Hough transformation, written in C

```c
extern "C" __global__ void transform(char *in, int *out, int *params) {
  int width          = params[0];
  int height         = params[1];
  int halfWidth      = params[2];
  int halfHeight     = params[3];
  int halfHoughWidth  = params[4];
  int houghWidth     = params[5];
  int houghHeight    = params[6];
  float thetaStep = 3.141592653589793f / houghHeight;

  const unsigned int theta = threadIdx.x + (blockIdx.x * blockDim.x);

  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      if (in[x + y * width] != 0) {
        int radius = (int)(cos(theta * thetaStep) *
              (x-halfWidth) - sin(theta * thetaStep) *
              (y-halfHeight)) + halfHoughWidth;

          if ((radius < 0) || (radius >= houghWidth)) {
            continue;
          }

          out[radius + theta * houghWidth]++;
      }
    }
  }
}
```

20-22 The radius is checked to see if it falls within the bounds of the Hough space graph (although it should never be outside those bounds).

24 The ‘bucket’ in the Hough space graph that corresponds to the current theta and radius is incremented by one.

4.4 Benchmarks

4.4.1 Hardware setup

The computer on which the benchmarks were run consists of:

- Intel Pentium 4 HT 2.8E processor
- 2x 1 GB dual channel DDR400 memory
- Intel D915GAG motherboard
- Nvidia Tesla C870 board (G80 processor with 1.5GB memory)

I emphasize that the Pentium was released in Februari 2004, whereas the Tesla was released in November 2007. To compensate for this fact, all measurements are adjusted to the time an Intel Core 2 Extreme QX6700 would have taken to execute the application (this processor was also released in November 2007 and had the same initial price of 1000 USD). The adjustment is calculated as follows: according to Intel, the Pentium 4 HT 2.8E can perform 5.6 GFLOPS and the Core 2 Extreme QX6700 performs 45,6 GFLOPS. This is a difference of a factor $45.6/5.6 = 8.14$, so all CPU measurements are divided by this number. My method does not take other differences into account like the memory speed and bus speed. However, the Tesla is also slowed down a bit by the slow host (the transfer bandwidth decreases by a factor 3). Lacking a faster test PC, the difference could not be quantified. It is expected that it is not significant.

4.4.2 Software setup

The execution time of the `transform` method is measured by calling the Java function `System.getCurrentTimeMillis()` twice, and calculating the difference. On the Linux 2.6 kernel this is a fairly reliable method of measuring execution time; the only discrepancies expected are due to the Java Virtual Machine interrupting the process for either the garbage collector or the hot spot compiler. It is worth mentioning that I measure not only the execution time on the Tesla (like Nvidia does in the examples supplied with CUDA), but that the overhead of copying to and from Tesla memory is included. This will of course have an adverse affect on the measured times, but this more accurately reflects real-world performance. Furthermore, the code of both the CPU and CUDA versions are not optimized. The sine and cosine instructions could have been put in a lookup table. This optimization is not performed by the compiler.

This is the software I am running on the test PC:

- Fedora Linux 9 with Linux kernel 2.6.27
- Nvidia graphics driver 180.06

- OpenJDK 1.6.0
- CUDA Toolkit 2.1, build 1635

I have benchmarked the software with four kinds of input images:

1. a completely black image (the radius calculation is skipped every time),
2. an image of a business card of which the edges have been detected,
3. an image of a business card which was held between the thumb and other fingers (causing the shape of the thumb to show up after the edge detection, shown in figure 4.2), and
4. a completely white image (causing a radius calculation on each pixel, making this transformation the most lengthy process).

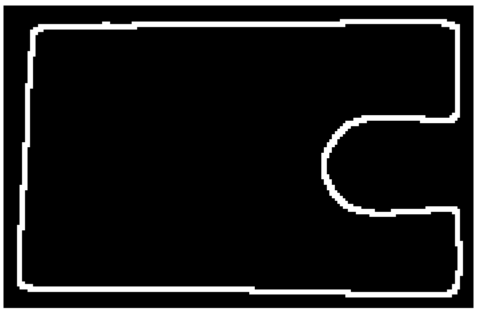


Fig. 4.2: An edge-detected business card, distorted by a thumb holding it.

Each of these images are used in two resolutions: one in the original resolution, as supplied by the prototype robot arm software used at Logica (176x113 pixels), and one in 'HD' (High Definition) resolution, which is used for high quality video (1920x1080 pixels). The high resolution image is a resized version of the original ones. The HD image is mainly used to illustrate the power of the Tesla processor, as it was found that the detection of lines did not improve with higher resolution images. The amount of white pixels in the business card images is 6.5% of the total number of pixels for both resolutions.

Also, the images were processed with different value for the precision, namely the values 1, 2 and 4. The distance between these values exists to show statistically significant differences.

4.4.3 Measurements

All tests were run five times. The highest and lowest execution times were discarded, after which the average was taken of the remaining three data points. The resulting values have an accuracy of +/- 10 milliseconds. The x86 values are adjusted to correct for the x86 processor speed, as described in section 4.4.1. This led to the measurements shown in table 4.1.

From the results, some observations can be made.

Tab. 4.1: Average measured times across five tests, in milliseconds

Precision	1		2		4	
Image	x86	CUDA	x86	CUDA	x86	CUDA
Original image						
Completely black	2	153	3	159	1	166
Normal card	12	162	27	162	44	160
Card with thumb	13	144	30	161	55	197
Completely white	204	205	404	195	758	198
High definition						
Completely black	15	1679	12	1549	14	1702
Normal card	1069	1668	2136	1724	4273	2020
Card with thumb	1266	1741	2541	1795	5077	2056
Completely white	19679	5621	39607	5846	78765	7763

- The Tesla is only faster when there is a big number of white pixels; it is slower otherwise. This is due to the fact that the number of white-pixel-tests (see line 15 in listing 4.2) is vastly larger for the Tesla. Whereas the CPU code does not compute any angles when a pixel is black, the GPU kernel has to check all pixels for every angle.

- For each increase in precision, the x86 processor shows a proportional increase in execution time. The Tesla however, shows roughly equal execution times, irrespective of the precision! This is caused by the Tesla not being able to harness its full power. It has 16 multiprocessors, of which only 4 are used with the precision of 1, 8 are used with the precision of 2, and all 16 are not used until the precision reaches 4.

- Even for very small data sets, the Tesla takes some 160ms. This is most likely due to the overhead associated with copying data to the Tesla, starting a kernel, and copying the data back to the host.

4.5 Discussion

I have shown that it is possible to build a Hough transformation service using the CUDA architecture, and that this service can be substantially faster than the equivalent x86-based service.

4.5.1 Benchmark results

The performance results are not as spectacular as Nvidia leads one to believe. This can be partially attributed to the fact that I measured not only the time it took a kernel to execute, but also included the initialization of Tesla memory. As mentioned, this reflects real-world usage. Furthermore, the kernel is not as optimized as it could have been. However, to accomplish full optimization, a serious amount of profiling and analysis needs to be done. Depending on the application, this might not always happen. Parallelizing the work over different pixels instead of different degrees might show a performance increase. I was unable to build a kernel showing this.

One other factor is the applicability of Amdahl’s Law[3] in this case. It states that the speedup of a parallel program is limited by the sequential part; if the sequential part of a calculation takes 10% of the time on a single processor, no matter the amount of processors, the speedup will be maximized at 10 times the original speed. In my application, the bitmap has to be transferred to the Tesla. This operation is sequential and it will slow the Tesla down.

4.5.2 Suggestions to Nvidia

As mentioned numerous times throughout this chapter, programming in the CUDA architecture isn’t as easy as one would think, or as easy as I expected in the beginning. To help out new developers, I highly recommend Nvidia builds the following:

- A tutorial ‘programming for the CUDA architecture’. Currently there exists only a tutorial to get to an ‘Hello world’ application; the rest of the documentation consists solely of examples. A tutorial showing how to build[5] one or more of the examples would be really helpful.
- A tutorial ‘setting up a development environment for the CUDA architecture’ including what to do if it doesn’t work would be a tremendous help.
- A small library of often-used functions with defaults. For example: copying an array to the Tesla usually involves both a memory allocation operation and a memory copy operation, so these could be combined. Or even better, just make it possible to send a host array to the device function, and make the conversion automatic! Every developer encounters these same instructions; one could keep the existing instructions for added flexibility. I already did this in my extension of the jCUDA library (which, unfortunately, is not maintained by Nvidia but by a third party).

[5] ‘Build’ as in ‘program the code’, not as in ‘compile’.

5. Selecting a software effort estimation method

5.1 Introduction

An implementation of a service for massively parallel computing has an impact on the effort of development of new software that uses massively parallel computing. It is useful to quantify this impact, i.e. find out how much it will cost to build software on top of the designed service, compared to how much it will cost to build software directly for the architecture.

Software effort estimation focuses on the amount of time (the effort) it takes to complete a software project. Software effort estimation is hard, as shown by Moløkken and Jørgensen [28]. 60-80% of software projects exceed their allocated time and/or budget. To increase the estimation accuracy (the deviation of the actual effort from the original estimate), many estimation approaches have been developed, with varying success. Also, I have taken a look at the methods used by Logica, to be able to use criteria from the real world. I used these criteria to compare methods from literature and to determine what is the best method.

In the next section, I show and explain the conceptual model used for this research. The third section explains the methods described in literature. The methods used by Logica are detailed in the fourth section. The final section gives recommendations on best practices and methods.

5.2 Conceptual model

To determine which estimation methods there are, a literature study is carried out. The basis for this literature study is research performed in 2007 by Jørgensen et al., who have catalogued all publicly available, peer reviewed research papers on cost estimation in their article *A Systematic Review of Software Development Cost Estimation Studies* [25]. They have also classified estimation approaches, enabling a focused research into each of the approaches while knowing all approaches have been covered. There are others who tried to list the different methods, but Jørgensen's research was by far the most comprehensive [6, 7].

The logical question that follows after what approaches there are, is what the best approach is. The most important criterium is of course accuracy. But other factors cannot be ignored: the amount of work involved for arriving at an estimate, the data needed for a specific approach (estimation by analogy needs data from other projects to compare the current project to, which may not be available) and possible other uses for the collected data (function point analysis for example allows for defect estimation). I will compare the methods to each other using criteria which are important to Logica.

There are two sources to gather information on software estimation within Logica. These sources are the documentation about the estimation processes and the people actually performing and using the estimates. The documentation is available within Logica as the Estimating Framework (EFW). People performing and using estimations are bid managers, project managers, delivery managers and those in charge of choosing an estimation method. Since the approaches mentioned by the documentation and employees might differ, it is helpful to consult them both.

The available documentation needs to be studied and mapped to the approach categories as defined by Jørgensen. To get the information from employees, I conduct interviews with four key people, being two project managers (of which one also conducted bids), a delivery manager and the 'lead expert estimating'. During the interviews I made notes which I have processed and combined in this text. A point of interest is also whether any accuracy statistics are available and what they are.

The conceptual model shown in figure 5.1 is applied to the gathered information, resulting in the recommendation explained in the last section.

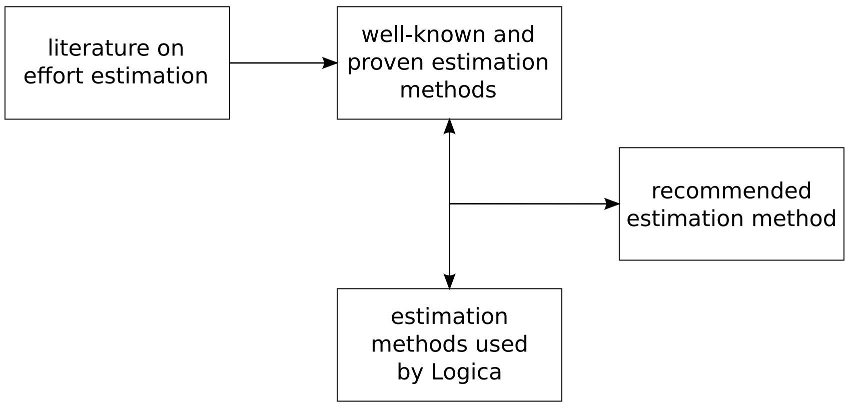

literature on effort estimation
well-known and proven estimation methods
recommended estimation method
estimation methods used by Logica

Fig. 5.1: The conceptual model for the recommendation on effort estimation methods

5.3 Estimation goals

First it is important to establish a strict terminology concerning effort estimation goals. There is no real widespread accepted use, as explained by Grimstad, which often leads to confusion [19, page 303]:

> *Lack of precision leads easily to a mix of processes with different purposes, e.g. a mix of processes with a focus on realism (estimation of most likely effort), a focus on efficient development work (estimation of planned effort), a focus on avoidance of budget overrun (decisions on budgeted effort), and a focus on winning a bid (estimation of price-to-win).*

Therefore, I will follow his guidelines regarding terminology of estimation goals. These goals refer to the different processes that produce the estimation (as mentioned on the previous page), often leading to a different estimate. The goals are:

actual effort The effort it actually took to build the software.

most likely effort The estimate that has the highest probability of becoming the actual effort.

planned effort Most likely effort with a margin to make it possible to work efficiently. For example, this takes the availability of developers into account, as that may change during a project, while it impacts the end date.

budgeted effort Most likely effort with a margin to account for project risks. If an error in the software pops up late in the project, the project can be severely delayed. This can be taken into account when estimating the project, even though the probability of a long delay happening may be low.

price-to-win Effort based on the price needed to make a sale to a customer. The ‘estimate’ is made backwards: a price is determined, after which it is decided how much effort can be spent for that price.

Finally, the *estimation accuracy* is the difference between the actual effort and the estimate. This can be measured in different ways, of which MRE (“mean of magnitude of relative error”) is the most popular [26]. According to MRE, if a project has a most likely effort of 10 months, but it took 11 months to complete, it was overrun by 10%. A large MRE does not mean that the estimator did a poor job, but merely that *this* estimate was incorrect. This can for example be caused by the complexity of the project or by insufficient project management [18]. Other methods of expressing estimation accuracy have been proposed [26], but whether they are superior remains to be seen.

To be able to compare estimation accuracy across projects, the same estimation goal must be used for each project [19]. Also, to compare the actual effort to one of the goals, it needs to be adjusted for delivered work. If the most likely effort includes a solution for a certain requirement, but that particular solution is not built, the actual effort cannot be directly compared to the most likely effort. Adjustments to the estimation of most likely effort are thus needed.

5.4 Estimation approaches

Because the cost of software is mainly dependent on the development effort, a lot of work has been put in devising an estimate of that effort. An estimation approach is a way to go from “this is the project” to “this is the effort the project is going to take”. ‘Effort’ usually means the amount of time, which translates into a certain cost. This can be the cost to the organization; it can also be the cost to the customer (in which case the effort translates into a price).

The different estimation approaches are described and categorized by Jørgensen. However, he states that his classification “was developed for the purpose of our review and is not intended to be a general-purpose classification of software effort estimation studies” [25, page 34]. I have taken his classification and recategorized them to better fit my research.

5.4.1 Expert judgment

Estimation based on expert judgment relies solely on the opinion of one or more experts. A structured approach to arrive at the estimate is not necessary. This method relies heavily on the experience and skill of the estimator. To mitigate the impact of subjectivity one can use the Delphi technique (a technique to reach consensus between multiple experts [23]). To aid experts in arriving at a most likely estimate, PERT may be used. PERT stands for Program Evaluation and Review Technique and averages multiple estimates (optimistic, pessimistic and most likely) with a weight of 4 for the most likely estimate and 1 for the others [8, pages 19–27]. This has been shown to provide a more accurate estimate.

The advantage of expert judgment based estimation is that no historic data is needed. The disadvantage is that when an experienced expert for the particular software project is not available, the estimation accuracy may be very low. Also, this method has a bias towards underestimation [24].

5.4.2 Analogy

Analogy-based estimation (and, closely related, case-based reasoning) utilizes the data of previously completed projects. The goal is to find projects that are at least partly similar in scope, staffing, etc. to the project to be estimated (hence 'analogy-based'). The actual effort of the found project is then used as the estimate for the new project, after modification to make sure both projects are comparable [14]. It is also possible to compare to a set of similar projects, increasing the accuracy of the estimate.

The advantage of the analogy-based method is that estimates have their direct basis in reality without any models, which makes it simple and fast. The disadvantage is that previously completed projects that are analogous to the current project are not always available [24].

5.4.3 Work breakdown

The work breakdown structure (WBS) decomposes the project into a subtask hierarchy. These subtasks can also consist of other subtasks, until a level has been reached where no more subtasks can be or need to be defined [36]. WBS is not a full estimation method by itself, but needs to be combined with another method like expert judgment or analogy-based estimation. The subtasks are estimated one by one, and lastly all estimates are aggregated into the final estimate (this is also called a bottom-up approach).

Because the work breakdown structure divides complex tasks into elementary subtasks, the complex tasks become less fuzzy allowing for more accurate estimates [24], which is its greatest advantage. The disadvantage is that the project must already be reasonably defined to be able to use this approach.

5.4.4 Function points

Estimation based on function point analysis (FPA) defines the software size in function points. These points are then multiplied by the expected time per function point, yielding a final estimate of the effort [15]. One could of course multiply the function points by a cost per function point, immediately yielding a price.

Function point analysis first establishes the size of the software in 'unadjusted function points'. In a nutshell, points are assigned based on the number of inputs and outputs, to be determined from the requirements specification. The number of points to assign have been determined by Albrecht, the inventor of FPA [1]. The unadjusted function points are then corrected for 14 complexity factors (called 'general system characteristics') of a project. Some examples of these factors are performance, usability and distributed data processing. As stated above, the final number of function points is then converted to a final estimate of the effort.

Advantages of this model are that function points have been defined in ISO/IEC standard 20926:2003, allowing for a repeatable estimation process. Function points are also used for estimates of other variables (than development effort), including the expected number of defects and the testing effort. The disadvantage is that, due to the dependency on inputs and outputs, function points are especially suited for administrative systems and not so much for technical (embedded) systems with highly complex algorithms.

5.4.5 Mathematical model

The idea of a mathematical model is that, for a set of cases, input variables and actual effort are measured. Then the effort is modeled as a function of the variables. For a new project, the estimator estimates or determines those variables and enters them into the model. An effort estimate will then be computed [14].

Mathematical models are not explicitly named in Jørgensen's paper. However, the following of his categories are essentially mathematical models: regression-based estimation, CART (classification and regression trees), simulations, neural networks, theory-derived estimation models, the Bayesian estimation model and "other techniques for estimation modeling". They differ only in the algorithm that establishes the model and in the specific input variables; the principle for the estimator remains the same. I will not go into detail on the specifics of these algorithms, but instead refer to Jørgensen's work [25]. A notable example of a method based on a mathematical model is COCOMO (COnstructive COst MOdel).

When constructing a mathematical model, the training data is very important. 'Training data' is the information that is fed into the model, i.e. the input variables and the actual effort of previous projects. The higher the similarity between the training data and the projects the model will be used for, the more accurate its results will be.

The advantage to using mathematical models is that a modification to the input variables immediately leads to a new estimate, thereby enabling the estimator to assess the impact of such changes. Examples of such variations are the level of experience of team members, whether a project planning tool is used, how many people are involved, etcetera. Also (depending on the input variables), it can be used before requirements are established.

Disadvantages are that it can take a lot of effort to calibrate a mathematical model for a specific software team or company. The assessment of input variables is subjective (however this also holds for expert judgment). Because the model has to be tailored for a specific situation or company, it is necessary that data of previous projects is available (although this data may be taken from different companies). Finally, some of the models have little empirical research to show their strengths and weaknesses.

5.4.6 Combination of methods

Because different estimation techniques achieve different accuracy results on different projects, it is useful to combine estimation methods. [27]. An example might be a work breakdown structure, in which a subtask is transformed into function points which is then used in a simulation [14]. Or, a combination of methods that is more common, expert estimation of subtasks of a work breakdown structure. As noted in the description of the work breakdown structure, it always needs to be combined with another method. The advantages and disadvantages of combinations differ with the respective combination. A 'best of both worlds' situation should be strived for, where the advantages of one method cancel out the disadvantages of another.

5.5 Case study on estimation methods

As mentioned before, I have conducted a case study on estimation methods used in a real business (in contrast to methods found in literature). Here also I made a distinction between estimation goals and estimation approaches.

5.5.1 Estimation goal

Within Logica, no official distinction is made between estimation goals. That being said, there are still some observations to be made:

Price-to-win is never used. An effort estimation, not a cost estimation, is made by a 'bid manager' (a manager which creates offers for potential customers). Once this estimation is done, the result is multiplied by an hour-rate which yields a total price. If this price does not suit the customer, deviations from the hourly rate are possible, but deviations from the estimate are not. This ensures that effort estimates are never compromised to win a bid, which is also recommended by Grimstad [19, page 308]. Compromised effort estimates lead to problems, because they don't reflect reality.

Actual effort is very precisely recorded for billing and project management purposes. Regarding estimation accuracy, it is only used on a per-project basis to determine the profit margin. It is not (yet) used to increase the estimation accuracy by using previous estimates for new projects.

There is no distinction between the other estimation goals. However, it is reasonable to assume that *planned effort* is used for variable-price projects (because an accurate end date is important) and *budgeted effort* is used for fixed-price projects (as more risks need to be taken into account, to prevent cost overruns). There are efforts (explained in the next paragraph) to make a clearer distinction between best-case, most likely, and worst-case estimates, which can then be used to more precisely calculate most-likely, planned and budgeted efforts.

5.5.2 Estimation approach

From the interviews I deduct that important criteria for Logica when choosing software effort estimation methods are:

- estimation accuracy (of course),
- simplicity (to lower the resistance of estimators to use a certain method),
- availability of estimates early in the process (for pricing purposes, while allowing for increasing accuracy later on),
- usability on a variety of systems and processes (not only software, but also consultancy projects).

At Logica, multiple software effort estimation methods are in use (as of 2008). These are 'unstructured expert-based', 'spreadsheet-based', 'work breakdown structure-based', 'stereotypes-based' and 'function point-based'. I will describe them here and explain to which method in literature they map.

- *Unstructured expert-based* estimation is being used by some bid managers throughout Logica and grew like this historically. It is essentially just expert judgment, but is called 'unstructured' to indicate the unsystematic nature. Due to the lack of structure and therefore lack of repeatability, this method is currently being phased out.
- *'Spreadsheet-based'* estimation is a method that employs self-made spreadsheets (by the estimator) to aid in the estimation process. These spreadsheets were seldomly created with awareness of existing estimation methods. No details are available on these methods, but they are probably similar to expert judgment, work breakdown, mathematical model and combination approaches. They have grown from the unstructured expert-based estimation method, and are also currently being phased out.
- The most used method is *work breakdown structure-based* estimation where subtasks are estimated using expert judgment. This method is the preferred method and efforts are made to standardize this method across Logica. This is also the method that is documented in the Estimating Framework. Its origins can be found in the company Admiral which was acquired by Logica (then CMG) in 2000. The method was subsequently engineered by Logica. 'PERT' (described under the expert judgment method on page 60) in addition to WBS is used in parts of the business unit Technical Software Engineering. This will be added to the estimation method in a later stage.
- Estimation using *'stereotypes'*, essentially analogy-based estimation is an addition to the previous method, where components of the software to be built are similar to components that have been built before (the stereotypes). The estimation data of that previous project (including the actual effort) will then be used to enhance the estimate of the current project. This method is in the pipeline to be added to the WBS when WBS is satisfactorily implemented and there is sufficient data collected to be actually implementable. This method was also inherited from Admiral.

- *Function point-based* estimation is used as a secondary estimation approach, usually combined with the WBS method. It is mainly applicable to administrative systems, which means that it is not used in all business units within Logica. Primarily customers of the business unit Finance request this method be used, because it also allows for the estimation of the number of defects (for more information see the description of function point-based estimation on page 60). This also explains why Logica uses it.

Summarizing, the estimation method that Logica intends to use in the future is work breakdown structure-based estimation where subtasks are estimated using a combination of expert judgment and analogy. Depending on the project function points might also be used.

5.6 Discussion

Given the requirements of Logica, the expert judgment and analogy-based estimation approaches fit very well. Expert judgment has shown to be fairly accurate, provided that the estimator is familiar with the environment. It also matches up to the other requirements, as it is intuitive, available from the beginning of a project (albeit roughly), and applicable to any kind of project. Because there is plenty of information available on previous projects, analogy-based estimation is feasible for the same reasons as expert judgment. The combination of analogy and expert judgment can compensate for the underestimation bias of expert judgment.

Mathematical models, which also use information of previous projects, are too complex or do not have a proven estimation accuracy. Function points are not usable on different systems and processes, and also cannot be used until the requirements specification is sufficiently detailed.

The work breakdown structure cannot be used in a project until the requirements are “reasonably defined” (as stated in the description of WBS on page 60). However, increasing accuracy is desired later on. It is relatively easy to retrofit the work breakdown structure into the estimation process; when refining or elaborating on early estimates, WBS is a perfect tool.

Logica intends to use WBS, expert judgment, analogy and function points in the future. Function points are required by their customers for specific projects; in that case, the disadvantage of the non-applicability to consulting projects is not an issue. Also in the businesses in question, software is specified very well.

In my opinion the criteria for estimation approaches are fair, except one: simplicity. Most of the methods which are deemed ‘too complex’ (i.e. the mathematical models) are complex to develop, but are not complex for the estimators. A mathematical model could then be used in conjunction with expert estimation and WBS, as a replacement for the analogies. This would reduce the amount of time needed for an estimation, while still compensating for the underestimation bias.

6. Discussion

In this chapter, I will show what has been learned on the fields that were mentioned in the introduction: massively parallel computing, service-oriented architecture and software effort estimation. These remarks answer the questions posed in the introduction.

6.1 Massively parallel computing

The Tesla is an excellent example of a massively parallel processor. Due to its low price, it enables researchers and developers to make use of MPC, where they previously could not due to budget constraints.

The Tesla is not suitable for all algorithms. Where it especially shines is when large datasets are used (millions of data points) or when the calculations per data point in the dataset are intensive (i.e. expressible in one function), but equal for each data point. This means that an algorithm has to be restructured so that the equality is ensured for each data point.

The Hough transformation, in this case used to detect lines in an image, is only partially suited for this hardware architecture. It does satisfy the criterium of intensive calculations (computing sine and cosine is relatively hard); however the criterium of millions of data points does not entirely hold. Even when the images are large, only a few percent of the pixels actually need calculation (in the case of an actual business card image). This also means that the calculation is not the same for every data point, causing a performance hit.

In the description of massively parallel computing on page 26, I stated "the data-parallel model fits this architecture very well". However, because the data and the application need to be loaded into the Tesla from a host computer, and because the Tesla can not synchronize all its data while running a kernel, the algorithm also needs to fit the master-slave model. The master process is then run on the host computer, and the slave processes on the Tesla.

The image can be split up in larger portions than just per pixel, and be sent of to different processing elements. The fact that not every portion is equal does not matter in a master-slave model. This also means that a cluster computing architecture could have been used instead of a massively parallel computing architecture.

I have been unable to find a definitive classification of algorithms, therefore it is difficult to say which classes of algorithms fit the Tesla architecture well. Still, algorithms from all fields of science and engineering have been successfully implemented on CUDA technology, as shown by Nvidia[1] Some examples are the fields of digital content creation, medical imaging and finance, which have implemented neural networks, data mining algorithms, signal processing

[1] `http://www.nvidia.com/object/cuda_home.html`, viewed on 21 october 2009.

algorithms and visual tracking algorithms. It is likely that the suitability of an algorithm for the Tesla processor needs to be decided on a case-by-case basis.

In the case of the Hough transformation, the Tesla processor suffers from relatively long startup times. This is because the time it takes for a kernel to load and start is constant. The business card application only works with data sets of at most a few megabytes, while the Tesla has the capability to store more than a gigabyte. Processing such a large dataset will make the load time of the kernel insignificant; it is not insignificant now.

The research questions mention the implementation of '3D image rendering' on a massively parallel processor. However, due to the complexity of the algorithms involved, the Hough transformation was used. The Hough transformation can be transformed to a demonstrator by integrating it into the business card application that already exists. I have not done this, as it was not necessary to answer the main research question.

6.2 Service-oriented architecture

In this section, I will show how I applied the design principles of Erl in my design. The design principles coupling, abstraction and statelessness are of specific importance for the Hough transformation case study.

With regard to the service model: for my purpose, where I want services to be able to harness the power of massively parallel computing, a utility service is the best fitting service model. It is technology oriented and agnostic of any business process.

Contracts

The business unit where I worked at Logica does not have any standardization in place for service-oriented architectures, therefore I could work according to my own standards. My service to use massively parallel computing in a service is relatively reusable, but only specializes in the Hough transformation. Therefore the data representation is very specific: it only accepts black-and-white images and returns images containing a Hough transformation.

Coupling

In the service I designed, I of course strived for Logic-to-Contract coupling. When developing a prototype, using Contract-to-Functional coupling speeds things up because you need less abstraction. However, that violates the recommendations in the next section too. The massively parallel computing service can be replaced by a 'normal', single-threaded computing service. That service is slower, but enables the use of a backup system.

Abstraction

The abstraction of functionality is where the focus lies in my design. Although the Tesla is capable of many things, only the basics will be used in a service-oriented architecture. If a developer needs to use specific capabilities, it is likely that he or she will program for the Tesla directly. To make the power of MPC available to service customers, the technology is abstracted away completely.

Reusability

I have built a utility service, which is reusable by definition. A utility service “provides functionality that is not tied to any business process”, i.e. can be used by all kinds of business processes, hence is reusable. So if the service is not reusable it won’t be a real utility service. Because the service is specific, I (or the service customer) don’t need to write data transformation logic.

Autonomy

To build a purely autonomous service, I need exclusive access to both the Nvidia Tesla and the data that needs to be processed. The former is not a problem, the latter is. I have designed the service so that all data is copied at runtime, ensuring exclusive access during execution of the task, but this increases transmission times. (Erl does not address this problem: he just states that the ideal is a seperate dedicated database for each service. How these databases should be synchronized is left unclear.) The alternative is sharing the database, resulting in *Service Logic Autonomy*, which is one step lower on the autonomy ladder.

An autonomous service should be scalable, and scalability often means concurrency. The design principles are not just meant to be used on each service by itself, but are meant to increase the usefulness of the entire service inventory. Unfortunately I cannot design the service so, that other services become scalable. This would require a parallelizing compiler, which is a research field on its own.

Statelessness

A utility service like mine should be stateless. If it is not, this has a serious effect the design principles reusability, autonomy and coupling. To reduce start-up times (as recommended by Erl), I have just used a binary format for the data to send to the Tesla, considering the amount of data it will process. This is in contrast to using an XML format.

Discoverability

As with abstraction, Logica does not have any templates or other standards for writing service meta information. This changes on a project-by-project basis. Neither is there any service registry. The documentation will thus have to be stored in Logica’s standard document management system. To comply with the advice of letting people with ‘required communication skills’ review the service profile, I let my project manager review it.

Composability

Existing bottlenecks may possibly be alleviated by adding a massive parallel computing service to the service composition, if the bottleneck is processing related, and not for example I/O related. The specific service that is the bottleneck should be changed, so it delegates the intensive processing to the new service.

Logica does not have a service inventory, so the inventory stages defined by Erl do not apply.

Other remarks

Designing a generic service to easily implement new algorithms is not possible without a performance hit. One would need to build all kinds of basic functions which could then be chained to an algorithm. As mentioned before, loading a new kernel takes a lot of time, and chaining kernels would increase the loading times accordingly.[2] What *would* be possible, is the approach of Silis and Hawick [22, 35], where one builds a service that accepts CUDA kernels, executes them and returns the accompanying data.

However, service-oriented solution logic is often written in more high-level languages like Java or C#, and it is not clear that the programmers of these services are also skillful at programming in C for the Tesla processor. Besides, even if the programmer is experienced in C programming, to get decent performance from the Tesla processor requires thorough knowledge of the physical hardware architecture of the Tesla processor.

Still, the advantage of such a system is that it then becomes easy to store kernels in a database, and retrieve them when necessary. This could be easily integrated into a service-oriented architecture. The disadvantages still remain: a kernel must be programmed manually, so parallelisation of existing services will not be automatic (and therefore, a lot of work). The design of a generic service enables the operation of datacenter services. Since plain datacenter services are not Logica's core business, such a datacenter service could only be used by other applications delivered by Logica. This could make deployment at customer's sites faster and cheaper, since the hardware is only installed at one location.

6.3 Software effort estimation

At this point, estimating the effort required for implementation of new software projects based on the Tesla processor is difficult. No estimation data is available on previous projects with the Tesla, which means that analogy-based estimation cannot be used. This also holds for approaches based on mathematical models. Function point-based estimation is not applicable for Tesla development, because the number of inputs and outputs (the basis for function points) is unrelated to the amount of effort needed to implement an algorithm. However, a combination of work breakdown structure and expert estimation is feasible, if the expert has experience with the Tesla or similar technologies.

It is difficult to envision a business service based solely on the Hough transformation service. However, a library of multiple graphics algorithms (i.e. implementing the AForge.NET library on the Tesla) would be very useful. It enables Logica to deliver 'heavier' applications than the competitors. The cost of this can be calculated using the WBS and expert estimation methods as described before.

[2] NVIDIA has promised that, in the newest generation of Tesla processors which will be introduced in 2010, kernels can be loaded while others are executing. This would make kernel chaining feasible.

7. Conclusion

So, how *can* massively parallel computing be offered as a (business) service?

As we have seen in the previous chapter, there are a few sides to this question. It is clear that massively parallel computing can be employed, cheaply, by using Nvidia Tesla processors. The use of these processors in a service-oriented architecture can then be accomplished in two ways:

- by creating an inventory of graphical algorithm services, or
- by building a service that runs arbitrary massively parallel applications on demand.

The business case for the first way is that new graphics-intensive applications can be built very fast, since they can reuse the already implemented services. The application will also perform really well in terms of execution times, because the services can be optimized.

The second way takes away the hassle of buying and setting up the hardware for each new project. Developers can simple reuse the existing hardware infrastructure. It is also more versatile, because the uses are not limited to already implemented algorithms.

Of course, both methods can be combined to benefit from the best of both worlds.

7.1 Future directions

I would recommend that the second way of utilizing the Tesla in service-oriented architectures (running arbitrary applications) be more thoroughly analyzed and tried. As I stated, the hassle of setting up the hardware is gone. It would make the power of massively parallel computing available for a lot more projects.

Furthermore, more research is necessary to determine ways to analyze the suitability of an algorithm to the master-slave and data-parallel models. This will enhance the decision process for which technologies to use for new projects. Due to the similarities between the algorithms suited for massively parallel computing and cluster computing, it might be possible to use the Tesla processor for prototyping purposes for cluster computing software and vice versa. This needs to be investigated.

Finally, I advise that the actual effort for additional projects on the Tesla be measured and used in estimations for new projects. Special attention should go to the time required to debug the applications.

Appendix: Building trial applications

To learn how to program for the Tesla and to demonstrate the power of the processor, I've composed two small programs. The first multiplies two matrices, a popular benchmark to measure raw performance. The second implements the basics of a naive image stitching algorithm.

Matrix multiplication

The first program multiplies two matrices and stores the result in a third matrix. This is a commonly used function to benchmark raw processing power [9], which is also often used in real-world applications (so it is not a purely synthetic benchmark). It works by creating two matrices of 10,000 by 10,000 elements (in this case, random single precision floating point numbers). These matrices are then multiplied using the CUBLAS library function `cublasSgemm()`, which is equivalent to the Intel Math Kernel Library (MKL) function `sgemm()`. It performs a general matrix multiplication, i.e. there are no optimizations for sparse or symmetric matrices. This implies a complexity of close to $O(n^3)$.

Executing the program on an Intel Pentium 4 processor running at 2.8 GHz takes about 20 minutes. Running it on the Nvidia Tesla takes roughly 8 seconds, an improvement of two orders of magnitude. This needs some correction for Moore's law, as the Pentium 4 was released in 2002 and the G80 in 2006.

Image stitching

I implemented another algorithm on the Tesla that was not trivial but not too difficult either. This time it was entirely built in kernel code. It is a naive stitching algorithm, i.e. an algorithm that determines a shift in the horizontal and vertical plane for one image to be layered on top of another, where the difference between the two images is minimized.

The algorithm works by moving image B over image A. At each position, the difference between the two images is determined by summing all absolute differences of the gray values of pixels that are over each other. Because the overlaid area differs each time, this needs to be compensated for. This is done for the vertical shift by taking only the area of image A that is overlaid by B, and calculating a correction factor that is dependend on the amount of vertical shift. The horizontal shift is compensated for by taking a fixed-size strip. The algorithm is naïve because it doesn't correct anything in the image like the perspective or rotation. It also doesn't take contrast between objects into account. In listing 7.1 the program is shown in the language Clean. The `Start` function nicely shows that the algorithm mainly consists of `map` functions, thereby showing the data parallelism.

Listing 7.1: The image stitching algorithm written in Clean

```
Width  = 5
Height = 3
Strip  = 2

imga :: {{Int}}
imga = {{ 1, 2, 3}
       ,{ 4, 5, 6}
       ,{ 7, 8, 9}
       ,{10,11,12}
       ,{13,14,15}}

imgb :: {{Int}}
imgb = {{ 1, 2, 3}
       ,{ 4, 5, 6}
       ,{ 7, 8, 9}
       ,{10,11,12}
       ,{13,14,15}}

ShiftX = [-4 .. -2]
ShiftY = [-1 .. 1]

Min :: [Int] -> Int
Min list = foldl Min 99999 list
where
    Min :: Int Int -> Int
    Min a b
        | a <= b = a
        | otherwise = b

Start::((Int,Int),Int)
Start = Minelement (map Sumelements (map Getelements
                                    (Shiftresults (diag2 ShiftX ShiftY))))

Minelement :: [((Int,Int),Int)] -> ((Int,Int),Int)
Minelement [((x, y),sum)] = ((x, y),sum)
Minelement [((x1,y1),sum1):((x2,y2),sum2):xs]
| sum1<=sum2 = Minelement ([((x1,y1),sum1)] ++ xs)
| otherwise  = Minelement ([((x2,y2),sum2)] ++ xs)


Sumelements :: ((Int,Int),[Int]) -> ((Int,Int),Int)
Sumelements ((x, y),list) = ((x,y), (foldl (+) 0 list)
                                     / (Strip * Height - abs(y)))

Getelements :: ((Int, Int),[((Int, Int),(Int,Int))]) -> ((Int,Int),[Int])
Getelements ((x, y),list) = ((x, y), (map Getelement list))
where
    Getelement :: ((Int, Int),(Int,Int)) -> Int
    Getelement ((xA, yA),(xB,yB)) = abs(imga.[xA,yA] - imgb.[xB,yB])

Shiftresults :: [(Int,Int)] -> [((Int, Int),[((Int, Int),(Int,Int))])]
Shiftresults [] = []
Shiftresults [(x,y):xs] = [((x,y),
         [((xA,yA), ((HorizontalShiftB xA x), (VerticalShiftB yA y))) \\
                                     xA <- HorizontalShiftA x
                                   , yA <- VerticalShiftA y
                                   ])] ++ Shiftresults xs

HorizontalShiftA :: Int -> [Int]
HorizontalShiftA x = [(Width + x) .. (Width + x + Strip - 1)]

VerticalShiftA :: Int -> [Int]
VerticalShiftA y = [(max 0 y) .. (min Height (Height + y) - 1)]

HorizontalShiftB :: Int Int -> Int
HorizontalShiftB xA x = 0 - x - (Width - xA)

VerticalShiftB :: Int Int -> Int
VerticalShiftB yA y = yA - y
```

Bibliography

[1] A. Albrecht. Measuring application development productivity. In *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, volume 83, page 92, 1979.

[2] G. S. Almasi and A. Gottlieb. *Highly parallel computing.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[4] Z. Baida, J. Gordijn, and B. Omelayenko. A shared service terminology for online service provisioning. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, pages 1–10, New York, NY, USA, 2004. ACM.

[5] R. G. Belleman, J. Bédorf, and S. F. Portegies Zwart. High performance direct gravitational N-body simulations on graphics processing units II: An implementation in cuda. *New Astronomy*, 13(2):103–112, 2007.

[6] B. Boehm, C. Abts, and S. Chulani. Software development cost estimation approaches – a survey. *Annals of Software Engineering*, 10(1-4):177–205, 2000.

[7] L. C. Briand and I. Wieczorek. Resource modeling in software engineering. *Second edition of the Encyclopedia of Software Engineering*, 2002.

[8] D. L. Cook. *Program evaluation and review technique – applications in education.* Office of Education, U.S. Department of Health, Education, and Welfare, Washington, D.C., 1966.

[9] J. J. Dongarra, P. Luszczek, and A. Petitet. *The LINPACK Benchmark: past, present and future*, volume 15 issue 9, pages 803–820. John Wiley & Sons, Ltd., University of Tennessee, Department of Computer Science, Knoxville, TN 37996-3450, U.S.A.; Sun Microsystems, Inc., Paris, France, 2003.

[10] R. O. Duda and P. E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[11] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[12] T. Erl. *SOA Principles of Service Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[13] R. Farber. The future looks bright for teraflop computing. *Scientific Computing*, 24(10):34, 2007.

[14] G. Finnie, G. Wittig, and J. Desharnais. A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models. *The Journal of Systems & Software*, 39(3):281–289, 1997.

[15] G. Finnie, G. Wittig, and J. Desharnais. Reassessing function points. *Australasian Journal of Information Systems*, 4(2), 1997.

[16] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[17] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley Longman, 2003.

[18] S. Grimstad and M. Jørgensen. A framework for the analysis of software cost estimation accuracy. In *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 58–65, New York, NY, USA, 2006. ACM.

[19] S. Grimstad, M. Jørgensen, and K. Moløkken-Østvold. Software effort estimation terminology: The tower of Babel. *Information and Software Technology*, 48(4):302–310, 2006.

[20] D. Guedes, W. Meira, and R. Ferreira. Anteater: A service-oriented architecture for high-performance data mining. *IEEE Internet Computing*, 10(4):36–43, July 2006.

[21] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[22] K. Hawick, H. James, C. Patten, and F. Vaughan. DISCWorld: A distributed high performance computing environment. In *HPCN Europe 1998: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 598–606, London, UK, 1998. Springer-Verlag.

[23] O. Helmer. Analysis of the future: the Delphi method, 1967. DTIC Research Report AD0649640.

[24] J. Hill, L. C. Thomas, and D. E. Allen. Experts' estimates of task durations in software development projects. *International Journal of Project Management*, 18(1):13–21, 2000.

[25] M. Jørgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007.

[26] B. W. Lo and X. Gao. Assessing software cost estimation models: criteria for accuracy, consistency and regression. *Australasian Journal of Information Systems*, 5(1), 1997.

[27] S. G. MacDonell and M. J. Shepperd. Combining techniques to optimize effort predictions in software project management. *Journal of Systems and Software*, 66(2):91–98, 2003.

[28] K. Moløkken and M. Jørgensen. A review of surveys on software effort estimation. In *Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE03)*, pages 223–230, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[29] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide v1.1*, nov 2007.

[30] NVIDIA Corporation. *NVIDIA CUDA CUBLAS Library v2.0*, March 2008.

[31] J. D. Owens, D. Luebke, N. Govindaraju, et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[32] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[33] S. F. Portegies Zwart, R. G. Belleman, and P. M. Geldof. High-performance direct gravitational N-body simulations on graphics processing units. *New Astronomy*, 12(8):641–650, 2007.

[34] I. H. F. Santos, A. B. Raposo, and M. Gattass. A service-oriented architecture for a collaborative engineering environment in petroleum engineering. In *Proceedings of Virtual Concept*, 2006.

[35] A. Silis and K. Hawick. World wide web server technology and interfaces for distributed, high-performance computing systems. Technical Report DHPC-017, Department of Computer Science, University of Adelaide, 1997.

[36] R. C. Tausworthe. The work breakdown structure in software project management. *Journal of Systems and Software*, 1(3):181–186, 1980.

[37] S. Vinoski. Web services interaction models. current practice. *IEEE Internet Computing*, 6(3):89–91, 2002.