Non-invasive software design for many-core architectures

Dan Nilsson

Luleå University of Technology
MSc Programmes in Engineering
Media Technology
Department of Computer Science and Electrical Engineering
Division of Computer Engineering
Preface
The work for this master’s thesis was conducted between October 1, 2008 and February 27, 2009 at Luleå University of Technology (LTU).
Abstract

There’s no doubt that the fundamentals of computer programming were broken at the launch of the multi-core processor. It enabled multiple computations to be processed simultaneously and independently, in parallel. Declared as many-core architecture programmable GPU and CPU enabled hardware provides teraflop performance to the consumer. This horizontal scaling reduces performance for software that is reluctant to optimize by widening the processing base. Instead new models must be developed which adaptively interchanges to the underlying hardware and enables the software to incorporate distributive computing where the workload is divided among the available processors. Models which reinstate the processing hardware capabilities as the definition to the limitations of performance and let software scale accordingly.

This thesis presents a design scheme for integrating the CPU and GPU, where their differences lay the foundation for the solution. Further, the capabilities of re-enabling the hardware as the performance limiter, in multi-core environments, are explored.

It’s fundamental to the design that it combines features with performance. Effectively allowing the parallelization of the application to drive the design choices, and therefore avoiding unnecessary overheads. The primary goal of the design is to create an inherent component structure, separating the different tasks early. Allowing multiple processing units of varying nature to be combined and process concurrently, while maintaining linear performance to the quantity of cores. Expanded upon in the actual implementation, this style of composition creates a dynamic underlying framework for the program to be built upon.

Build from the ground up around the notion of concurrent processing, the design is developed for maximal core utilization, adapting to current workloads of different tasks and issuing load balancing to attain the best performance. A hybrid approach is used for concurrency with a clash of both the fine grained model and the coarse threaded model.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>2</td>
</tr>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Introduction</td>
<td>6</td>
</tr>
<tr>
<td><strong>Definition of abbreviations and terminology</strong></td>
<td>7</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>7</td>
</tr>
<tr>
<td>Terms</td>
<td>7</td>
</tr>
<tr>
<td><strong>Theory</strong></td>
<td>9</td>
</tr>
<tr>
<td>Many cores converge</td>
<td>9</td>
</tr>
<tr>
<td>Central Processing Unit</td>
<td>10</td>
</tr>
<tr>
<td>64-bit extensions</td>
<td>10</td>
</tr>
<tr>
<td>Graphics Processing Unit</td>
<td>11</td>
</tr>
<tr>
<td>The vector processor...</td>
<td>11</td>
</tr>
<tr>
<td>...and its evolution</td>
<td>12</td>
</tr>
<tr>
<td>Nvidia’s unified shader model architecture</td>
<td>13</td>
</tr>
<tr>
<td>Supporting structures</td>
<td>15</td>
</tr>
<tr>
<td>Compute Unified Device Architecture (CUDA)</td>
<td>16</td>
</tr>
<tr>
<td>Open graphics library (OpenGL)</td>
<td>17</td>
</tr>
<tr>
<td>Coherence and consistency</td>
<td>19</td>
</tr>
<tr>
<td>Coherence misses</td>
<td>19</td>
</tr>
<tr>
<td>Race condition</td>
<td>20</td>
</tr>
<tr>
<td>Deadlock</td>
<td>20</td>
</tr>
<tr>
<td><strong>Designing for the n-core machine</strong></td>
<td>22</td>
</tr>
<tr>
<td>Memory protection</td>
<td>22</td>
</tr>
<tr>
<td>Non-invasive defense</td>
<td>23</td>
</tr>
<tr>
<td>Work stealing</td>
<td>25</td>
</tr>
<tr>
<td>Distributed computing</td>
<td>25</td>
</tr>
<tr>
<td>OpenGL</td>
<td>26</td>
</tr>
<tr>
<td>CUDA</td>
<td>26</td>
</tr>
<tr>
<td>Symbiotic integration</td>
<td>27</td>
</tr>
<tr>
<td>Design reusability</td>
<td>28</td>
</tr>
<tr>
<td>Details of design implementation</td>
<td>29</td>
</tr>
<tr>
<td><strong>Analysis of results</strong></td>
<td>32</td>
</tr>
<tr>
<td><strong>Discussion</strong></td>
<td>37</td>
</tr>
<tr>
<td>Further development</td>
<td>38</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>39</td>
</tr>
</tbody>
</table>
Appendix A System requirements 40
   Hardware 40
   Software 40

Appendix B Performance data 41

Appendix C Data apportion 43
   CUDA 43
   OpenGL 44

Appendix D Program code excerpts 45
   Declarative files 45
   Synchronization code 46
Introduction

The computer processing environment is shifting from discrete entity devices, where disconnected tasks are processed individually, toward complex inter-networked processors. In-order centralized sequential tasks become distributed and parallel, fused together to contribute collectively. The boundaries between processing units are blurred to form a large intertwined general-purpose processing grid. While the common general-purpose model relies on fast attainable data from caches, its significance decreases as data are scattered in the network. Instead data are streamed directly whilst the latency for loading is hidden through computations. The model, further, causes data migration to become expensive between processing units and provokes integration penalties due to excessive locking overhead. As the network grows the performance becomes a response to the quality of the structural design and either increases with it or ends up smaller than its parts.

In networks which only share the smallest common denominator, communication is handled through abstractions. These are used to hide the variety of hardware specializations that may be present. The prime example is graphics processors which only processes pure graphical elements like vertices and triangles. These are tedious to work with since it’s hard to map computational structures to them, so instead abstractions of the graphical elements are used. By extracting the few core instructional set and combine these to abstract features, general-purpose computing is achievable. However the price of convenience proves to be high and introduce new problems.

This thesis will investigate and propose a combined software design which bridges the gaps between all processing units within a personal computer. Generally only the CPU and GPU allow general computation to be executed, however the design should also be extendable to involve future devices. Further, to achieve full utilization the solution should adapt to the underlying hardware. As the computer field have become non-uniformed with producers offering processors with 1-8 cores per chip and GPUs with 32-240 cores per chip the program design must be flexible enough to dynamically adapt to all possible combinations of hardware that may arise. As the design adapts to the computer hardware the performance should follow the quantity increase of CPU cores, while significantly be increased as a GPU is enabled. The research only considers consumer level hardware which excludes multiple GPU devices and CPU processors. The performance will be tested on a developed implementation which strictly conforms to the result of the design.

The theoretical software design should be possible to implement independently of the underlying hardware. Preeminent performance is only available on the x86-64 architecture due to hardware specific optimizations. The full requirements for the hardware and software of the implemented program are listed in appendix A. Since GPU hardware doesn’t conform to any industry standard and thus are subject to more rapidly change older hardware aren’t compatible.

---

3 That is multiple sockets.
Definition of abbreviations and terminology
This section defines the abbreviations and terms used in this thesis.

Abbreviations
ABI Application binary interface
ALU Arithmetic logic unit
API Application programming interface
CPU Central processing unit
DLP Data-level parallelism
DPC Data-parallel computing
FLOP Floating-point operations per second
GAS GNU assembler
GHz Gigahertz
GLSL OpenGL shading Language
GPGPU General-purpose computation on graphics processing units
GPU Graphics processing unit
ILP Instruction-level parallelism
MIMD Multiple instruction multiple data
OpenGL, OGL Open graphics library
SIMD Single instruction multiple data
SM Streaming multiprocessors
SP Scalar processor
SSE Streaming SIMD extensions instruction
TFLOP Tera floating-point operations per second \((2^{12} \text{ FLOPS})\)
TLP Task-level parallelism
TPC Thread processor clusters
VPU Vector processing unit

Terms
Cache The highest levels of a memory hierarchy.
Concurrency The act of giving an illusion of parallelism, usually pursued by preemptive multitasking.
Double word 4 bytes or 32 bits.
Fragment The unit of execution for the fragment processor stage in OpenGL
GNU Assembler An assembly language which uses the AT&T syntax.
Gather A data read which is done indirectly.
Many-core The concept of using all available processing elements on an individual machine.
Multi-core The concept of using all available CPU cores on an individual machine.
OpenGL An API for primarily developing real-time graphics.
Parallelism Multiple computations proceeding independently and separately of each other. Parallelism can only occur on a processor with multiple cores.
Scatter A data write which is done indirectly.
Shader A small program devoted to run on the GPU.
Thread An execution context.
Vector A set of numbers packed into a single operand.
x86-64 A processor architecture with 64 bit extensions, featured in almost all personal computers.
Theory
Many-core realizes general-purpose computing and scales it among the available hardware by digesting a blend of different underlying frameworks of supporting theories. Most of them are intended for either high-performance computing (HPC) or have highly specialized fields of usage and thus aren’t directly mapable to general-purpose sequential programs. However, it’s the cohesive environment they shape that defines the concept of many-core, the notion of a unified platform.

This chapter describes the fundamental theories behind many-core computing and expands them in the context of the subject. It provides both a broad view into the subject and as an introduction to the main thesis. Further, the chapter handles both the hardware and software sides from low-level implementations to high-level abstractions.

Many cores converge
The recent transition in x86 CPU design introduced multi-core computing, which scales horizontally as oppose too vertically. The previous continuous trend of vertical scaling by increasing the core CPU frequency lowers the cycle time, since \( frequency = \frac{1}{cycle\ time} \). By lowering the cycle time large performance gains can be achieved, due to the raised instructional throughput, cheaply without altering the hardware architecture. This link between CPU and application performance allowed applications that wasn’t constrained by the hard drive to freely increase its performance, since execution time = \( \frac{cycles}{freq} \). However, as the frequency is raised the current leakage and thermal issues is exacerbated and beyond a certain point vertical scaling isn’t viable due to the dramatically increased power consumption and heat. In late 2005 AMD released the first multi-core x86 based CPU which scaled horizontally by adding a physical core to the CPU while decreasing the core frequencies. It was found that by scaling back 400 MHZ or 10 % of the maximum frequency the CPU power consumption decreases by 40 % (Anderson, 2006). With the lower frequency suddenly, for the first time, programs performed slower than on the previous hardware generation even though the theoretical performance potential was raised significantly.

With this change the fundamental sequential nature of all programs need to be re-architected to realize the multiple CPUs. However, the non-cohesive user base, segmented by the number of core, there’s no clear choice of platform to architect for. The general conception of the solution is to statically define the hardware at compile time. The resulting programs are completely unaware of its hardware and unable to adapt to it, thus a gap between the program architecture and the hardware is formed. While older CPUs with fewer cores will perform worse than the reference hardware due to unnecessary context switching future hardware won’t have their full potential realized. These performance “gaps” are the product of poor hardware management and none scaling software design which can only increase with time.

To make the situation worse the term many-core was coined which realizes all available processing power on a single computer. For desktop development this means utilizing both the graphical processing unit (GPU) and CPU.

In 2002 the sub-programmable GPU pipeline was introduced with 40 execution units which removed the fixed functionality from the vertex- and fragment processors. Currently the fastest single-die GPU can perform 0.9331 TFLOPS, as Equation 1 shows.

\[
\begin{align*}
30 \text{ streaming multiprocessors} \times \\
8 \text{ scalar processors} \times \\
(1 \text{ multiply and add (MAD) } + 1 \text{ multiply (MUL)}) \times \\
1.296 \text{ GHz} & = 0.9331 \text{ TFLOPS}
\end{align*}
\]

\textbf{Equation 1} With its 240 cores that can perform 3 FLOPS each the Nvidia GTX 280 reaches almost 1 TFLOPS.

In comparison a quad core CPU at 3.2 GHz can perform 102.4 GFLOPS (using 8 flops / clock cycle) and IBM’s Blue Gene/L with 65 536 CPUs performs 135 TFLOPS. However, current
algorithms are hard to map to the GPU. The pipeline is highly optimized for real-time interactive graphics thus advance abstractions are needed for general-purpose computation on graphics processing units (GPGPU).

The GPU architecture is fundamentally different from the CPU but the same rules apply to it; full utilization is only achievable without performance gaps. Since the introduction of the GPU there’s been an annual increase in processor cores of about 1.6 times while frequencies increase about 1.3 times during the same period (Hensley, 2007). In a year the fastest processor will probably have well over 300 cores which would result in that the model for determining hardware statically performs even worse for GPUs.

Central Processing Unit

While the central processing unit (CPU) is very complex this section will only cover the core architectural features in the context of many-core. The following text will refer to the 32-bit version of Intel’s IA-32 CPU architecture as IA-32 and the IA-32 architecture with AMD64 extensions as x86-64.

The performance of sequential programs is generally dictated by its cache and the degree of instruction level parallelism (ILP) exploitation. Intel’s IA-32 architecture incorporates a partly shared hierarchy of caches which generally vary in size among chips while the hierarchy structure is constant. Current processors uses two 32 KB private level 1 (L1) cache for instructions and data residing in each core and two general off-core 6 MB level 2 (L2) caches which are used for sharing among cores.

64-bit extensions

The original x86 design used a 16-bit address space which was increased to 24-, 32- and recently 64 bits. The architecture with 64 bits of addressable space is generally referred to as x86-64² which also defines new features and instruction sets; AMD64 and Intel 64. With a word size of 32 bits only $2^{32}$ bytes (4 GB) of memory could be addressed while with an extended 64-bit word size allow $2^{64}$ of addresses in theory³. The specification defines 16 general-purpose registers (GPR), 16 SSE registers and 8 floating point registers which are 64-, 128- and 80 bits wide respectively. Further additions are pointers and long data types now require 8 bytes. Due to these changes the x86-64 architecture permits further optimizations than IA-32 did. The additional 8 GPRs loosen the constraints for stack usages. With up to 6 arguments⁴ residing in the registry most function arguments can reside entirely in registers instead of the stack. As the registry is located much closer to the core then stack memory this result in better performance which is shown in Listing 1.

---

² IA64 is a separate non-x86 compatible Intel architecture commonly named (Itanium).

³ Generally the motherboard manufactures suppress the amount of supported memory below the theoretical limit.

⁴ This statement only considers the C language access patterns.
<table>
<thead>
<tr>
<th>IA-32 version</th>
<th>x86-64 version</th>
</tr>
</thead>
<tbody>
<tr>
<td>daxpy:</td>
<td>daxpy:</td>
</tr>
<tr>
<td>pushl %ebp</td>
<td>cvtsi2sdq %rdi, %xmm2</td>
</tr>
<tr>
<td>movl %esp, %ebp</td>
<td>mulsd %xmm0, %xmm2</td>
</tr>
<tr>
<td>fildl 24(%ebp)</td>
<td>movapd %xmm2, %xmm0</td>
</tr>
<tr>
<td>fmull 8(%ebp)</td>
<td>addsd %xmm1, %xmm0</td>
</tr>
<tr>
<td>faddl 16(%ebp)</td>
<td>ret</td>
</tr>
<tr>
<td>popl %ebp</td>
<td>ret</td>
</tr>
</tbody>
</table>

7 instructions | 5 instructions

Listing 1 A comparison between a 32 bit and a 64 bit function, both are based on DAXPY (Double precision a multiplied by X plus Y). Both receive a, X and Y as arguments, the resulting code clearly shows that the x86-64 version doesn’t use the stack at all which result in less instruction.

**Graphics Processing Unit**

There’s no standard design for GPUs, as with Intel’s x86 CPU design. Instead the graphics APIs functions as standardization among the hardware. The fears of patent infringement keep the manufacturers from unraveling most of the low-level technical details for the hardware. The lack of a consistent platform with full documentation constrains the technical freedom and subdivides the market.

This section will describe the differences and similarities between CPUs and different GPUs, the limitations of heavy parallelization and the fundamental concepts for GPU computing.

**The vector processor...**

The spiritual predecessor to the concept of many-core is the vector processor (VPU) that was originally designed for maximizing throughput through parallelization. This is achieved by combining a scalar pipeline with multiple vector pipelines which executes different sets of data on multiple instances of each task in parallel. Until recently the model were used primary by scientific institutions for simulations on large amount of data with high arithmetic calculation intensity. As instructional level architecture focus on attaining stable execution time, across all type of applications, their performance is linked to cache size and to what degree the instruction level parallelism (ILP) is exploited. Through the specialized hardware, VPU architectures are capable of increasing efficiency while maintaining large datasets without caches. Performance are thus more complex to increase on parallel applications, Amdahl’s law (see Equation 2) states the parallel fraction of the code, number of execution units as complemented factors. Other factors are the amount of synchronization and dataset size.

\[
S = \frac{1}{\frac{f_{\text{parallel}}}{P} + (1 - f_{\text{parallel}})}
\]

Equation 2 Amdahl’s law, where $S$ is the speedup increase, $f_{\text{parallel}}$ is the fraction of parallelization and $P$ is the number of execution units.
Utilizing parallelism through instanced tasks on multiple data has numerous consequences on the instructional bandwidth and data throughput. The single pipelined instruction based processor is only capable of instruction level parallelism (ILP) and must iterate over multiple data; in contrast each VPU is scheduled only one data element. In Listing 2 the difference is confirmed, the substantial difference in instructional count decreases the instruction bandwidth for individual VPUs. The non-sequential behavior of the vector code allow further optimizations by exclusion of instructions for counter management and branching which also decreases the risk of pipeline stalling due to interlocking.

<table>
<thead>
<tr>
<th>x86-64 processor code</th>
<th>Vector processor code</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.equ size, 128</td>
<td>movq eleX(,%edi,4), %rax</td>
</tr>
<tr>
<td>movq $0, %rdi</td>
<td>movq eleY(,%edi,4), %rbx</td>
</tr>
<tr>
<td>loop1:</td>
<td>imulq $a, %rax</td>
</tr>
<tr>
<td>movq eleX(,%edi,4), %rax</td>
<td>addq %rax, %rbx</td>
</tr>
<tr>
<td>movq eleY(,%edi,4), %rbx</td>
<td>...</td>
</tr>
<tr>
<td>imulq $a, %rax</td>
<td></td>
</tr>
<tr>
<td>addq %rax, %rbx</td>
<td></td>
</tr>
<tr>
<td>addb $1, %dil</td>
<td></td>
</tr>
<tr>
<td>cmp $size, %dil</td>
<td></td>
</tr>
<tr>
<td>jne loop1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

897 instructions / processor 4 instructions / processor

Listing 2 Compares x86-64- with fictitious vector processor code for $Y = a \times X + Y$, for a data element of size 128 the x86-64 code would result in 897 instructions while a VPU with a width of 128 only need 4. The difference in instructions is a result of the sequential nature of CPUs, which forces it to loop over the data. The loop of 7 instructions is executed 128 times plus the clearing instruction on row 2, giving a total of 897 instructions. Since each VPU execute exactly one data element only the core processing instructions are needed, thus giving a total count of 4.

...and its evolution
This section describes the differences to the CPU, the basics of the hardware model and the fundamentals of the architecture.

Execution units of any form contain, much generalized; control, data-path and storage. The control handles in which direction the flow of computation should proceed. The actual computations are executed in the data-path, which usually contain multiple arithmetic units. A series of differently leveled storage chips is provided from where data is gathered. The distribution of the resources in these three areas determines the characteristics of the final design.

Modern x86 based CPUs were originally architected for retaining fast single element general computing, and therefore contain complex solutions for storage and control. Most resources on the CPU are dedicated towards storage; modern chips incorporate hierarchies of caches with up to three levels. The data-caching is optimized for minimizing the latencies for data fetching but the transient behavior decreases the throughput significantly due to the usage of write through. For a multi-core CPU that uses shared caching between cores the problem gets worse, due to the coherent dependency. Cacheability control instructions that perform non-temporal loads and stores by bypassing the cache were introduced through SSE2 to minimize the cache pollution. However stores

---

5 With the die area as measurement
6 See section about Coherence and consistency
7 The instructions are MOVNTDQ, MOVNTPD, MOVNTI & MASKMOVDQU
performed by these instructions explicitly states that data may not be written in program order and may be brought back into the cache due to the speculative nature of the CPU data. The flow control for x86 was dedicated more resources than the actual computations units to allow features like advance branch prediction and out of order execution. The data-path of each CPU core don’t expose any stream parallelism and thus may only execute exactly one stream. Some parallelism is provided by the SSE units which have 8 ALUs that handle floating-point operation over 128-bit registers (see section Central Processing Unit).

The fundamental concept behind GPUs is: processing single tasks slow while allow fast multi-task processing. The CPU model of data handling, task-level parallelism (TLP), executes tasks on precisely one set of data. If the tasks are changed to execute multiple data elements in parallel, data-level parallelism (DLP) is exposed. GPUs may use any of these models, but was optimized to use data-parallel computing (DPC) which incorporates DLP with a third model instruction-level parallelism (ILP) that details parallelism within the data.

Commonly Processors are categorized by Flynn’s taxonomy (Chapman, Jost, & Van Der Pas, 2008), where the characteristics of the instruction stream and data stream are used as index. The four different possibilities together with the most common processors are listed in Table 1.

<table>
<thead>
<tr>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>Single core processors</td>
</tr>
<tr>
<td></td>
<td>Vector- &amp;</td>
</tr>
<tr>
<td></td>
<td>Fragmentprocessors (SP)</td>
</tr>
</tbody>
</table>

Table 1 Flynn’s taxonomy categorizes computers by instruction and data architecture into four distinct groups; SISD, SIMD, MISD and MIMD.

**Nvidia’s unified shader model architecture**

Older GPUs were divided into discrete stages, three of which were programmable while the rest used a fixed function. The three programmable stages; vertex, geometry (primitive) and fragment (pixel), all had different dependencies and characteristics for performing GPGPU operations. With the advent of modern GPUs the designs were unified for better flexibility, in this design evolution the shaders were converted into regular threads.

![Figure 1](image1.png)  
**Figure 1** Shown are two GPU architectures for thread processor cluster (TPC). Both incorporate 8 scalar processors (SP) per streaming multiprocessors (SM), instruction cache, scalar memory, shared memory, register memory and SFU units for transcendental (sin, cos, tan) & anisotropic texture filtering. While the older G80 GPU uses 2 SMs per TPC, newer uses 3 SMs per TPC.
Each graphics device is segmented in different streaming multiprocessors (SM), see Figure 1 for a schematic view of a GPU processor. The count varies greatly between products while the Nvidia GTX 280 have 30 the Nvidia 8800GT have 14. As the name implies the platform architecture (see Figure 2) is alike among all models, each SM containing 8 scalar processors (SP), two functions for transcendental, an instructional unit and a block of memory used for sharing. It’s responsible for the thread management which is all hardware based for zero-overhead. The MIMD design handles different programs in parallel which allows them to effectively process vertices in graphical applications. Created threads are mapped to the SPs by the thread manager in groups which correlate to the SIMD width. The width of the SIMD instruction stream is 32 fragments, i.e. threads, wide. Each SP can hold a maximum of 4 threads and operates on one of these fragments every clock cycle. A group of 32 fragments is therefore executed in 4 cycles.

A shared memory is available on all SMs, which is only typically 16KB, but discards the latency-heavy technique of thread interaction through the texture (video) memory which previous generations relied on. Because each streaming (vertex) multiprocessor handles multiple instructions they can use multiple contexts to handle TLP type algorithms. As the 8 SPs within a SM are SIMD devices they share the instructions, control unit, cache and the memory from where registers allocations are drawn. The shared instruction stream explicitly states that all data must be independently processed. Due to the absence of individual control units, caches and program counters GPUs can effectively use many more ALUs than a CPU. It’s the bare-bone design that allow the GPU to have a high arithmetic utilization, the GTX 280 is packed with 480 ALUs while the high-end quad CPUs only have 32 (8 per core). This big difference is the main reason why GPUs are currently processing 1000 GFLOPs while CPUs only do 100.

Figure 2 A simplified architectural view of the GPU is shown above. Most of the components are self handled and non-accessible to the developer or user. With a TPC count of 10, 3 SMs per TPC and 8 SPs per SM a total of 240 cores are enabled (10*3*8).
As previously stated, the GPU lacks all forms of significant cache. For streaming data this increases throughput, but general applications that usually rely heavily on non-sequential memory access suffers from large latency memory loads. This problem is solved by two techniques: memory coalescing and multithreading. Coalescing is achieved in software by the developer and involves grouping multiple loads into single transactions from memory. Instead of fetching 64 bits in 16 transactions they can be coalesced into a single transaction to achieve maximum bandwidth.

For off-chip memory operations, like the texture memory, latencies reach from 100 to 1000 cycles. Naturally these operations would stall the execution pipeline but GPUs hide these through computations. The concept is the same as with CPU multithreading; multiple contexts, defined as a group of 8 threads, are created and constantly switched. With the tight constraints for each SM, a memory of 32 KB dedicated for contexts, only a limited number of contexts can be handled. A context is executed until it’s either finished or a pipeline stall occurs due to a pipeline dependency, these can effectively be hidden through calculation by a context switch. It’s only by increasing the runtime for individual contexts that the maximal throughput is attained.

Supporting structures

This section gives an overview of the supporting structures to the many-core concept. A supporting structure defines a set of rules; usually in the form of an application programming interface (API), software development kit (SDK) or as a model of conception. The following structures, see Figure 3, are only a very small subset from what is available. The ones described in more detailed were chosen because they meet the requirements that were stated for this project.

![Figure 3](image)

**Figure 3** Above a hierarchical view of some supporting structures which solve parallel processing is shown. The figure is divided into sections which form a layered view where the lowest layer represent hardware with low capabilities and the top features high capability hardware with multiple CPUs and GPUs. The individual scaling properties varies considerably between supporting structures for hardware compatibility. As shown Pthreads and OpenGL maintains the best support but lacks support for the newest features.

The many-core programs should fully utilize all of the execution units irrelevant of the underlying hardware concept, it makes no distinction between new or old, high-end or low-end hardware. Naturally there’s a trade-off between feature offerings and maintaining backwards compatibility while scaling for new features and performance. All consumer GPUs support OpenGL but due to its graphics heritage it lacks all GPGPU functionality. Nvidia proclaims that CUDA is supported by 85 million devices thus a considerable user base already exists, but since none of those are AMD, Intel devices or Nvidia GeForce 7-series (G71) cards or older, is it viable to incorporate CUDA in consumer products? Vertical scaling is necessary to transpire in both directions, capitalizing on new hardware advantages while maintain backwards compatibility. In Figure 2 an abstract view of the individual scaling properties for different supporting structures are shown in consideration to each other. Using Pthreads for coarse threading tasks maintains backwards compatibility but can be hard to scale beyond 8 threads within certain models. OpenMP scales well beyond 8 threads by fine threading the tasks for a hybrid threading solution scales, but come with almost linear overhead as threads increase.
All of the available supporting structures focus on excelling in a small feature set. Only by incorporating multiple supporting structures in hierarchies can the functional scaling be preserved, when using many-core architectures, while increasing hardware compatibility.

**Compute Unified Device Architecture (CUDA)**

With the launch of Geforce 8 (G80) architecture Nvidia launched its unified shader model architecture (see section Nvidia’s unified shader model architecture) which every GPU from the company have incorporated since. It was an initiative to ease the burden of programming for the GPU that allow a new general abstraction of the hardware to be used in favor of the previously graphical APIs. The central GPGPU innovation of the architecture is Compute Unified Device Architecture (CUDA) which fundamentally is a model for programming Nvidia’s hardware. It defines a concrete API and abstract concepts for GPU resource management which are executed natively on the hardware. Features defined by the hardware are bundled into profiles, called compute capabilities, that can be queried by software for compatibility. The API C++ compliant for the CPU host and uses an extended version of C on the device clients. The tool chain includes a special version of the Open64 compiler for compiling the device code to Parallel Thread Execution (PTX) assembly which is assembled into machine code by the assembler (ptxas).

Two modes are specified: a low-level driver API and a high-level runtime API. The latter is an interface for the former to simplify the model and the language syntaxes. Since the runtime API provides an extra layer it performs slower than its counterpart, therefore it won’t be used. All references to CUDA in this thesis implicitly mean the CUDA driver API if nothing else is explicitly stated.

A schematic view of the full data allocation procedure is presented in Figure 4. The primitive unit of execution in CUDA is software threads which are abstractions of the executed hardware threads in every SP. There’s no equivalent to CPU thread affinity masks, instead threads are created, scheduled and deleted by the hardware. Threads are grouped into 1-, 2- or 3-dimensional blocks which can be of any size inside the boundary of hardware memory, possibilities of further clustering is available through grids which are collections of blocks. Each block is mapped into warps which are groups of 32 threads. If a block contains 128 software threads, four warps will be created. The warps of each block are managed by one SM, which schedules them individually. Multiple warps are managed equivalently to CPU multi-threading, where contexts are constantly exchanged. However due to the SIMD architecture of GPUs the context of warps, i.e. the threads contained, is executed concurrently. As every SM contains eight SPs that perform four passes each SP is assigned multiples of four threads. Single threaded blocks provide, like vertex shaders, MIMD behavior where each SM executes different instructional contexts.

Since the SIMD width (warp size) of each SM is 32, blocks that aren’t declared as multiples of 32, result in non-utilized SPs. As stalls of 100s of cycles are common, due to the lack of any significant data cache, multiple contexts must be maintained per SM. With all resources shared among SPs for every SM applications have to balance the number of contexts to individual memory requirements.
Figure 4 The CUDA environment involves both hardware and software, the fundamental unit is threads which are grouped and mapped onto highly parallel processors.

A hierarchy of memories is available for the threads. Latencies and sizes vary significantly between memory types while latencies are consistent between GPU architectures. All memory types allow reads, while only the thread-, block- and global memories allow writes. For each thread a private local memory is maintained where only specific API variables are accessed. Further block-, global-, scalar- and texture memories are available, while the former are shared only among the threads in a single block. The other are shared among all threads on the GPU. The block memory offers the same latency as registers access. The texture- and scalar memories are cached and are thus capable of the same low-latency access provided that a cache miss don’t occur.

**Open graphics library (OpenGL)**

This section only considers the general-purpose aspects of OpenGL including the programmable pipeline, different shader models and its relationship with GPGPU.

Originally developed by Silicon Graphics Inc. (SGI) in 1992 OpenGL is the most established graphics library on Linux. Currently it’s controlled by a coalition of different companies called the OpenGL Architectural Review Board (ARB) to separate its development from individual company desires. The OpenGL specification is heavily specialized to graphical content management with no functionality dedicated towards general-purpose programming. It uses a pipeline model (Figure 5a) which is divided into individually managed stages that performs an extremely specialized task. Since version 2.0 the pipeline incorporates vertex, geometry⁸ and fragment programmable stages but is still largely reliant of fixed functions. Functionality like filtering is 12 to 40 times (Larry Seiler, 2008) more efficient in hardware implementations than software and thus will continue to reside as a fixed function in the pipeline. The pipeline is highly optimized thus each programmable stage’s set of capabilities is constrained by the character of its task. The varying capabilities of each stage

---

⁸ Per primitive processing is available through OpenGL extensions with hardware support.
specializes the hardware even further and complicates utilizing the GPU as a cohesive unit. Instead the stage that has the most appealing functionality is generally used solely without consideration to the other stages in the pipeline. By abstracting general concepts and mapping them to the graphical pipeline it’s possible to execute general-purpose programs through OpenGL. In its Geforce 8-series (G80) architecture Nvidia replaced the discrete design (Figure 5a) which used individual processors for each stage in the pipeline. Instead a unified design (Figure 5b) was introduced that maps shaders onto threads that can be run on a shared processor.

![Figure 5](image-url) The older discrete model of execution (a) which used dedicated hardware for each stage in the pipeline. Current GPUs have a unified model (b) which uses threads to map shaders onto shared processors.

The first stage of the pipeline is the vertex processing which executes shaders on each input vertex. The vertex processors are executed as either SIMD devices where identical shaders are executed on the vertices or as MIMD devices where each vertex is treated individually. This behavior is possible as each vertex is managed by a single streaming multiprocessor on Nvidia’s unified shader design. Each vertex inputs exactly 128 bits of data which is handled as four 32 bit vectors (RGBA) and outputs exactly 128 bits of data. Therefore n different vertices are needed to output n different elements. Vertex processors are connected to the texture cache which allows them to fetch texture data during runtime. Only available on recent hardware, the vertex processors in models previous to Nvidia’s Geforce 6-series GPU wasn’t connected to the texture cache and thus couldn’t access it. Processed vertices may be scattered (written to any location) but not gathered (read from any location), the individual vertices are executed in isolation to each other and can’t access or use vertex data that isn’t owned by them.

On compliant hardware a programmable geometry unit is available through external extensions. Geometry shaders provide similarly behavior as vertex processing with texture access, lack of gathering support and MIMD processing. However as the geometry shader isn’t constricted to single vertex output variable length outputs is allowed, where the size may be determined during runtime. The output may be scattered at any arbitrary location on the frame buffer which in turn is either sent to the rasterizer or outputted directly to a feedback buffer by bypassing the rest of the pipeline using a feature called stream-out.

Since the rasterizer is implemented using a fixed-function its functionality is hard to use in GPGPU applications. However it provides fast interpolation, clipping and culling functions which may be used for interpolating scalar or texture data and discarding unwanted data.

The fragment shader inputs fragments, created by the rasterizer, which are executed on each scalar processor on Nvidia’s architecture in a SIMD manner. Other fragments in the stream may be gathered, however there is no support for communication or synchronization between processing fragments. All GPUs support texture access for the fragment processors and provide lower latency times for memory access than other stages (Houston, 2008). Further all fragments are constrained to single element output, defined by 128 bits, which can’t be scattered.
Shaders for OpenGL are programmed in either low-level GPU assembly language or in one of the two C based high-level language Cg or OpenGL shading Language (GLSL) which are Nvidia’s proprietary language and the official OpenGL shading language respectively. All of the alternatives are closely tied to graphics operation where operations must be done on vectors instead of scalars, data must be mapped to textures and computations is achieved by drawing geometry on screen to attain full utilization.

**Coherence and consistency**

Enabling shared data is a powerful model since it provides the ability to maintain threads, but with multi-core CPUs it may also result in scaling limitations. Caching performances are dictated by both hardware and software. Limitations in hardware may lead to false sharing and updates without any order of software synchronization lead to races. These are topics which are described in the following section.

All consumer multi-core processors are symmetric multi-processors (SMP) which uses one or more shared caches for fast data transfers between cores. The SMP architecture uses unified memory access for all cores which provide equal latency through a shared bus. Data which reside in any level of a SMP hierarchy is labeled as either private or shared. Private data is only used by one core, while data which multiple cores have read are labeled shared. These caches handles both replication and migration of data, during a private read the data is migrated (moved) from the shared cache to a private cache while a shared read result in replicating (copying) the data into a private cache. Shared caches provide many benefits for multi-core CPUs but also some of its scaling limitations due to cache coherence problem (Chapman, Jost, & Van Der Pas, 2008).

The fundamental concept of multi-core computers uses cache coherence which states that data must contain the same values in every replication, with SMPs this gets non-trivial. The hierarchy structure of private and shared caches in modern CPUs allow shared data to be replicated by multiple cores and later written to by one of the cores while the other core still uses the old data. Cache coherence solves this problem by coherence and consistency. Coherence and consistency is defined by (Hennessy & Patterson, 2007) as what values can be returned by a read and when a written value will be returned by a read respectively. Intel and AMD use “snooping” for coherence which initiates a search for other instances in all other caches during a write. Other instances of the same data are invalidated which forces the next read to cache miss. Data which are private don’t only reside in the one private cache and doesn’t suffer from coherence or consistency problems.

**Coherence misses**

Memory transactions are a balance act between latency and throughput. Memory addresses are grouped into units, caches uses cache lines and system memory uses pages which are transferred in it’s entirely. The smallest addressable unit is a byte. However if every request would result in individual byte transfers the memory bus would soon be congested with unnecessary requests. Instead transactions occur on larger units, the sizes vary between hardware and operating systems but usually cache lines are 64 B and pages 4 KB. The implication of this is that a request for a byte that resides on the hard drive by one of the cores will result in a chain of transactions. First a cache miss and page fault occurs, since the data isn’t located in either the cache nor system memory. The missing 4 KB page that contains the requested byte is transferred from the hard drive to system memory and then to the cache. The cache subdivides the page in smaller cache lines and transfers the appropriate one to CPU registry.

For multi-core hardware these problems are exacerbated, because of cache coherence (discussed in the previous section). Each core must lock the cache line during a write to serialize the writes between cores. Naturally since the whole cache line is locked no other cores may update data which reside in the same cache line. Due to the “snooping” protocol data which are updated by a core result in invalidation of replications and cache misses in other core’s private caches. Multiple replications of data which are updated are called true sharing. Multiple reads by different cores on different data that reside in the same cache line is called false sharing.
False sharing limits parallelism considerably by serializing writes to cache lines and increases CPU stalls due to unnecessary evictions of cache lines. The naïve solution of putting every variable in its own cache line clearly wastes too much space. Instead grouping variables by their behavior i.e. read-only, read-write and write-only. Since read-only variables don’t transpire into false sharing these should be grouped into cache lines with the excessive space padded. Further write variables should be grouped on a per thread basis, where all data used by individual threads share cache lines.

**Race condition**

Shared data that is replicated in the cache are shared resources that suffer from memory consistency problem, which is solved by implementing cache coherence protocols. Shared software variables suffer from the same problems, where it’s possible for two or more threads to read and update a single variable concurrently. Without synchronization between the threads the result will vary depending as the instructions may be executed out-of-order an occurrence which is called races.

Races are solved by declaring mutual exclusivity to the shared variables, usually a lock must be acquired by the thread which serializes the actions performed on the data. However as variables are only shared while multiple thread contexts use it which generally isn’t for its whole lifetime protection is only necessary during writes and reads. Synchronizations of data eliminate all performance gains from parallelization while also adding execution time for lock acquiring, and thus be avoided.

With the variant behavior as the only characteristic races are very hard to detect, usually the variant behavior is detected far from the place it was generated. Further the problems aren’t isolated to multi-core processors but may arise due to multi-threading in single core processors as well.

**Deadlock**

Deadlock is a phenomenon that arises from synchronization by restricting ownership of shared resources. As a thread acquires ownership over a resource A and need a resource B to continue, another thread may create a deadlock by concurrently first acquire resource B and try to acquire resource A. None of the threads can continue since another thread holds the requested resource therefore an indefinite waiting is started. Generally deadlocking involves threading structures acquiring locks but can happen to any blocking synchronization\(^9\) like message passing or signaling.

Unlike races (see Race condition) the characteristic behavior of deadlocks is easily diagnosed. Since threads commence in an infinite waiting deadlocking never evolve into incorrect results.

---

\(^9\) One of the famous non-computer related deadlocking examples is the dining philosophers problem (Dijkstra, 1971)
Designing for the n-core machine

The problem at hand was to design and develop a software framework around CPU and GPU interactivity. The framework should divide and distribute a given set of data to all available processing cores. The cooperation between the CPU- and GPU cores should be handled in a seamless fashion, where no core idles during execution due to poor workload distribution. To reach full parallelism among the processing units the solution should be completely lock-less and independent of the underlying hardware. That means, it should attempt at preserving a linear scaling to the available cores during CPU only execution, whilst also giving a substantial boost when adding a GPU. The target platform is consumer level hardware running version 2.6 of the Linux kernel.

As both modern CPUs and GPUs are divided by increasing amount of cores the distribution of workload becomes the central piece to remain fixed efficiency. The load should ideally be divided equally among the different processing units for the lowest execution time. Further, to not burden the total execution time the overhead for handling the GPU data must be taken into consideration, as it resides off-chip. While the processing time generally is lower for the GPU than for the CPU, neglected overhead lead to a significant increase in total execution time. The overhead for data intended for processing on the GPU includes transfers to and from the device and also initiating and spawning the threads. The prime limitation for GPU computing on consumer level hardware, thus also the largest challenge, is the X window processing timeout. For GPUs with a monitor attached to it, all device processing are restricted to less than 5 seconds or the window system will terminate the process. This limitation affects all compatible operating systems. Since the quantity of processed data is restricted by a fixed time limit instead of on-board memory the processing time is considerably decreased.

The final design and implementation incorporates a mixture of the algorithms provided by (Michael & Scott) and (Sutter, 2008). As data is generally loaded from hard drive a single producer/single consumer approach is sufficient and preferred. Hard drives are most efficient during single thread access due to unnecessary head movement during multiple thread access. The single consumer/single producer design restricts the access by design to only allow one thread access. Further, with loaded data being distributed to multiple execution units it’s necessary to provide access to multiple consumers and multiple producers. Each core is assigned a specific queue from where its data is taken. Special loader threads access the hard drive and distribute the data evenly among the queues. Processed data are moved from the individual queues and saved to the hard drive. As queues get emptied the core checks the other queues for loaded data while terminating if all data have been processed. The specific details of the actual implementation will be discussed in the following section.

The overall software design presented in this thesis is data independent as it handles data by bits instead of type. As the code implementation strictly follows the design it uses a data independent base. For testing purposes the code was added with encrypting /decrypting functionality.

Memory protection

To eliminate all occurrences of races and hazardous data, shared memory that is read and written need to be protected with exclusive accesses. Generally a lock of some form is initiated and acquired during reads and writes. Locks guarantee exclusivity which protects the memory from other threads reading and writing to the data, but it also guarantees sequential access. While acquiring and releasing locks are expensive they eliminate parallelism during the execution of the locked region. The solution is to reduce the areas which are affected by the locking. This instead increases the total overhead cost as locks are acquired more frequently.
Non-invasive defense
A different approach toward protecting memory from concurrent access is using atomic instructions for non-invasive behavior. While guaranteeing the same memory exclusivity as locks, atomic instructions don’t lock entire region, thus aren’t victims to deadlocks and sequential regions. Only a small subset of the Intel 64 and AMD64 instruction sets are atomically executable. The atomic instructions must be preceded by the prefix “LOCK” which engages exclusivity during reading and writing. The largest restriction to lock-less designs is the small operand sizes for the instructions, which limits its usage to a small number of structures.

As previously stated in the specification for this thesis is that the implementation should use a completely lock-less approach. In the following section I’ll describe the specific details of the memory protection scheme of the design for this thesis.

The fundamental building stone for the queue design is an instruction called cmpxchg which is a contraction of compare and exchange. Cmpxchg have three operands which are used throughout its two stages. The first stage compares a set of registers to the contents of a memory address. If the two match the zero flag is set and the instruction moves in to the second stage which stores a third operand into the memory address. Instead if the two are unequal, the zero flag is cleared and the compared registers are loaded into the memory address. The functionality is used to swap memory addresses which connect the data encapsulations. The usage of the instruction introduces a new problem which may occur only during either multiple consumer or multiple producer accesses.

In a nearly empty queue which consists of an element “a” that points to an element “b”, is accessed by multiple threads. A first thread is set to dequeue the head element “a” and re-point the head pointer to the address of the last element, “b”. However, naturally there’s a delay before the thread may actually initiates the compare and exchange instruction. During that delay a second and third thread may dequeue data element “a” and the last queued element “b” respectively. If then an element “c” and the previous element “a” is re-queued, the first thread will destroy the queue as it continues due to the unnoticed swap. Since it is instructed that “a” is followed by “b” the head pointer will be re-point to the non-queued element “b” instead of “c”. This problem is called the ABA-problem.

The solution is non-trivial, each node is assigned a variable which counts the number of accesses. This variable must be accessed concurrently with the pointer to be effective. As the AMD64 instruction set uses 64-bit addresses for memory pointers, the need of a cmpxchg instruction version which handles two 64-bit memory pointers, or 128 bits. The 128-bit version is called cmpxchg16b and uses the RAX-RDX instead of the EAX-EDX registers. Unfortunately the 128-bit version of cmpxchg, cmpxchg16b, is relatively new and wasn’t supported by the earliest AMD 64-bit processors. However, through the CPUID instruction it’s possible to check whether the instruction is supported, which is sufficient and necessary to ensure compatibility. The 128-bit version of cmpxchg is only available in the assembly instruction set, without any representation in the form of a compiler intrinsic. For this specific reason all queues and queue functionality were build using GAS during the code implementation.

All of the file handling is done by a single producer/single consumer queue (see Figure 6), where a thread loads the demanded data from disk and queues it. Another thread may then split and distribute the queued elements to the appropriate processing element. As accesses are restricted to single threads ABA-counters are unnecessary, and the queue functionality is performed using the 64-bit version of compare and exchange. The basis for the queue is a modified version of (Sutter, 2008) and follows the basic single linked list pattern with head and tail pointers. While the consumer pushes a splitter pointer forward to demonstrates the line between processed and non-processed elements.

---

10 ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR
Each thread is assigned a processing queue (threadQueue) at program initiation, to minimize contention between threads multiple queues are used. The nodes are constructed of necessary components to process its data; like the pointer to the raw data and its size the file handler. Further, contrary to the single access version of the queue the next element pointer is moved out of the element body and combined with a aba-counter. As displayed in Figure 7 this agreement applies to all reference able pointers of type (threadQueue).

The base algorithm used, was first presented in (Michael & Scott). While that version is entirely built for 32-bit processors it had to be edited to work in a 64-bit environment. The nature of the compare and exchange algorithm requires that the next element pointer and the ABA-counter are read at the same instant. If this isn’t obeyed other threads may change the values during the interval between the two reads. Due to the combined size of 128 bits the general-purpose registers, that are 64 bits, are inadequate. Instead the streaming registers were used which are 128 bits. As the SSE extensions provide great functionality for comparison, packing and loading, it was an ideal solution. Further, to handle the 128-bit compound of pointer and counter the cmpxchf16b instruction was used for swapping node elements.

**Figure 6** The file queue is restricted to single thread access, the consumer moves the splitter pointer as elements are processed. Single element objects are called FileNodes (FN) and contains a pointer to the next element and also pointers to the file meta data.

**Figure 7** As it’s required to allow multiple threads to access the same queue, a multiple consumer/multiple producer queue is necessary. The queue may suffer from the aba-problem whereas a special counter was introduced with each reference pointer of the queue type ThreadNode (TN). The consumers and producers dequesues and queues respectively through a shared interface which protects memory corruption.
Work stealing

To distribute the load equally across all cores a load balancer is used. It uses an algorithm called work stealing (see Figure 8) where processing units first are assigned specific queues at initialization. Further as queues are emptied their associated cores may take work off of some other queue. Different schemes exist for how work elements are used among processing units (PU). The implementation for this thesis uses a neighbor scheme. The PUs first take work from the next queue and iterates through each queue as they are emptied; PU 0 from PU 1 and later from PU 2 etc. This applies both to cores of CPU and GPU type since the GPU queue is non-different from the others. As showed by (Wierman & Neill) it provides very good performance when queues have low variability together with low affinity overhead. Affinity overhead arises as data are migrated between caches, like false sharing, and reduces performance due to invalidation of cache lines.

While a single globally shared queue would be sufficient to distribute the workload among the processing elements the solution isn’t as efficient as the multiple queue setup used in the design proposed in this thesis. Although there’s no formal queue ownership in the design, there’s always an attachment present between the allocated queue and the processing element. This is because the work distribution algorithm minimizes work stealing. The constant connection enables the processor to efficiently optimize latencies as data can be preloaded and remain in cache with minimal waste. Further, multiple work queues is an effective measure against contention between processing elements.

Figure 8 The work load balancing scheme uses the work stealing algorithm. Each processing unit takes work from their nearest neighbor as their own work queue is emptied. This ensures the lowest execution time since cores never idle. In the figure above queues 1 & n-1 are empty so instead of idling until new data is loaded they take work from non-empty queues.

Distributed computing

Because of the off-chip nature of GPUs its data need to be continuously streamed from system memory. As this migration is expensive, a special dequeeuer is used to bundle the data before transferring it.

The available selection of supporting structure on which to base an implementation on shouldn’t be limited by the design. Instead it should be based on the range of hardware compatibility and ease of development that’s required. CUDA offers a C based implementation environment which lowers development time. As CUDA is a hardware architecture only devices that incorporates it supports it thus no backwards compatibility for older GPU architectures exists. Moreover CUDA is developed and maintained by a single developer, Nvidia, thus is and will only be supported by Nvidia products. The maximum screen drawing interval of 5 seconds used by operating systems on hardware setups featuring only a single GPU is perhaps the largest problem for CUDA as it significantly decreases performance.
OpenGL delivers some form of compatibility with all GPU irrelevant of origin and is natively supported by all major operating systems. However the model for development is highly based on graphical abstractions thus isn’t suited for general-purpose computations. Its architecture is based on extensions which continuously extends the core set of instructions as new hardware is developed. These extensions can be exploited to reach full compatibility by segmenting the hardware capabilities and level the implementation accordingly.

**OpenGL**

As noted previously, one of the major problems when integrating the GPU into consumer programs is compatibility. The lack of a standard divides the support for hardware both by brand and card capabilities. However with OpenGL multiple shaders can be used with varying specifications to create layered compatibility. The implementation for this thesis uses the following extensions as oppose to the OpenGL library:

- EXTmultidrawarrays
- NVgupuprogram4
- NVparameterbufferobject
- NVtransformfeedback
- NVvertexprogram4

The original vertex program specification, created by the OpenGL architectural review board (ARB) in 2002, wasn’t developed for general computing and thus is very sparse on functionality. To provide some of the basic functionality; like bit-shifting, subroutine calling & conditional operators, some extensions were used.

All of the basic controlling functionality and shader instructions are specified by the GPU program 4 extension. It provides a very broad and generalized set of instructions which cover vertex-, geometry and fragment shaders. With the vertex program 4 specification, relevant vertex shader specific functions are offered. Its main functionality used in the implementation is the native integer support for attributes, which passes the attributes as integer instead of having to convert them to float.

The data input is handled by functionality provided by the parameter buffer extension. It divides a section of data into 128-bit fractions which are passed to the vertex processors for processing. The transform buffer records the output from the vertex primitives. As the rasterizer converts the vertices to fragments the output is right after the geometry shader. While turning off the rasterizer, half of the pipeline is ignored, thus the data doesn’t need to pass all of the stages. The recorded data is stored as linear memory and thus can be access by either pointers or converted to arrays.

**CUDA**

Since the timeout limit of five seconds which is set by the X window system (X11) the amount of data which may be processed is significantly decreased. Instead of stream process the assigned data from beginning to end the device must return control to the host with five second intervals, or the process will terminate the process and return an error (number 702). This only apply to graphics cards with monitors attached to them, thus computers with multiple graphics card that isn’t linked through Scalable link interface (SLI) or Crossfire are free from this limitation.

Each GPU is specified with a capability number, a similar initiative as the CPUID instruction. The number describes what features that are available. Further, the GPU provides information about the hardware layout: like the quantity of streaming multi-processors, maximum number of threads per block, available shared memory, core speed etc. To ensure capability among different computers with varying specification it’s required to query the device for this information and adapt the application appropriately. As presented in the analyzing section the execution time is lowest while the block size is a multiple of the SM quantity and the spawned threads are multiples of the SIMD width.
Symbiotic integration

This section and its subsections contains many references to the code of the implementation part for this thesis which is available in the form of either a complete download or as a listing of the header files in appendix D.

The advent of parallelism granted tasks to be executed simultaneous and effectively replaces concurrency as a measure for expressing groups of worker threads. Moreover concurrency which is simulated by operating system kernels with context switching, inherently executes sequentially. Therefore it suffers from insufficient utilization and excessive overhead during execution on multiple cores. While concurrency remains a central component for processors it doesn’t expose parallelism and is very ineffective for multi-processing.

Parallel performance is the product of various entities many more than concurrent programs which arguably is limited by pipeline length and core frequency speed. Instead performance of parallel programs is more complex. The increasing core density on each processor decreases the individual bus bandwidth to external devices\(^\text{11}\). The immediate effect is that improved thread efficiency throttles bandwidth, by increased memory requests, and decreases performance. Modern software designs therefore have to maintain a constant balance between excessive threading and draining the processing pipeline through resource management and processing entities. Further, data inconsistencies for multi-core processors due to cache coherence and processing delays due to bus latencies for GPU computing are multi-core and many-core specific issues that must be considered during deployment. Combined with the eclectic nature of modern computer components it creates an unmanaged environment for conventional program design most likely found in today’s software. These broadened processing characteristics further expand the necessity for many-core purpose designs. Program designs of such should be general enough to justify its usage although sufficiently defined to retain its native optimization advantages.

An effective design must actively change the underlying application structure, consider and resolve performance issues that may arise before or during processing to contrive the best foundation without any significant code restructuring by the implementer. This usually takes the form of lowering inter-communication which is a response to sharing. Further application specific bottlenecks which originate from blocking and data arrangement are harder to control and can only indirectly be influenced by the design. To minimize their influence on the final performance blocking is eliminated as have been presented previously in this thesis. Instead private resource ownership is handled by transactional non-invasive instructions. This approach reduces both bandwidth usage and stalls as data sharing is minimized.

The main intent for a parallel software design is to strive for full utilization of the available hardware during deployment without compromising performance. As computing isn’t inherently parallel - task segregation becomes fundamental and necessary as hardware scaling is achieved horizontally by the addition of processing entities. It’s the degree to which tasks are sundered that dictates the terms for the simultaneous executed throughput. Although sundering is essential to maintain performance, extended segregation increases contention among threads which introduces superfluous overhead and idling. This disparity between segregation and processing overhead is the cause to why parallelism is hard to exploit, moreover it’s the reason for its potential.

---

\(^\text{11}\) AMD 64 and Intel i7 processors uses integrated memory controllers which eliminates bandwidth restriction to memory.
Design reusability

The design uses a component structure from which parts may be substituted with minimum effort. Although there exists many multiples of smaller components there are four major distinctive components which are relevant to present. The four components are listed below.

- Processor
- Loader
- Thread management
- Load balancer

In brief the program functionality is largely defined by the capabilities of the executing processor. Depending on the number of cores that are available an appropriate number of threads is created and attached to the cores. The loader handles communication with the hard drive from which data is extracted, the loaded files are then further segmented and queued. Threads are handled in a separate component that first performs some hardware analysis to start the appropriate number of threads for each device. The load balancing functionality is strongly coupled with the processing. This is because the independence of each processing unit, as each is very specific to how its processing is achieved individual balancing is needed.

Whether a program, based on legacy code or not, uses the fundamentals of the design proposed in this thesis certain program specific attributes must be exposed. Some form of intensive work by either data, processing or both is needed. Further, it’s necessary for them to have some leverage as the overhead from the parallelization for small processing jobs stiffs performance. To reach full utilization of the hardware the processing of data elements can’t be tightly coupled but instead independent of sharing constraints.

Normally the only component that needs a functional definition by the implementer is the processor, which performs the computing. To accommodate full freedom of general program designs the parts which relates to processing are independent of both hardware and the remaining program. Therefore any combination of the supported processing structures may be used. Generally only the one device implementation is needed, however to reach full compatibility multiple is required with some instances (e.g. CUDA).

No prerequisites are made for implementations following the design specification proposed in this thesis. With the common base for all processors hardware may be exchanged, added or removed at will without any serious consequences except changes in performance. The final implementation may utilize only the processing units which are found useful. At a later instant the remaining unused processing power may be enabled by adding a single line of code, given that an appropriate processing implementation is supplied. By attaching the device processing specific code to the particular control code (i.e. processor.c, Gcontrol or GGcontrol.c) and enabling its usage the program can start to utilize the device.

Due to the contrary specifics among processing elements and supporting structures the interface isn’t completely consistent. Whereas a common interface would unify the processors and ease development the performance trade off would be exceedingly high as only the common ground would be exposed. For example the CPU processors loads data separately on demand as it’s needed. This isn’t a viable technique for external devices residing off-chip, the latency for each data element travelling on the bus separately would be exceedingly high. Instead the program transfers large chunks of data to local device memory.

Implementations of processing functionality is only restricted to the constraints of the supporting structures and is thus independent of programming language\textsuperscript{12}. CUDA is natively compiled to C++ but may use any programming language. OpenGL only supports native assembly, Nvidia Cg (C for graphics) or GLSL (OpenGL shader language) code. With both CUDA and OpenGL having cross-platform support only the CPU threading code isn’t directly portable due to the inconsistencies between threading libraries among different operating systems. This limitation is expected to

\textsuperscript{12} A prefered workflow scheme is presented in Figure 9.
disappear for implementations that exploits the shared threading platform in C++0x\textsuperscript{13} when it’s released. As for implementations of the complete design proposal it’s nearly programming language independent. As previously described (see Non-invasive defense) non-invasive functionality depends on hardware. Whereas much of the hardware instructions are supported either natively or by compiler intrinsic cmpxchgl6b isn’t by any compiler, thus 64-bit implementations are required to use assembly language for memory protection. The effected section is the multiple consumer/multiple producer queues which uses cmpxchgl6b as measure against ABA-issues. This only applies to x86-64 operating systems running both 64-bit and emulated 32-bit programs as they use pointers which are 64 bits wide.

**Details of design implementation**

Binding CPU threads and GPU threads together require a compound of programming languages, compilers libraries and assemblers. This section explains the workflow layout for the implemented program.

The overall schematic is presented in Figure 9, which divides the workflow by device and host. Two fundamentally different approaches where used for the GPU device, a low-level and a high-level solution. The high-level implementation was programmed with CUDA which, based on C, provide a very abstract layer for hardware interaction. As CUDA was designed from the start to streamline the GPU workflow it doesn’t provide any reasonable optimization options or alternative solution approaches. The CUDA specific device executable code was coded using the driver API which was compiled with Open64 (aka NVCC) and further optimized with Nvidia’s PTX language and assembled using PTXAS. However due to some inconsistencies in the PTX language, which is as of this writing under development, optimizations where few and far between. Since program uses the driver API the code were assembled into device specific binaries directly and dynamically linked at runtime. Communication between the device- and host code is handled by C++ and the CUDA driver libraries.

The second approach for GPU execution was OpenGL. Instead of using a shader language like Cg or OpenGL shader language (GLSL) the native assembly language was used for all device code. By only executing the code in the vertex shaders the output data could be streamed to main memory directly using transform buffers. Instead of having data passing through the entire pipeline data is streamed out before the rasterizer, see Figure 5a. As this approach allows half of the pipeline to be shutdown the resulting performance is increased. In conjunction with the OpenGL specification the shader code is compiled and linked at runtime by the driver, however the code is prepared at compile time using the GCC preprocessor.

The base for all host code was GAS (GNU Assembly) which handles all queue-, compatibility, and processing functionality. Further, C was used to control the thread management and necessary language specifics. Compilation and assembly is handled by GCC and GNU assembler respectively, while GNU Id was used for linking the code to a 64-bit binary ELF executable.

The workflow was chosen solely for its flexibility. Managed languages like JAVA and C# with .net may provide better ease of use for both threading and library functionality than C and GAS. However, their nature don’t allow for low-level optimization, assembler extensions and provide subliminal performance due to many reasons but primary their garbage collectors and non-native execution. For these reasons they were never an alternative. The same reasons apply to device coding. By using native assembly language instead of high-level shader languages the code isn’t restrained to their implemented functionality.

The code implementation uses an encrypting and decrypting scheme to simulate heavy processing. An algorithm called DES encryption is used, which may offer poor security due to its limited key space of 56 bits. However the heavy use of both arithmetic computation and constant values provide a good common processing base for simulation. The algorithm is broken down into five separate steps, where the first two manages the keys, the third prepares the input and last two encrypts the text using the keys from the first steps.

\textsuperscript{13} Only a draft of the C++0x has been relased by the ISO/IEC C++ committee.
The CPU and CUDA implementations are similar in that they process 64 bits in sequence per core. While the CPU is constrained by its architecture which processes data sequentially the CUDA implementation is restricted by the syntax which doesn’t allow concurrent data execution. Instead OpenGL allow each core to execute data concurrently in 4 by 32 bits (2 longs).

**Figure 9** The workflow for the implementation uses a low-level approach with established components. The host code uses GAS(GNU assembly) for all functionality while C is used for thread management. This combination proved to be very effective as the functionality for queues and encryption are of most need for optimization which is only possible in assembly. The alternative would be to write all code in C and compile it to assembly and optimize that code which is much more tedious. As CUDA natively require C++ the control code was written in C++ and linked together with the other code. Two solutions for device computation was explored, CUDA provided a high-level solution and OpenGL a low-level solution. The device code for the OpenGL version uses vertex assembly language while the control code uses C. The main reason why vertex assembly was used is that it provides the smallest layer between the code and hardware.
### Analysis of results

The performance goal for the implementation of the design was to retain a linear increase to the quantity of cores and provide a significant boost as a GPU is integrated into the system.

All of the results in this section were rendered using the hardware and software found in Table 2. As the CPU derives off of the Intel Penryn design it’s not a strict quad core CPU with four individual cores with separate caches. Instead Penryn fuses two strict dual cores into a single core, the result is a split L2 cache where each part is shared by only two cores. The resulting performance is reduced because of the non-shared cache. The CPU gets more sensitive toward cache coherence, as data must use the bus when migrating.

<table>
<thead>
<tr>
<th>Hardware specification</th>
<th>Software specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong> Quad-core CPU @ 3.0 GHz</td>
<td></td>
</tr>
<tr>
<td>‣ 8x32 KB L1 (i+d)</td>
<td></td>
</tr>
<tr>
<td>‣ 2x12 MB L2</td>
<td></td>
</tr>
<tr>
<td>‣ 1333 MHz FSB</td>
<td></td>
</tr>
<tr>
<td><strong>GPU</strong> Nvidia 8800 GT</td>
<td></td>
</tr>
<tr>
<td>‣ 14 SMs</td>
<td></td>
</tr>
<tr>
<td>‣ 8 SPs / SM</td>
<td></td>
</tr>
<tr>
<td>‣ 2 SMs / TPC</td>
<td></td>
</tr>
<tr>
<td>‣ 1.5 GHz core</td>
<td></td>
</tr>
<tr>
<td>‣ 512 MB DRAM</td>
<td></td>
</tr>
<tr>
<td><strong>Operating system</strong> Linux version 2.6.22.18 x86-64</td>
<td></td>
</tr>
<tr>
<td><strong>GPU driver</strong> Nvidia 177.67</td>
<td></td>
</tr>
</tbody>
</table>

Table 2

Figure 10 presents the performance data of the implemented encryption scheme. The same files where loaded into memory and encrypted with a varying number of cores active. Because of a slightly larger overhead for the single threaded run there’s some noise in the result. For the CPU cores a clear linear trend is presented, where the performance increases with core quantity. When the GPU runs CUDA code together with 4 cores a significant boost is presented with an execution time of over 12x to the single threaded result. A GPU timeout were active during these tests which limited the GPU to process the data in 45 MB windows. Larger batch sizes than 45 MB results in an execution time of more than 5 seconds and are aborted. With an increase of 30-40 times the single threaded version and 4 times faster than the CUDA version the OpenGL implementation shows the GPU performance superiority to the CPU.

As OpenGL performs on a per call basis where a batch of data is transferred to device memory the resulting performance become linked to the size of the batches. In Figure 12 the performance is analyzed to the size of each batch size. Further, Figure 12 confirms what this thesis is all about that the only limitation to a software design is hardware. As expected the execution time is highly dependent of the file size (see Figure 11). For smaller file sizes the differences between the techniques are indistinguishable. However as the file size increases a rapid growth for the CPU execution time can be seen. As CUDA is constantly faced with a limiter of 5 second executions the software design is irrelevant. Only a very small increase in performance is provided on faster hardware, as the limiting factor lies in the OS and not in hardware. However the test results for the OpenGL implementation present that the performance of the program design is linked to the underlying hardware. Low-end card with small arrays of data may provide 7x or 8x performance increases but as the code is executed on cards with more memory the performance is increased. The program spawns and attaches one thread per active core plus one for GPU management and two for hard drive access and queue management.
To increase throughput and minimize load balancing all input files are divided into smaller chunks if the combined file size is larger than a threshold value. These chunks are then passed to the load balancer which queues them in the available queues. As processing units only are allowed to dequeue elements wholly, the chunk sizes are reflected on the final execution time. The balance act between thread contention and overhead is presented in Figure 13. As the chunk size increases the contention and processing unit idling increases with it. Since the GPU provides most of the finished work the chunk sizes should be reduced to a relatively small size. Because as the distributed chunks gets larger the CPUs dequeues more data than necessary so the GPU must idle while waiting. This problem is visible in Figure 13, as chunk sizes rise so does execution time.
Figure 10 The performance figures presented above are increase in performance in conjunction to the single threaded CPU performance time. An encryption scheme using DES was used on 9 files where one file contained ~530 MB of data while the additional 8 files where small text files. The linear performance figures for the CPU show that the program design is linked to the hardware. Both the CUDA and OpenGL numbers show the superiority of the GPU to the CPU. The impact on having the CPU running concurrently with the GPU give a very small to zero increase of speed as the GPU does most of the processing. The non-normalized performance data can be found in Appendix B.

Figure 11 The numbers shown in the graph above was conducted from encrypting various files of varying sizes. As the file sizes grow so does the difference between the CPU and GPU time. To minimize the impact of hard drive access time all files where loaded into main memory before encryption.
OpenGL operates on chunks of memory which are transferred to device memory at runtime. As is shown the software design presented in this thesis is only limited by the hardware by which it is run on. The test system ran out of memory at 1380 slots per call which also proved to produce the fastest time. The speed increase is a result of the decrease in transfer overhead as the batch sizes grow. While using a device card with more memory the performance is expected to continue to increases due to the shape of the curve. The non-normalized performance data can be found in Appendix B.

![OpenGL performance over memory size per call](image)

**Figure 12** OpenGL operates on chunks of memory which are transferred to device memory at runtime. As is shown the software design presented in this thesis is only limited by the hardware by which it is run on. The test system ran out of memory at 1380 slots per call which also proved to produce the fastest time. The speed increase is a result of the decrease in transfer overhead as the batch sizes grow. While using a device card with more memory the performance is expected to continue to increases due to the shape of the curve. The non-normalized performance data can be found in Appendix B.

![Contention](image)

**Figure 13** Each loaded file is split up into smaller batches if the total file size is larger than a threshold value. The graph above presents the time to encrypt 530,6 MBs of data as the chunks size for each file varies. The purpose of the graph is to map the contention between the CPU and GPU. While very small sizes show how large the overhead is; very large values present the contention. As some of the high values for the CUDA version couldn’t produce a value due to the 5 second x window timeout some values from its graph are absent. As the CPU mainly works in a sequential order its time is fairly constant with only a small variance for the extreme values. The rapid increase in execution time for the OpenGL version at very large sizes is due to the CPU shortcomings. The data sizes for each processing run gets larger as the batch size increases and so the GPU must wait for the CPUs to finish their data chunks.
Discussion

The developed design scheme for acknowledging the GPU as a viable processor, by integrating it with the CPU, has proved to be efficient and effective. It provides a uniform environment without any special requirements, where all cores are used as processing units and data is managed without locks. Through the usage of the compare and exchange (CAS) instruction most of the overhead from locking could be reduced. Alternatively the LL/SC (load linked and store conditional) instructions may be used in a similar manner on architectures where CAS isn’t supported, since it offer the same functionality. Further, the usage of multiple queues lowered thread contention while also introducing processor idling due to the quantity of queues. With the introduction of a load balancer which used work stealing full efficiency was possible.

By integrating the design with three different platforms the viability of the design could be tested, limitations searched and performance measured. To introduce some functionality to the design implementation the encryption algorithm DES was used. Apart from its rather unsecure algorithm, featuring a 56-bit key, it proved to be very computational intensive which was the sought after feature and provided a good example for real-world applications.

The predefined goal was set to deliver by the quantity of processor cores a very realistic goal given the theoretical performance limitation of multi-core CPUs. With the complexity and diversity among GPU architectures, performance goals are problematical to set for many-core enabled programs. However with its large performance capability a considerable boost was to be expected. Instead the ambition was to reissue the hardware as the limiting factor for scaling, reinstating the free lunch.

Performance wise the design works well, scaling linearly giving almost 100% parallelism. The performance for the OpenGL version of the implementation flattened not showing any impact as cores were enabled. Instead of being considered a design flaw it confirms the superiority of the GPU. This was also proved by Figure 13 as it gave the CPU more influence by increasing its data quota, when the quota increased performance decreased. All in all the combined speed increase as all available cores were enabled was over 4 times faster than a single core processor. As tests showed, the performance outcome was heavily dependent of the quantity of device memory; with the lowest execution times created as all memory was accessible. This dependency confirmed that the limitations of the software lay in hardware. Instead of being constrained to a predetermined set of threads the software adapts and scales by the capabilities of the hardware.

GPU scaling is hard to measure given the restrictions that the software is constructed on. While CUDA suffers from the 5 second time out OpenGL relies on vertices to represent data processing elements. Therefore to measure scaling in CUDA the quantity of used streaming multiprocessors would have to be changed through variations of both the quantity of threads and blocks. However as cores are disabled execution time runs to long and is aborted by the processing time frame. OpenGL uses data windows which are mapped to vertices, and disabling any one of the vertices produces an incomplete result as some data isn’t processed. Instead scaling measurements are done through varying memory which is shown in Figure 12 and Figure 13.

Some clear limitations to performance are presented by the test results. Most obvious is device memory which was limited to a total of 512 MB. With more memory performance would have gone up, which is shown by Figure 12. Further, the 32-bit native device of the test system isn’t an ideal fit. Many of the encryption computations are required to be performed in multiples of 64 bits, with the lack of a native 8-byte variable type performance suffers. The limitations come from overhead of sequentially processing 32 bits which suffer performance. While current GPUs supports double precision, performance is expected to increase as these devices are used.
The many-core prototype defined in this thesis strictly conforms to the profile of a mainstream consumer which don’t have multiple GPUs, processor sockets or even a predefined specification for hardware. None of the technologies which were used in the implementation was created specifically for accelerating consumer software. CUDA is primarily a small high performance computing (HPC) framework targeted to scientific application, while OpenGL is a real-time graphical API. Both provide an unreachable layer of performance for common application used by mainstream users. Only by using a software design which unifies their abstractions with general-purpose computing of regular programming APIs a common platform for developing general-purpose instruction is attainable.

**Further development**

To extend the support for different hardware additional development and optimization are in place. A 64-bit port of the device code could significantly increase performance for cards that support it. It would require the current 4x32 bits scheme to be changed to 2x64 bits.

Much of the compatibility for hardware with low capabilities isn’t currently supported due to project time restrictions. However full range support is desired and capable by the design scheme.
Bibliography


Hensley, J. (2007). Close to the Metal. GPGPU.


Appendix A System requirements

The specified versions are the only versions that the program is guaranteed to work with, the program may compile with either older or newer version however this haven’t been tested.

Hardware

<table>
<thead>
<tr>
<th>CPU</th>
<th>Graphics card</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 based CPU with 64-bit extensions, also support for SSE, SSE2 &amp; SSE3 instructions is required.</td>
<td>An OpenGL shader model 2 or newer compatible graphics card for OpenGL optimizations. Further the following extensions must be supported via hardware:</td>
</tr>
<tr>
<td></td>
<td>› EXTmultidrawarrays</td>
</tr>
<tr>
<td></td>
<td>› NVtransformfeedback</td>
</tr>
<tr>
<td></td>
<td>› NVparameterbufferobject</td>
</tr>
<tr>
<td></td>
<td>› NVgpuprogram4</td>
</tr>
</tbody>
</table>

An Nvidia GeForce 8 (G80) or newer graphics card for CUDA optimizations.

Table 3

Software

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Compilers/Assemblers/Linker</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 2.6.22.18 x86-64</td>
<td>GNU assembler 2.17.50 GNU ld 2.17.50 GCC 4.2.1 Nvidia PTXAS release 2.0 Nvidia NVCC release 2.0 Trolltech QT MOC 59</td>
<td>Nvidia graphics driver 177.67 OpenMP 3.0 Nvidia Cg 2.1 OpenGL 2.1 Trolltech QT X11 4.4.1 opensource ed. IEEE POSIX 1003.1c standard (1995)</td>
</tr>
</tbody>
</table>

Table 4
Appendix B Performance data

The data was taken as a direct output from the implementation following this thesis. The performed task used a DES encryption on 9 files with a combined size of 530.6 MB.

<table>
<thead>
<tr>
<th>Hardware specification</th>
<th>Time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>One CPU core</td>
<td>764</td>
</tr>
<tr>
<td>Two CPU cores</td>
<td>350</td>
</tr>
<tr>
<td>Three CPU cores</td>
<td>234</td>
</tr>
<tr>
<td>Four CPU cores</td>
<td>174</td>
</tr>
<tr>
<td>Only GPU(CUDA)</td>
<td>77</td>
</tr>
<tr>
<td>One CPU core + GPU(CUDA)</td>
<td>81</td>
</tr>
<tr>
<td>Two CPU cores + GPU(CUDA)</td>
<td>65</td>
</tr>
<tr>
<td>Three CPU cores + GPU(CUDA)</td>
<td>69</td>
</tr>
<tr>
<td>Four CPU cores + GPU(CUDA)</td>
<td>77</td>
</tr>
<tr>
<td>Only GPU(OpenGL)</td>
<td>21</td>
</tr>
<tr>
<td>One CPU core + GPU(OpenGL)</td>
<td>21</td>
</tr>
<tr>
<td>Two CPU cores + GPU(OpenGL)</td>
<td>20</td>
</tr>
<tr>
<td>Three CPU cores + GPU(OpenGL)</td>
<td>19</td>
</tr>
<tr>
<td>Four CPU cores + GPU(OpenGL)</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 5 Raw performance data for Figure 10.

<table>
<thead>
<tr>
<th>File size(MB)</th>
<th>OpenGL</th>
<th>CUDA</th>
<th>4 CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.5</td>
<td>0.4</td>
<td>1.5</td>
</tr>
<tr>
<td>2</td>
<td>0.4</td>
<td>0.9</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>0.9</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>0.5</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>0.6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>76</td>
<td>3</td>
<td>12</td>
<td>29</td>
</tr>
<tr>
<td>115</td>
<td>5</td>
<td>18</td>
<td>44</td>
</tr>
<tr>
<td>186</td>
<td>7</td>
<td>30</td>
<td>71</td>
</tr>
<tr>
<td>373</td>
<td>15</td>
<td>59</td>
<td>144</td>
</tr>
<tr>
<td>530</td>
<td>19</td>
<td>77</td>
<td>174</td>
</tr>
<tr>
<td>1012</td>
<td>44</td>
<td>160</td>
<td>390</td>
</tr>
<tr>
<td>1536</td>
<td>123</td>
<td>242</td>
<td>587</td>
</tr>
</tbody>
</table>

Table 6 Raw performance data for Figure 11.

<table>
<thead>
<tr>
<th>Slots per call</th>
<th>Time(seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>677</td>
</tr>
<tr>
<td>64</td>
<td>339</td>
</tr>
<tr>
<td>128</td>
<td>171</td>
</tr>
<tr>
<td>256</td>
<td>89</td>
</tr>
<tr>
<td>512</td>
<td>47</td>
</tr>
<tr>
<td>1024</td>
<td>26</td>
</tr>
<tr>
<td>1380</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 7 Raw performance data for Figure 12.
<table>
<thead>
<tr>
<th>Memory (MB)</th>
<th>4 CPUs + GPU(OpenGL)</th>
<th>4 CPUs + GPU(CUDA)</th>
<th>4 CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>20</td>
<td>61</td>
<td>193</td>
</tr>
<tr>
<td>0.25</td>
<td>20</td>
<td>62</td>
<td>194</td>
</tr>
<tr>
<td>0.5</td>
<td>20</td>
<td>62</td>
<td>194</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>63</td>
<td>194</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>64</td>
<td>195</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>NA</td>
<td>197</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>NA</td>
<td>199</td>
</tr>
<tr>
<td>10</td>
<td>28</td>
<td>NA</td>
<td>202</td>
</tr>
<tr>
<td>20</td>
<td>36</td>
<td>NA</td>
<td>214</td>
</tr>
<tr>
<td>30</td>
<td>53</td>
<td>NA</td>
<td>224</td>
</tr>
<tr>
<td>40</td>
<td>163</td>
<td>NA</td>
<td>224</td>
</tr>
</tbody>
</table>

**Table 8** Raw performance data for Figure 13.
Appendix C Data apportion

Processing data on an external device isn’t trivial. This section explains the details of dividing data for the many cores which reside on the GPU device. Much of the functionality is specific to the framework for which it is used and is therefore separated into two smaller sections.

CUDA

Data

An array of memory allocations

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | ...

Divide by block

*Each SM should be populated by at least 2 blocks to reach full utilization.*

**SM 0**

| block n/2 |
| block 0 |
| **SM 1** |
| block 1 |
| ...

**SM i-1**

| block n-1 |
| block n2-1 |

Divide by thread

*Every block is assigned a multiple of 2^SIMD width threads.*

**SIMD width**

| block 0 |
| Thread 0 |
| Thread 1 |
| Thread 2 |
| ...
| Thread C-1 |

**SIMD width**

| block 1 |
| Thread C |
| Thread C+1 |
| Thread C+2 |
| ...
| Thread 2xC-1 |

**SIMD width**

| block n-1 |
| Thread (Y-1)xC |
| Thread (Y-1)xC+1 |
| Thread (Y-1)xC+2 |
| ...
| Thread YxC-1 |

**SIMD width**

| Block (Y)xC |

C = 2^SIMD width

Y = number of blocks

Divide and access data

*The data is then divided equally among the threads and accessed in a continuous fashion. The illustrated access is coalesced into one memory transaction.*

**Transactional memory size (i.e. 128 bytes)**

| 64 bit word |
| 64 bit word |
| 64 bit word |
| 64 bit word |
| ...
| 64 bit word |
| 64 bit word |

| Thread 0 |
| Thread 1 |
| Thread 2 |
| Thread 3 |

| Thread (x2/12) |
| Thread (x1/2) |

x = SIMD width

Figure 14

The data from the input is combined into a cohesive chunk in linear memory and transferred to the device for processing. Each chunk of data is accessed by blocks which contain multiple threads. The threading functionality is handled by the thread execution manager which sets the affinity and assigns the blocks to the streaming multiprocessors (SM). To hide the latency of data latency, when reading or writing, multiple blocks are assigned to each SM. This allows the SMs to context switch between the blocks as needed.
Further, the blocks create threads, to process the data, in quantities of the core SIMD width (aka warp). Nvidia recommends developers to allocate more threads per block as it provides better time slicing per thread. However as register memory is a shared resource among all cores the register count is a result of the thread count, thus less register are available as threads are created. Blocks that doesn’t use threads in SIMD width multiples won’t reach full utilization as the unused threads idles during processing. Each thread is assigned 64 bits of data which it processes in its private memory.

**OpenGL**

**Data**

*Multiple strings of data are used as input and combined into one.*

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | ... | \( m \geq 2^{m-1} \)

**Apportion**

*The data is then divided among vertices (points)*

```c
if (mode or count is invalid)
gen appropriate error
else
{
    Begin(mode);
    for (int i = 0; i < count; i++)
        ArrayElement[first + i];
End();
}
```

**Processing & output**

*As graphics processing cores are capable of handling 128 bits each core handles 4x32 bits concurrently.*

*As the pipeline is disabled from the rasterizer and forward the output is streamed after the geometry stage. The output is stored as entries in a texture.*

![Figure 15](image)

The OpenGL implementation works differently from the CUDA implementation as it works with GPU calls rather than strict data transferring. The first step is to assemble the data which usually originates from multiple sources. The data is stored in linear memory and passed to the drawArray function which passes 128 bits of the data to each created vertex. Therefore the complexity of the drawn geometry is determined by the input data size. All variables in OpenGL handles 128 bits of data in 4 multiples of 32 and since the cores process the same amount of data it’s possible to process four elements concurrently. The output from the pipeline is streamed from the geometry shader output to the framebuffer which can be mapped and transferred to main memory.
Appendix D Program code excerpts

Declarative files
The following code segments are excerpts from the respective source files.

```
***excerpt***
char **files;

typedef struct fileQueue
{
    struct fileNode *head, *split,*tail;
}fileQueue;

typedef struct fileNode
{
    struct fileMeta *filePtr;
    struct fileNode *next;
}fileNode;

typedef struct fileMeta
{
    unsigned long size;
    unsigned long numDiv;
    unsigned long currentDiv;
    char brokenFile;
    bool end;
    FILE *fdin;
    unsigned long realSize;
    long currentLoc;
    void *startFile;
    char *name;
    char padding[cachelineSIZE - sizeof(struct fileMeta)];
}fileMeta;

extern unsigned long enqueueXchg64(struct fileQueue *fq);
extern unsigned long dequeueXchg64(struct fileQueue *fq);
extern unsigned long consumeXchg64(struct fileQueue *fq);
extern void updateXchg64(unsigned long *ptrVar, int adder);

***excerpt***
Listing 3 An excerpt from loader.h which presents the definitions for the single consumer/single producer queue.

```

```
***excerpt***
typedef struct tQueue
{
    struct threadNodePtr *head;
    struct threadNodePtr *tail;
}tQueue;

typedef struct threadNode
{
    void *dataPtr;
    unsigned long *fragmentCounter;
    struct threadNodePtr *next;
    unsigned long *size;
}threadNode;

```
typedef struct threadNodePtr
{
    struct threadNode *ptr;
    unsigned long ABAcounter;
}threadNodePtr;

extern long enqueueCas128(tQueue *tq, void *data, void *pCounter, void *pSize);
extern long dequeueCas128(tQueue *tq, void *data, void *pCounter, void *pSize);

Listing 4 An excerpt from threadQueue.h which presents the definitions for the work queues.

Synchronization code

The following code segment is the memory protection functions for the working queues. The code is programmed for the x86-64 architecture thus using 64-bit pointers and the extended instructions and registers.

###excerpt#

```
#-----------------------------------------------------#
#Name:               enqueueCas128                     #
#Description:        Atomically enqueues a given element to a specified queue. #
#                    The queue must be initialized from start with a NULL #
#                    element or the enqueueing will fail.  #
#Register usage:     # GPR       Destroys all but r11.      #
#                    # XMM       XMM1 through XMM5 used.      #
#                    # MMX       None used         #
#                    # x87       None used                           #
#Argument 1(rdi):    # Desc:  A pointer to the queue that the element should be #
#                    # added to.                         #
#                    # Type:  tQueue* (8 bytes)                 #
#Argument 2(rsi):    # Desc:  The address of the new element's data.      #
#                    # Type:  void* (8 bytes).                      #
#Argument 3(rdx):    # Desc:  The address of the new element's fragment counter. #
#                    # Type:  unsigned long* (8 bytes).               #
#Argument 4(rcx):    # Desc:  The address of the size variable for the new element's. #
#                    # Type:  unsigned long* (8 bytes).                   #
#Parameters(rax):    # Type:  signed char (1 byte)                         #
#Values:            #  0x1 on a successful enqueueing
#                    # unspecified if the enqueueing generates an error
#-----------------------------------------------------#
.globl enqueueCas128
enqueueCas128:
  movq %rdi, %r12
  movq %rsi, %r13
  movq %rdx, %r14
  movq %rcx, %r15
enqueueCas128_break:
  push %r12
```

```
push %r13
push %r14
push %r15

movq $16, %rdi #tnp
call malloc

push %rax
movq $0, (%rax)
movq $0, 8(%rax)

movq $32, %rdi #tn
call malloc

pop %r9
pop %r15
pop %r14
pop %r13
pop %r12
movq %r13, (%rax)
movq %r14, 8(%rax)
movq %r9, 16(%rax)
movq %r15, 24(%rax)

#------------------------------------------------
movq %r12, %rdi
movq %rax, %rsi
movq 8(%rdi), %rax #address of tq-tail
movq %rax, %rbx #address of tq-tail
#tq->tail xmm1=both of the TNP ptrs for queue->tail
lddqu (%rax), %xmm1
movq (%rax), %rax
movq 16(%rax), %rax
lddqu (%rax), %xmm2 #tq->tail->ptr->next hi=count lo=ptr

movapd %xmm1, %xmm3
movapd %xmm2, %xmm4
#------------------------------------------------
cmpdd $0, (%rbx), %xmm3
movhlps %xmm3, %xmm4
movq %xmm3, %rcx
jrcxz recycleJmp
movq %xmm4, %rcx
jrcxz recycleJmp

movq %xmm2, %rcx
jrcxz cas128 #jmp if tail->next == NULL

movq %xmm1, %rax
movhlps %xmm1, %xmm5
movq %xmm5, %rdx

movq %xmm2, %rbx
movq %rdx, %rcx
add $1, %rcx

movq 8(%rdi), %r10
lock
cmpxchg16b (%r10)
jmp recycleJmp

cas128:
    movq $xmm1, %rax
    movq 16(%rax), %r10

    movhlp $xmm2, %xmm4
    movq %xmm2, %rax #ptr
    movq %xmm4, %rdx #ABAcounter

    movq %rsi, %rbx #ptr
    movq %rdx, %rcx #ABAcounter
    add $1, %rcx

    lock
    cmpxchg16b (%r10)

    jne recycleJmp #loop on fail
    mov $1, %rax #else return true
    jmp forwardTail

recycleJmp:
    push %r12
    push %r13
    push %r14
    push %r15

    movq %r9, %rdi
    call free
    movq %rsi, %rdi
    call free

    pop %r15
    pop %r14
    pop %r13
    pop %r12

    jmp enqueueCas128_break

forwardTail:
    movq $xmm1, %rax
    movhlp $xmm1, %xmm5
    movq %xmm5, %rdx

    movq %rsi, %rbx #ptr
    movq %rdx, %rcx
    add $1, %rcx #counter

    movq 8(%rdi), %r10

    lock
    cmpxchg16b (%r10)

    mov $1, %rax
    ret
# Name: dequeueCas128
# Description: Atomically dequeues the first finished element in a specified tQueue. If the queue is empty or if no element is finished (head->ptr == split->ptr) it returns false.
# Register usage:
# GPR   Destroys all but r9, r11, r12 & r13.
# XMM   XMM0, XMM2, XMM3, XMM4, XMM5, XMM9 & XMM10 used.
# MMX   None used
# x87   None used

# Argument 1(rdi):
#   Desc: A pointer to the the queue that the element should be added to.
#   Type: tQueue* (8 bytes)

# Argument 2(rsi):
#   Desc: The address of a pointer that should hold the element's data pointer.
#   Type: void* (8 bytes)

# Argument 3(rdx):
#   Desc: The address of the pointer that should hold the fragment counter.
#   Type: unsigned long* (8 bytes)

# Argument 4(rcx):
#   Desc: The address of the pointer that should hold the size quantity.
#   Type: unsigned long* (8 bytes)

# Parameters(rax):
#   Type: signed char (1 byte)
#   Values:
#   0x1 on a successful dequeueing
#   0x0 if queue is empty

.globl dequeueCas128
dequeueCas128:
    movq %rdx, %r15
    movq %rcx, %r14
    dequeueCas128break:
        movq 0(%rdi), %rax # address of tq-head
        movq %rax, %r8 # address of tq-head
        lddq (%rax), %xmm0 # Q->head loaded into xmm0

        movq 8(%rdi), %rax # address of tq-tail
        movq %rax, %r10 # address of tq-tail
        lddq (%rax), %xmm2 # Q->tail loaded into xmm2

        movq %xmm0, %rax
        movq 16(%rax), %rax
        lddq (%rax), %xmm3 # head->next loaded into xmm3 hi=counter lo=ptr

        movupd %xmm0, %xmm4
        cmppd $0, (%r8), %xmm4
        movhlps %xmm4, %xmm5
        movq %xmm4, %rcx

jrcxz dequeueCas128break
movq %xmm5, %rcx
jrcxz dequeueCas128break

movq %xmm0, %rax
movq %xmm2, %rbx
cmp %rax, %rbx
jne forward

movq %xmm3, %rcx
jrcxz end

movq %xmm2, %rax
movhlps %xmm2, %xmm5
movq %xmm5, %rdx

movq %xmm3, %rbx
movq %rdx, %rcx
add $1, %rcx

lock cmpxchg16b (%r10)
jmp dequeueCas128break

end:
  movq $0x0, %rax
  ret

forward:  #else
  movq %xmm3, %rax
  lddqu (%rax), %xmm9  #value into rax
  movq %xmm9, (%rsi)  #write out

  movhlps %xmm9, %xmm10
  movq %xmm10, (%r15)  #write out

  movq 24(%rax), %rax
  movq %rax, (%r14)  #write out

  movq %xmm0, %rax
  movhlps %xmm0, %xmm5
  movq %xmm5, %rdx

  movq %xmm3, %rbx
  movq %rdx, %rcx
  add $0x1, %rcx

  lock cmpxchg16b (%r8)
jne dequeueCas128break

  movq %xmm0, %rdi
  movq 16(%rdi), %rdi
  call free
  movq $0, (%rdi)
movq %xmm0, %rdi
        call free
movq $0, (%rdi)

movq $0x1, %rax
ret

***excerpt***

Listing 5 dequeueCas128 and enqueueCas128 handles the atomic memory protection for the work queues. By utilizing cmpxchg16b 64-bit programs can use non-blocking functionality.