MASTER

Evaluation and design of multi-processor architectures

Wan, W.K.

Award date:
2008

Link to publication
Abstract

This master’s thesis consists of an evaluation of two very promising parallel processor architectures, namely the Cell Broadband Engine and the NVIDIA GeForce 8 Series GPU and a proposal for the design of a new architecture.

The evaluation is done with respect to the gaming application domain, in particular the 3D graphics pipeline. The graphics pipeline is chosen as it consists of all the steps that are necessary to project 3D objects onto a 2D screen and is therefore an important part in games.

For the evaluation, the most time consuming parts of the graphics pipeline are mapped onto the two architectures, followed by performing optimizations and measurements and documenting the most important lessons learned.

The results from this evaluation are ultimately used to propose a new architecture, which is able to process the steps in the graphics pipeline with high performance and can therefore be used as the first step towards our goal of obtaining “the ultimate gaming architecture”.
Acknowledgements

I want to thank my supervisor Henk Corporaal and my tutor Bart Mesman for introducing me to this project and for their time, support and feedback for the entire duration of my project.

Furthermore, I want to thank fellow student Paul Meys, who worked on the same project as I did and who provided valuable feedback.

I also want to thank my other fellow graduation students Chong Sheau Pei (Esther), Alberto Falcon, Raymond Frijns, Jochem van der Meer and Patricia Fuentes Montoya for providing the enjoyable times in the student room on the ninth floor in the Electrical Engineering building.

Other people I want to mention are my friends Wim Cools, Frank Ophelders, Tim Paffen, Marcel Steine, Mohammed El-Kebir, Martin Harwig, Diederik van Houten, Nick Matthijssen, Fred van Nijnatten and Alex Onete for joining several of the weekly pool meetings and doing other fun things for the entire duration of the project.

I want to thank the people in entire Electronic Systems Group as well, for the enjoyable times, like the daily coffee breaks, during my project.

Finally, I want to thank my parents and sisters for their support.
# Table of Contents

**List of Tables** ix  
**List of Figures** xi  
**List of Acronyms and Abbreviations** xiii

1 Introduction 1  

2 Graphics Pipeline 3  
2.1 Description of the graphics pipeline 3  
2.2 Software Implementation 7  
2.2.1 The Irrlicht Engine 8

3 Multi-Processor Architectures 11  
3.1 Cell Broadband Engine Architecture 11  
3.1.1 PowerPC Processor Element 11  
3.1.2 Synergistic Processor Element 13  
3.1.3 Element Interconnect Bus 15  
3.1.4 Memory 16  
3.1.5 Input/Output 17  
3.1.6 Programming 17  
3.1.7 PlayStation 3 18  
3.2 NVIDIA GeForce 8 Series Graphics Processing Unit 19  
3.2.1 Stream Processor 20  
3.2.2 Memory 20  
3.2.3 Programming 21  
3.2.4 GeForce 8800 GT 24  
3.3 Other architectures 25

4 Getting familiar with the architectures 29  
4.1 Application 29  
4.2 Mapping on Cell 31
List of Tables

3.1 Latency of SPU instructions .................................................. 15
3.2 Summary of architectures ...................................................... 27
4.1 Results of the YUV to RGB conversion program ...................' 33
4.2 Results of the kernels of the YUV to RGB conversion program .... 34
5.1 Mapping on the Cell ........................................................... 42
5.2 Mapping on the GPU ......................................................... 51
6.1 Number of tiles compared to size of memories ...................... 69
E.1 Timings of program increasing all elements in an array .......... 86
E.2 Timings of program increasing one variable ......................... 86
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The graphics pipeline</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Vertex processing</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>Rasterization</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>Anti-Aliasing</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>Texture mapping</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>The demo application</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Block diagram of the Cell Broadband Engine processor</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Vector types on the SPU</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>SPU Functional Units</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Element Interconnect Bus</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Block diagram of GeForce 8800</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>Scatter operations</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>Gather operations</td>
<td>21</td>
</tr>
<tr>
<td>3.8</td>
<td>Compilation trajectory of CUDA</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Drawing one line of a triangle</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>First function</td>
<td>38</td>
</tr>
<tr>
<td>5.3</td>
<td>Fixpoint number</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Sending pointers to SPEs</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>Mapping of first function on Cell</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>Drawing two triangles in parallel</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>Illustration of locking lines</td>
<td>44</td>
</tr>
<tr>
<td>5.8</td>
<td>Transfers between system memory and GPU memory</td>
<td>46</td>
</tr>
</tbody>
</table>
# List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>Two Dimensional</td>
</tr>
<tr>
<td>3D</td>
<td>Three Dimensional</td>
</tr>
<tr>
<td>AA</td>
<td>Anti-Aliasing</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmer's Interface</td>
</tr>
<tr>
<td>BIF</td>
<td>Cell Broadband Engine Interface</td>
</tr>
<tr>
<td>CA</td>
<td>Communication Assist</td>
</tr>
<tr>
<td>CAL</td>
<td>Compute Abstraction Layer</td>
</tr>
<tr>
<td>CBEA</td>
<td>Cell Broadband Engine Architecture</td>
</tr>
<tr>
<td>Cell BE</td>
<td>Cell Broadband Engine</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>ECC</td>
<td>Error-Correcting Code</td>
</tr>
<tr>
<td>EIB</td>
<td>Element Interconnect Bus</td>
</tr>
<tr>
<td>FAQ</td>
<td>Frequently Asked Questions</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FPOA</td>
<td>Field Programmable Object Array</td>
</tr>
<tr>
<td>GB</td>
<td>Gigabyte</td>
</tr>
<tr>
<td>Gbit</td>
<td>Gigabit</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GDDR3</td>
<td>Graphics Double Data Rate, version 3</td>
</tr>
<tr>
<td>GDDR4</td>
<td>Graphics Double Data Rate, version 4</td>
</tr>
<tr>
<td>GDDR5</td>
<td>Graphics Double Data Rate, version 5</td>
</tr>
<tr>
<td>GHz</td>
<td>Gigahertz</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HD</td>
<td>High Definition</td>
</tr>
<tr>
<td>Hz</td>
<td>Hertz</td>
</tr>
<tr>
<td>I/O</td>
<td>Input and Output</td>
</tr>
<tr>
<td>ID</td>
<td>Identifier</td>
</tr>
<tr>
<td>IOIF</td>
<td>I/O Interface</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>KB</td>
<td>Kilobyte</td>
</tr>
<tr>
<td>L1</td>
<td>Level-1</td>
</tr>
<tr>
<td>L2</td>
<td>Level-2</td>
</tr>
<tr>
<td>LS</td>
<td>Local Store</td>
</tr>
<tr>
<td>MB</td>
<td>Megabyte</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Mbit</td>
<td>Megabit</td>
</tr>
<tr>
<td>MFC</td>
<td>Memory Flow Controller</td>
</tr>
<tr>
<td>MHz</td>
<td>Megahertz</td>
</tr>
<tr>
<td>MIC</td>
<td>Memory Interface Controller</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction, Multiple Data</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Experts Group</td>
</tr>
<tr>
<td>ms</td>
<td>Millisecond</td>
</tr>
<tr>
<td>mutex</td>
<td>Mutual Exclusion</td>
</tr>
<tr>
<td>NaN</td>
<td>Not a Number</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>pixel</td>
<td>Picture element</td>
</tr>
<tr>
<td>PPE</td>
<td>PowerPC Processor Element</td>
</tr>
<tr>
<td>PPM</td>
<td>Portable Pixel Map</td>
</tr>
<tr>
<td>PS3</td>
<td>PLAYSTATION 3</td>
</tr>
<tr>
<td>PTX</td>
<td>Parallel Thread Execution</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality-of-Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RGB</td>
<td>Red Green Blue</td>
</tr>
<tr>
<td>ROP</td>
<td>Render Output unit</td>
</tr>
<tr>
<td>s</td>
<td>Second</td>
</tr>
</tbody>
</table>
SDK  Software Development Kit
SFU  Special Function Unit
SIMD  Single Instruction, Multiple Data
SP  Stream Processor
SPE  Synergistic Processor Element
SPU  Synergistic Processor Unit
texel  Texture element
TnL  Transform and Lighting
VLIW  Very Long Instruction Word
VMX  Vector Multimedia Extensions
WMV  Windows Media Video
XDR  Extreme Data Rate
XLC  XL C/C++ Compiler
YUV  Luminance-Chrominance
µs  Microsecond
Chapter 1

Introduction

This report serves as the final report for the graduation project started in January 2008, which consists of an evaluation of the performance of certain currently available highly parallel processor architectures with respect to the gaming application domain. Using the results, conclusions and lessons learned from the evaluation, a new architecture is proposed.

The 3D graphics pipeline is the main aspect for the evaluation of the gaming application domain and is explained in chapter 2. It is expected that this aspect requires quite an amount of resources such that the user can obtain a realistic view of the in-game world. This aspect can both be decomposed into functional parallelism, by running each part of the pipeline independently, or into data parallelism, by splitting the input over multiple processors. A hybrid approach is also possible, yielding several possibilities for parallelisation.

Other aspects like physics processing and artificial intelligence in games may also require a lot of computational power, but these aspects are not evaluated in this graduation project and will therefore not be covered in this report.

As Paul Meys is a fellow student who is also working on the same project, this report will contain certain parts of the work he performed, either to explain things, to compare results or because those parts are important for this work. These parts will be denoted as such explicitly, unless it is unambiguously clear that these parts were Paul’s work.

In chapter 2 the graphics pipeline is described in more detail and a software implementation is
mentioned. After that comes chapter 3, which covers several currently available multi-processor architectures. Two of these architectures are covered in more detail as they are main architectures in this project. Following that is chapter 4, which is about a small assignment in order to get familiar with the architectures used. The mapping described in chapter 5 includes plenty of details and is the basis for the proposal for a new architecture made in chapter 6. At the end of this report are conclusions, followed by several appendices and the bibliography.
Chapter 2

Graphics Pipeline

This chapter describes the stages of the 3D graphics pipeline in section 2.1 and also describes a software implementation of the pipeline and the rationale for using this implementation. The mapping of this implementation onto the architectures is evaluated and used as a starting point for the proposal of the new architecture.

2.1 Description of the graphics pipeline

The 3D graphics pipeline, shown in Figure 2.1 [FHWZ04], is a multi-stage pipeline which takes a model of a 3D world as input and outputs pixels corresponding to that model [FHWZ04]. Each stage of the pipeline is optionally split into multiple parts, which are described below. For a more detailed explanation about the operations in the graphics pipeline, the reader is referred to books about computer graphics, for example [HB03].

In the first stage of the pipeline, which is the vertex processing stage and which is done by either the Transform and Lighting (TnL) unit or by the vertex shader unit on the Graphics Processing Unit (GPU), the positions of vertices in the 3D world are transformed and the attributes of the vertices are calculated as can be seen in Figure 2.2 [FHWZ04].

In the first part the vertices that make up a model are transformed to the correct positions in the 3D world. After that the positions are adjusted to the eye or camera, the point from which
CHAPTER 2. GRAPHICS PIPELINE

Figure 2.1: The graphics pipeline

Figure 2.2: Vertex processing


2.1. DESCRIPTION OF THE GRAPHICS PIPELINE

the 3D world is perceived. Then the positions are projected onto the screen, which can include perspective correction\(^1\). In every step it is possible that vertices are removed from the pipeline, if they are not necessary for getting the same result in the final image\(^2\).

During the lighting part, the colour of the vertices is calculated, as are light and material properties\(^3\). It is possible to replace this entire stage by a vertex shader program if desired.

During the lighting part, the colour of the vertices is calculated, as are light and material properties\(^3\). It is possible to replace this entire stage by a vertex shader program if desired.

In the rasterization stage, shown in Figure 2.3 [FHWZ04], each triangle or other primitive\(^4\) is converted to an image consisting of pixel fragments or just fragments\(^5\). The colours of the fragments are then calculated, for example by texture mapping or by running through a custom written pixel shader (also known as fragment shader) program. After that each fragment will go through the Render Output unit (ROP), also known as Raster Operations unit\(^6\), in which

\[^1\]Perspective projection makes objects which are farther away seem smaller compared to objects with the same height which are closer to the camera.

\[^2\]For example, vertices of a triangle that lies entirely behind the camera.

\[^3\]For example the position and direction of light and whether the material is specular, diffuse or ambient.

\[^4\]For example points or lines.

\[^5\]The fragments will be considered pixels if they are written into the framebuffer. [NVI06]

\[^6\]The observative reader may notice that the ROP was not mentioned in the architecture of the NVIDIA GeForce 8 Series GPU. This is because the ROP is not programmable in Compute Unified Device Architecture (CUDA), but even when using a 3D graphics Application Programmer’s Interface (API), it is only possible to vary parameters for
several tests are performed for deciding whether the fragment will become a pixel. Several of these tests are described below, although current GPUs may perform more or less tests and in a different order for various reasons, as to obtain higher performance or better quality. As an example, the NVIDIA GeForce 8 Series GPU uses the ROP for Anti-Aliasing (AA) \[\text{NVI06}\]. AA is used to remove subpixel artifacts, see Figure 2.4 for an example.

![Figure 2.4: Anti-Aliasing](image)

The scissor test checks if the fragment is within a specified rectangle \[\text{SA06}\]. If the test is disabled or if the fragment is inside the rectangle, it will continue to the next test, otherwise it will be discarded.

In the alpha test, the alpha value of a fragment is compared to one specified constant value and the fragment is discarded if the comparison fails and the test is enabled.

The stencil test compares the value at the current fragment position in the stencil buffer against a specified value and discards the fragment if the comparison fails and the test is enabled.

The Z-test or depth buffer test compares the depth of the current fragment position with the depth in the depth buffer and discards the fragment if the comparison fails and the test is enabled.

During alpha blending, the fragments are blended with the current colour at the current fragment position in the framebuffer. This blending depends on the alpha value of the fragment and/or the current colour in the framebuffer.

---

7These tests can be disabled, but enabling them may result in faster rendering as fragments may possibly be discarded.

8An example where this test is usable is when the 3D world is projected onto a smaller rectangle than the screen and the border contains a 2D image.

9For example to remove complete transparent pixels.

10This test can be used for generating shadows for example.
2.2. SOFTWARE IMPLEMENTATION

In current DirectX 10 compliant GPUs, there is also an option to run a custom geometry shader program before rasterization, which can for example break a large triangle into multiple smaller triangles [NVI06]. Furthermore, in DirectX 10 compliant GPUs vertex, pixel and geometry shader programs run on the same shader units, the so-called unified shaders, whereas on older GPUs the shader units for executing vertex and pixel shader programs could be and were different.

2.2 Software Implementation

To be able to implement parts of the graphics pipeline onto either the Cell or the GPU, an existing software implementation of the pipeline, known as a software renderer, can be used as a starting point, as opposed to writing an entire renderer from scratch. Seeing as both Direct3D [D3D] and OpenGL [OGL] are used as 3D graphics, an idea would be to look for software renderers in either API, or looking for software implementations of either API.

Direct3D is a proprietary 3D API by Microsoft and is only available for Windows, whereas the programming environment for the Cell supports only Linux. Using Direct3D is therefore deemed impractical in this project, so no further effort is put into finding a software implementation of Direct3D.

OpenGL on the other hand is a cross-platform 3D API and is available for Windows, Mac OS X and various UNIX flavours. As it is possible to use the OpenGL API on free UNIX variants like Linux and FreeBSD, there must be a free open source implementation of the OpenGL specification available. Such a free implementation is the Mesa 3D Graphics Library (Mesa) [MES] and it provides hooks for graphics drivers in order to allow hardware acceleration.

To be able to use this software implementation of the OpenGL specification in this graduation project, the rendering part must be isolated from the rest. The Mesa source code is a mess however. It has to provide the mentioned hooks for optional hardware acceleration by using a lot of if-statements and function pointers. It also uses a lot of C-macros, which can result in different functionality for the same C-macro depending on previously defined C-macros and also

\[^11\text{This is the opinion of the author and does not necessarily express the view of others. The FAQ on the Mesa website does mention that parts are not well documented due to lack of spare time.}\]
on the order in which C-header files are included. Furthermore, most of the functions it uses are actually function pointers stored in an array. These function pointers are assigned at certain points during execution and the perceived functionality of such a function call can therefore differ depending on whether the function pointer changed or not. Isolating the graphics pipeline is not an easy task, due to the small amount of documentation\footnote{The documentation must be separately built and is not included in normal distributions of Mesa.} and because the source code is not modular\footnote{The Mesa developers are currently building a new architecture for 3D graphics drivers, which is more modular and which goal is to allow for smaller and simpler drivers. Perhaps interesting to note is that the Mesa developers are in the process of creating a Cell driver for this new architecture\footnote{Direct3D is only supported on Windows.}.} \footnote{\cite{mesa}}. The idea of using Mesa for the graphics pipeline is abandoned for the above reasons.

The search for a software renderer finally leads to the Irrlicht Engine (Irrlicht), which is “an open source high performance realtime 3D engine” \cite{irrlicht}.

\subsection{The Irrlicht Engine}

Irrlicht is a free open source 3D engine written in C++ and supports Windows, Linux and Mac OS X. It can use both Direct3D and OpenGL as backend\footnote{\cite{direct3d}} and comes with two software renderers. One of these software renderers is quite fast, but only works correctly for 2D graphics and the other one supports high quality 3D graphics. Due to its lack of correct support for 3D graphics, the first mentioned software renderer will not be considered in the rest of this report; when mentioning the Irrlicht software renderer, the reader should read it as if the second software renderer is explicitly mentioned.

The Irrlicht engine is written in a modular way: the renderers are contained in their own set of source files. Functionality that is not part of the graphics pipeline is in different C++ source files. The API is also quite well documented and the function names, together with their class names, are quite clear\footnote{For example, there are functions named “drawTriangle”\footnote{See Figure 2.5 \cite{fhwz04} for an example.}} at least for the part that is of interest: the software renderer.

The 3D graphics capabilities of the software renderer matches some of the capabilities of current GPUs. For example, it does perspective correct texture mapping\footnote{See Figure 2.5 \cite{fhwz04} for an example.} and can do bilinear texture filtering, which means that texture lookups in a 2D texture return the weighted average of the
2.2. SOFTWARE IMPLEMENTATION

Figure 2.5: Texture mapping

four Texture elements (texels) nearest to the pixel fragment that is currently computed. It also covers both the geometry and the rasterization stage of the graphics pipeline, although there is no support for custom shader programs and some of the optional tests for discarding fragment pixels are left out.

Irrlicht comes with several sample programs and a demo application, which demonstrates the capabilities of the engine. An overview of this demo application can be found in Figure 2.6. Source code for the samples and the demo is also included. The demo can use Direct3D, OpenGL and both software renderers for showing the capabilities of the engine and it appears that output on the screen by the software renderer matches the output by Direct3D and OpenGL. This demo is profiled to see which functions in the software renderer take the most time, as accelerating parts\(^\text{17}\) that take only very little time has no noticeable impact on the whole application.

Irrlicht has been compiled under Windows XP using the C++ compiler that is part of Microsoft Visual Studio 2005. It has also been compiled under Linux using GNU Compiler Collection (GCC), although there were some fixes needed in the source code. More information about the compilation under Linux can be found in Appendix C.

\(^{17}\)By mapping them onto the Cell or the GPU.
CHAPTER 2. GRAPHICS PIPELINE

Figure 2.6: The demo application (Arrows indicate function calls to manage the objects)
Chapter 3

Multi-Processor Architectures

In this chapter several currently available multi-processor architectures are mentioned. The two architectures used in the evaluation are covered in more detail, namely the Cell Broadband Engine Architecture by IBM, Sony and Toshiba in section 3.1 and NVIDIA’s GeForce 8 Series Graphics Processing Unit in section 3.2. A short overview of other multi-processor architectures that were considered, but not used for the evaluation can be found in section 3.3.

3.1 Cell Broadband Engine Architecture

The Cell Broadband Engine (Cell BE), also known as the Cell Broadband Engine Architecture (CBEA) or just Cell, consists of several components as can be seen in Figure 3.1 [CRDI07]. It contains one PowerPC Processor Element (PPE), eight Synergistic Processor Elements (SPEs) and controllers for interfacing with memory and Input and Output (I/O), all of which is connected by an Element Interconnect Bus (EIB).

3.1.1 PowerPC Processor Element

The PPE is a dual-threaded, dual-issue, in-order PowerPC processor and has support for vector instructions by using the integrated Vector Multimedia Extensions (VMX), which are also known¹

¹The numbers in this figure assume a 3.2 GHz clock speed.
CHAPTER 3. Multi-processor Architectures

Figure 3.1: Block diagram of the Cell Broadband Engine processor

as AltiVec. It is responsible for the overall control, running the Operating System (OS) and can be used to allocate tasks to the SPEs [CRDI07].

As mentioned, the PPE is an in-order machine, but it has some of the benefits of out-of-order execution incorporated into it. The processor can, for example, continue on a load cache miss until it encounters an instruction that actually depends on that load. Up to eight requests to the L2 cache can be sent using this mechanism without stalling. The PPE can also move instructions that would stall during the issue stage to special “delayed execution pipelines” that executes the instructions at a later, but specified point in time.
3.1.2 Synergistic Processor Element

The SPE consists of a Synergistic Processor Unit (SPU), 256 KB of Local Store (LS), also known as local storage, and a Memory Flow Controller (MFC) and communicates with the PPE, with other SPEs, with main memory and with other devices through so called channels [CRDI07][IBM07b].

The SPU is a dual-issue, in-order big-endian machine having a 128 entry unified register file; each entry is 128 bits wide. It can only operate on data in its register file and in its LS and has to rely on the MFC to asynchronously transfer data from and to main memory and from and to other SPEs’ local stores. The instructions it executes must also be stored in the LS, meaning that the data that can be stored in the LS is actually less than 256 KB.

![Figure 3.2: Vector types on the SPU](image)

Being based on an SIMD architecture the SPU can perform operations on 128 bit vectors. These vectors can be sixteen 8-bit, eight 16-bit, four 32-bit, two 64-bit integers or one 128-bit integer as can be seen in Figure 3.2. Besides vectors of integers, it is also possible to put four single precision or two double precision floating point numbers in a vector. The SPU can operate on scalar values by storing scalars in the so-called preferred slot of a vector register and performing operations on that register. The grey parts in Figure 3.2 denote the preferred slots.

Shown in Figure 3.3[IBM07b] are the functional units of the SPU. The SPU is able to issue two instructions per cycle, one on the odd and one on the even pipeline. Recall that the SPU is an
in-order machine, so this is only possible if one of the two issued instructions is an odd and the other is an even pipeline instruction type. Of course it must also be the case that there are no dependencies between these instructions and both instructions must have been prefetched from the LS.

Table 3.1 provides an overview of the latency for instructions on both the even and the odd pipeline [IBM07b]. A branch miss has a penalty of 18 to 19 cycles and there is a 6 cycle stall after an operation on double precision floating point numbers.

The channel interface provides two outbound mailboxes per SPE for sending up to four 32-bit messages per mailbox to the PPE, one inbound mailbox per SPE is available to retrieve data from the PPE and other SPEs. Furthermore, the channel interface supports two signal-notification channels per SPE. These can be used to send one 32-bit message per signal-notification channel to an SPE.

Signal-notification channels can be configured to be in OR-mode or in overwrite mode. In the
3.1. CELL BROADBAND ENGINE ARCHITECTURE

Table 3.1: Latency of SPU instructions

<table>
<thead>
<tr>
<th>Even pipeline instructions</th>
<th>Latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision floating point operations</td>
<td>6</td>
</tr>
<tr>
<td>Integer multiplies, convert between floating point and integer, interpolate</td>
<td>7</td>
</tr>
<tr>
<td>Immediate loads, logical operations, integer add and subtract, signed extend, count leading zeros, select bits, carry and borrow generate</td>
<td>2</td>
</tr>
<tr>
<td>Double precision floating point operations</td>
<td>7</td>
</tr>
<tr>
<td>Element rotates and shifts</td>
<td>4</td>
</tr>
<tr>
<td>Byte operations (count ones, absolute difference, average, sum)</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Odd pipeline instructions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Shuffle bytes, quadword rotates, and shifts</td>
<td>4</td>
</tr>
<tr>
<td>Gather, mask, generate insertion control</td>
<td>4</td>
</tr>
<tr>
<td>Estimate</td>
<td>4</td>
</tr>
<tr>
<td>Loads</td>
<td>6</td>
</tr>
<tr>
<td>Branches</td>
<td>4</td>
</tr>
<tr>
<td>Channel operations, move to/from special purpose registers</td>
<td>6</td>
</tr>
</tbody>
</table>

first case, all messages send to a signal-notification channel will be combined using a logical-OR until the SPU issues a read message from said channel. After reading the message, the channel is reset to zero. In the latter case, messages send to a signal-notification channel will possibly overwrite a previous message. Previous messages will be lost if the SPU had no chance to read them.

3.1.3 Element Interconnect Bus

![Element Interconnect Bus Diagram]

Figure 3.4: Element Interconnect Bus
CHAPTER 3. MULTI-PROCESSOR ARCHITECTURES

The EIB consists of one address bus and four data rings that connect the PPE, the SPEs, main memory and other I/O devices and operates at half the processor frequency [CRDI07] [IBM07b]. Two of those data rings transfer data in clockwise and the other two rings transfer data in counter clockwise direction. Each ring, which is 16 byte wide, allows up to three concurrent transfers, as long as the paths of all the transfers are disjoint. Furthermore, all elements connected to the EIB can both receive and send data at the same time and the order in which elements are connected to the EIB is fixed, as shown in Figure 3.4 [CRDI07].

Each time an element wants to send data to another element, it makes a request to the arbiter. The arbiter decides which ring is granted to which element and when this ring is granted to that element. Obviously, a ring will not be granted to an element if the transfer would overlap with another transfer on that ring. The arbiter will not grant a ring either if the transfer would need to cross half the ring. Given that there are twelve elements connected to the EIB, as can be seen in Figure 3.4 the maximum number of hops that a transfer will take is six.

The EIB guarantees that progress will be made, but does not support other forms of Quality-of-Service (QoS). It does implement resource allocation to allow controlled access to I/O and memory.

3.1.4 Memory

The Memory Interface Controller (MIC) acts as an interface between the EIB and main memory [CRDI07] [IBM07b]. Two controller channels are supported, each one connecting to ECC-protected Rambus XDR DRAM and both channels operate at octal data rate. Each channel has separate read and write queues allowing up to 64 reads and writes to be queued.

Memory accesses have sizes of multiples of 16 bytes up to 128 bytes. It is possible to read data that is less than 16 bytes in size. Writes of sizes less than 16 bytes are possible by using a read-modify-write operation.

The MIC supports several modes that can be used to enhance performance and which can be controlled in software. These modes include a fast-path mode, in which latency can be reduced by servicing a request immediately when the queues are empty, high-priority read mode, in which SPE reads are prioritized, early read mode, in which a read is started before a previous
write is completed, speculative read mode, in which reads can be dispatched to the memory before a response is received from the EIB, and a slow mode, which can be used for power management.

3.1.5 Input/Output

Two Rambus FlexIO interfaces are supported by the Cell [IBM07b]. The interface named IOIF1 only supports the so called noncoherent I/O Interface (IOIF) protocol, which is used to connect I/O devices to the Cell. The other interface is known as IOIF0 or as BIF/IOIF0 and software can select which protocol to use: the IOIF protocol or the memory-coherent Cell Broadband Engine Interface (BIF) protocol. An example for the usage of the latter protocol is for connecting to another Cell processor.

3.1.6 Programming

The PPE is compatible with version 2.0.2 of the PowerPC Instruction Set Architecture (ISA) [IBM07b], which is described in [Fre07]. As mentioned, it also supports the VMX instructions for vector operations. Programming the PPE can therefore be done using assembly code or any (already available) compiler that targets the PowerPC ISA.

The ISA of the SPU is described in [IBM07d], making it possible to write an SPU program in assembly. This ISA includes operations for performing requests to the MFC, for example to start a DMA transfer.

There is an additional document which describes extensions to the C and C++ language for the Cell Broadband Engine [IBM07a]. This document describes how to specify vector types in C/C++ and contains intrinsics that should be used as an alternative to assembly instructions in the ISA that do not have a direct equivalent in C/C++.

IBM provides a Software Development Kit (SDK) on its website for developing Cell applications [CEL]. This SDK runs under the Linux Operating System and includes cross-compilers targeting both the PPE and the SPU ISA, a system simulator, samples including source code and libraries for the Cell. Throughout this project, version 2.1 of the SDK is used.
Two cross-compilers are provided in the SDK for the Linux OS: the C/C++ compilers that are part of the GCC [GCC] and IBM’s XL C/C++ Compiler (XLC) [XLC]. Both compilers come with separate versions for the PPE and the SPU ISA and it is up to the programmer to decide which compiler to use, although the makefiles default to GCC. Source code is not 100% exchangeable between the two compilers. As an example, when writing C++ code GCC overloads common operators like addition (+) and subtraction (−) for use with vector types, whereas XLC requires the use of the intrinsics defined in [IBM07a].

The simulator simulates an entire Linux system containing a Cell processor [IBM07c] and allows transferring files, for example Cell executables, from and to this Linux system. It has three modes: simple, fast and cycle. In simple mode the effect of the instructions is simulated, but the time required to execute these instructions is not accurate. Fast mode is similar to simple mode, but makes it impossible to collect statistics about for example the number of instructions. Cycle mode is also like simple mode, except it does attempt to simulate accurate timings.

The libraries in the SDK can be used in the PPE to start and wait for SPEs to upload SPU programs and to communicate between the PPE and a SPE using standard C/C++ function calls among other things. They also implement the part of the C and C++ standard libraries that is described in [IBM07a] for the SPU. An example of starting SPEs and waiting for them on the PPE can be seen in Appendix A.

3.1.7 PlayStation 3

The Sony PLAYSTATION 3 (PS3) or just PlayStation 3 is a video game console that contains a 3.2 GHz Cell processor and 256 MB main memory [PS3]. Of the eight SPEs that are in the Cell processor, one is disabled for yield reasons.

It is possible to install Linux on the PS3, after which programming the Cell processor is an option. Linux on the PS3 will run on top of a hypervisor layer, which reserves one SPE permanently, leaving only six SPEs available for use [See08].

Yellow Dog Linux 5 was installed on the PS3 in the Electrical Engineering Department by Kris Hoogendoorn for an assignment in the 5KK70 course. 10 GB of harddisk space was allocated.
for use with Linux and the above mentioned SDK was not installed. A separate Linux image in a virtual machine was provided that had the SDK installed. This image can be used with VMware Player, which runs under Windows and Linux and is available free of charge from VMware’s website [VMW].

3.2 NVIDIA GeForce 8 Series Graphics Processing Unit

The block diagram of the programmable part of the GeForce 8 Series GPU can be seen in Figure 3.5 [NVI06]. GeForce 8 Series GPUs require a host CPU from which they receive the commands and the data they have to process. The input assembler retrieves data from buffers, converts them and generates IDs which are used for repeated operations. The three thread issuers issue vertex, geometry and pixel shader operations to the Stream Processors (SPs). These SPs are
grouped and have access to shared L1 caches and texture units. At the bottom are texture and L2 caches, which are connected to the video memory. Part of the onboard video memory forms the framebuffer, which contains pixels that are to be displayed on the screen.

### 3.2.1 Stream Processor

The SPs support both single precision floating point and integer operations [NVI06] and run at a different clock speed compared to the rest of the GPU. Depending on the GPU model, there may be more or fewer SPs available, giving higher or lower performance.

The SPs are grouped per eight into so called multiprocessors, with all eight SPs performing the same instruction in an SIMD fashion [NVI07]. The eight SPs per group have a banked register file with a total of 8192 registers\(^2\), so each SP has 1024 registers available. Each group has its own L1 cache\(^3\), which is also known as shared memory and has a size of 16 KB organized into 16 banks. Besides L1 caches, the diagram shows that there are also L2 caches, but the structure and the sizes are not public.

### 3.2.2 Memory

Onboard memory runs, just like the SPs at a different clock speed compared to the rest of the GPU. Of the total memory that is available, there is a part that is reserved for the framebuffer [NVI07] [NVI06]. There is a maximum of 64 KB reserved for so-called constant memory, which has its own cache of 8 KB per group of SPs.

Textures refer to data in the GPU memory, but have special properties. It is for example possible to refer to a location outside the texture and unlike C or C++, where accessing arrays out of bounds is undefined behaviour, it is possible to clamp coordinates to a valid range or to wrap coordinates. Textures also have an 8 KB cache per group of SPs, with the property that the cache is optimized for 2D spatial locality. Details about this 2D spatial locality are not given.

---

\(^2\)The size of a register is not specified, but given the fact that the GPU works with single precision floating point numbers and can work with 8-, 16-, 32- and 64-bit numbers, it is presumed to be 32-bit wide.

\(^3\)Although it is called L1 cache, it can be used as scratchpad memory depending on the programming model.
3.2.3 Programming

To program the GPU, it is possible to write so called shader programs that run in the programmable parts of the graphics pipeline and this require the use of a 3D graphics API like OpenGL or Direct3D. This kind of programming of the GPU is not considered in this report\(^4\). Reasons for this are the need to map parts of algorithms onto this API, where computations on elements of an array have to be done by drawing a triangle overlapping all elements in a texture, and because there is no support for scatter operations\(^5\), although gather operations\(^6\) are supported.

The use of the CUDA technology [NVI07] is what will be considered. CUDA provides a C interface to an NVIDIA GeForce 8 or newer Series GPU with limited C++ support and some other limitations: the lack of support for the C standard library, no function pointers, no support for recursion and all data must be written to and retrieved from video memory by the host system. It provides both a high-level API and a low-level API; only the high-level API will be covered in the rest of this subsection.

\(^4\)The interested reader can find chapters for programming the GPU in this way in books like [Fer04] and [Pha05].

\(^5\)Being able to write arbitrary places in memory, see Figure 3.6 [NVI07]

\(^6\)Being able to read from arbitrary places in memory, see Figure 3.7 [NVI07]
To write CUDA programs, the CUDA toolkit needs to be installed from [CUDA] and this toolkit is available for Windows, Linux and Mac OS X. The same website also contains an SDK, which comes with several sample projects that can be used as a starting point or to get an understanding of how CUDA works. Throughout this project, version 1.1 of the toolkit is used.

CUDA extends C by providing ways to define whether a variable resides in normal system RAM or whether it resides in video memory, constant memory or shared memory. It also provides ways to specify whether a function resides on the host system or on the GPU. If a function resides on the GPU, it can be specified to be an entry-point in the GPU (a kernel), just like main is generally the entry-point function in C and C++.

To introduce parallelism on the GPU, CUDA uses the notion of blocks, warps and threads. A thread is the unit of execution and performs the instructions in an entry-point function from the start to the end. 32 threads are grouped together in what is called a warp and these threads will perform the same instructions, which are optionally prefixed with guard statements. Threads are also grouped into blocks and if the number of threads in a block is not a multiple of the warp size, it will be padded to a multiple of the warp size. The threads in one block will, during the lifetime of the entry-function, always be executed on the same group of SPs. It is possible to insert synchronization points at certain parts of the code; all threads in one block will wait until all the other threads in the same block have reached the synchronization point before continuing.

The programmer is the one who specifies the number of blocks and the number of threads that must be used when calling the entry function on the GPU from normal C or C++ code. All threads in one block will be time-sliced and executed concurrently. Whether more blocks are executing in parallel depends on the number of SPs available, more SPs mean that more blocks can run concurrently, the number of registers and size of shared memory used in a block, a low number means that there can be more active blocks, and the number of threads in a block, as there is a limit to the number of threads per group of SPs. Using hundreds of threads per group of SPs is advised to hide latency and overhead. A large number of blocks is also advised, such

---

7 This is not always the case. Windows programs for example can have WinMain as their entry point and it may be the case that the C/C++ Runtime Library has a different entry point which calls “main”, so it can load external libraries, initialize variables or run constructors in C++ code among other things.

8 A use of this is when some of the threads in a warp take a different branch than other threads.
that the GPU program will scale to future generations (NVIDIA) GPUs. An example of a CUDA program is available in Appendix B.

Common instructions take four clock cycles per warp; more complex instructions can take sixteen clock cycles or, if they are implemented using a combination of several instructions, even more. Branches takes as long as all the paths that must be executed per warp.

Accessing memory is rather costly, it can take several hundreds of clock cycles. Constant memory is as fast as reading registers, if the value to be read is cached and all threads in a warp read the same address. Shared memory can also be as fast as reading registers, as long as there are no bank conflicts. Using the right patterns for memory access can result in higher bandwidth and using the right memory can also result in faster execution times. This is thus an optimization strategy that can be employed if deemed necessary; this and other optimization strategies can be found in [NVI07].

![Compilation trajectory of CUDA](image)

The compilation trajectory of CUDA, shown in Figure 3.8, differs from normal compilation of C
or C++ code and is as follows. First CUDA source files will be preprocessed and normal C and C++ code is separated from code that is to run on the GPU. After that the separated C/C++ code is compiled with a normal C/C++ compiler, as are the normal C/C++ source files. The GPU code will be compiled into an assembly-like language called Parallel Thread Execution (PTX), which is optimized and assembled into a binary object file and embedded into the host executable resulting from the normal C/C++ code. At runtime there is actually an additional step whenever a GPU entry function is called the first time, as the GPU driver can recompile the object code and optionally perform extra optimizations for the GPU code. This allows forward compatibility without the need to compile the CUDA source files from scratch.

It is possible to compile CUDA files using so called device emulation mode, in which all functions will be executed on the host CPU rather than the GPU. This mode is useful for debugging purposes, although certain errors can not be found using this mode, as the GPU is not simulated. For example, in device emulation mode all threads will run sequentially and all memory resides in the host, so dereferencing pointers referring to simulated "GPU" memory will work in this mode.

### 3.2.4 GeForce 8800 GT

The NVIDIA GeForce 8800 GT chip assembled by the manufacture EVGA is the GPU that is available for use during this graduation project. The NVIDIA 8800 GT reference GPU has a core clock speed of 600 MHz, comes with 112 SPs running at 1500 MHz each, has 512 MB GDDR3 DRAM running at 900 MHz with a 256 bit memory bus [GTN]. The available EVGA GPU has slightly different clock speeds: 650 MHz for the core, 1620 MHz for the SPs and 950 MHz for the memory [GTE].

The GeForce 8800 GT is actually a newer revision of the GeForce 8 series. One of the things that are supported by newer revisions are atomic operations on 32 bit integers [NVI07]. These operations are guaranteed to complete without interruption from other threads as soon as they are started. Under this revision, it possible to asynchronously transfer data between the host system and GPU memory, while the GPU is running a program [GTE]. It also supports PCI Express

---

9 Using atomic operations on the GPU does not come with nuclear risks.

10 Modifying data that the GPU program is using may not yield the desired results however.
2.3. OTHER ARCHITECTURES

and there is support for hardware accelerated video decoding of H.264, VC-1, WMV and MPEG-2 [NVI06], but this part of the GPU is not accessible using CUDA [NVI06].

The host system containing the GPU has Windows XP Service Pack 2 installed on it, contains 2 GB of internal memory and comes with a quadcore Intel Core 2 Quad CPU model Q6600, which runs at 2.40 GHz.

3.3 Other architectures

The Cell processor and the GeForce 8 Series GPU are not the only currently available multi-processor architectures. This section contains a list of other multi-processor architectures that look promising for mapping the graphics pipeline on and these architectures may or may not be readily available at the time of this writing.

The first architecture in this list is AMD/ATI’s R600 GPU [AMD]. The R600 has up to 320 stream processors and is based on a VLIW architecture [Som07]. The 320 Stream Processors are split into groups of five Processing Elements (PEs), where four of the five PEs can only do simple operations and one PE can also do more complex operations. As for programming, AMD has an SDK available for free [AMD]. There are two languages that can be used to program the R600: Compute Abstraction Layer (CAL) and Brook+. CAL is an assembly-like intermediate language which allows for forward compatibility. Brook+ is a C-like language, but one of the major disadvantages is that it does not support scatter operations yet.

AGEIA’s PhysX [AGE] is an add-on card for PCs that accelerates physics operations in games that make use of the card. Unfortunately there is hardly any information available about the architecture [AGE], and although there is a free SDK available [AGE], this SDK does not allow direct access to the underlying PEs.

---

11 The motherboard however does not support PCI Express 2.0, so the PCI Express 1.1 speeds will be used for transfers between the host system and the GPU.
12 More precisely, it is not documented as there is no mention of it in the CUDA manual at all.
13 ATI used to be a separate company, but is nowadays part of AMD.
14 After registration.
15 Information on the internet is based on speculation and patent descriptions. There is no information from AGEIA about the architecture.
16 After registration.
CHAPTER 3. MULTI-PROCESSOR ARCHITECTURES

MathStar’s Field Programmable Object Array (FPOA) [MAT] is an architecture that contains hundreds of objects and is a programmable device like an FPGA. There are three types of core objects, which run at up to 1 GHz: an Arithmetic Logic Unit (ALU), a multiply accumulator or a register file. Besides these core objects, there are also objects for accessing I/O. MathStar has evaluation boards and tools available, but these are not free; the documentation for programming in SystemC is part of the tools.

The picoChip’s PC102 picoArray is an MIMD architecture consisting of 308 PEs [pic08]. Each PE can be one of the following two types: a 16-bit Harvard architecture processor and hardware co-processors. The hardware co-processors are the PEs which used to accelerate functions. picoChip offers tools and development platforms for an unknown fee. Documentation for programming the PC102 processor in a C-like language is available for free.

Rapport’s KC256 chip consists of 256 PEs running at 100 MHz and is based on a MIMD design [RAP]. These 256 8-bit PEs are grouped per stripes of 16 PEs; each stripe can be (re)configured in one clock cycle. It is possible to cluster multiple KC256 chips for more performance if desired. Rapport offers a development platform containing a KC256 chip and offers tools; both are available for a fee. The documentation for programming the chip in a C-like language is part of the tools.

ClearSpeed’s CSX600 is an SIMD architecture which consists of 96 PEs [CLE]. These PEs can do both single and double precision floating point additions and multiplications and contain an integer ALU and multiply accumulator. ClearSpeed offers boards containing the CSX600 as add-ons for normal PCs for a fee. It also offers an SDK for programming the CSX600 for a fee. The documentation for programming the CSX600 in a C-like language is available for free.

A summary of these architectures are given in Table 3.2. This list of architectures is obviously incomplete. Besides commercial offerings, there are also research projects that work on multi-processor architectures. Among these are the MIT’s raw Architecture Workstation [RAW], Stanford Smart Memories project [SMA], Berkeley’s RAMP [RAM] and Austin’s TRIPS project [TRI]. The architectures in these research projects are not covered here, as the level of detail is quite high and the amount of information available is quite extensive.
### Table 3.2: Summary of architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Number of PEs</th>
<th>Parallelisation</th>
<th>Programming</th>
<th>Floating point support</th>
</tr>
</thead>
<tbody>
<tr>
<td>R600</td>
<td>320</td>
<td>VLIW</td>
<td>Assembly level C-like</td>
<td>Single precision</td>
</tr>
<tr>
<td>PhysX</td>
<td>Unknown</td>
<td>Unknown</td>
<td>High-level API</td>
<td>Most likely</td>
</tr>
<tr>
<td>FPOA</td>
<td>Hundreds</td>
<td>Combination of different types of PEs</td>
<td>SystemC</td>
<td>No</td>
</tr>
<tr>
<td>PC102</td>
<td>308</td>
<td>MIMD</td>
<td>C-like</td>
<td>No</td>
</tr>
<tr>
<td>KC256</td>
<td>256</td>
<td>MIMD</td>
<td>C-like</td>
<td>No</td>
</tr>
<tr>
<td>CSX600</td>
<td>96</td>
<td>SIMD</td>
<td>C-like</td>
<td>Single precision Double precision</td>
</tr>
</tbody>
</table>
Chapter 4

Getting familiar with the architectures

This chapter describes a small assignment that was used as a preparation for the mapping of parts of the graphics pipeline. In particular, the assignment was to get familiar with both the Cell and the GPU architectures, with mapping and optimizing programs and to get an understanding of the performance.

4.1 Application

The application chosen for this part was a program that converted images from one colour space to another, namely from YUV to RGB. This application is also used in an assignment in the 5KK70 (formerly 5KK10) course [PBD].

The program takes three files as input; one file for the Y, U and V data each. More specifically, the input is in the YCrCb420 format. The output consists of one file containing the RGB colour data in Portable Pixel Map (PPM) format.

Using the input, the U and V data are scaled separately in vertical direction to obtain YCrCb422 format. After this the U and V data are, again separately, scaled horizontally to obtain YCrCb444 format. Finally, the RGB colour values are computed, after which the data is written into the output file. In pseudocode, the program looks as given in Source listing 4.1. The actual C-source code can be found on the web [PBD].
Source listing 4.1 YUV to RGB program

Read input into Y
Read input into Cr420
Read input into Cb420

// Vertical scaling
for row = 0 to ROWS/2-1 do
    for col = 0 to COLS/2-1 do
        Using Cr420[row-3...row+2][col] // Assume row clamps to 0 and ROWS/2-1
        Compute Cr422[2*row][col]
        Using Cr420[row-2...row+3][col] // Assume row clamps to 0 and ROWS/2-1
        Compute Cr422[2*row+1][col]
    end for
end for
Compute Cb422 using Cb420 in the same way

// Horizontal scaling
for row = 0 to ROWS-1 do
    for col = 0 to COLS/2-1 do
        Using Cr422[row][col-3...col+2] // Assume col clamps to 0 and COLS/2-1
        Compute Cr444[row][2*col]
        Using Cr422[row][col-2...col+3] // Assume col clamps to 0 and COLS/2-1
        Compute Cr444[row][2*col+1]
    end for
end for
Compute Cb444 using Cb422 in the same way

// Compute RGB
for row = 0 to ROWS-1 do
    for col = 0 to COLS-1 do
        Using Y[row][col], Cr444[row][col] and Cb444[row][col]
        Compute R value in RGB[row][col][0]
        Compute G value in RGB[row][col][1]
        Compute B value in RGB[row][col][2]
    end for
end for
Write RGB to file

There is another version of this program derived from the original version. The aim of this loop-merged version was to minimize memory footprint by interleaving the different loops. In that case, certain array locations can be reused. For example, it is obvious that the Y, Cr444 and Cb444 arrays can be replaced by scalar variables, as each position is used only once for computing the RGB values. In the horizontal scaling, it is possible to use an array of size six and for the vertical scaling, only six rows are necessary. It is left as an exercise for the reader to come up with a correct implementation of this version of the program, if desired.\footnote{Another way to obtain this implementation is by nicely asking the people who attended the course.}

During this small assignment, Paul Meys mapped the original version of the program on both
the Cell and the GPU [Mey08], whereas I mapped the loop-merged version.

4.2 Mapping on Cell

First the loop-merged version is compiled to run on the PPE, from which the actual mapping starts. As SPES do not have access to files, the input files are read into main memory by the PPE and the output will be written to the output file by the PPE at the end of the program. After this, the loop-merged version is mapped onto one SPE. This required enlarging buffers to multiple of sixteen elements, to enable the use of DMA transfers from main memory to local store.

After the mapping on one SPE is working, the mapping is extended to multiple SPES. Given that there is one big loop, it makes sense to use data parallelism as opposed to functional parallelism, which is used by Paul. The computation of the output image is evenly split over the six available SPES, by dividing the number of rows over the number of SPES. If the number of rows is not a multiple of the number of SPES, some SPES compute an additional row of the output. This is done in such a way that the entire image is processed and that no two SPES compute the same rows of the output image.

After the mapping is done, where each SPE runs the same program but on different data, it is time to optimize the program. The following list of optimizations were applied to the SPU program:

- Loop unrolling for small loops with a constant number of iterations.
- Unrolling iterations in loops to remove if-statements that perform different computations depending whether the current row and/or column are even or odd.
- Use of conditional writes (spu_sel() intrinsic) for computing the minimum and maximum, instead of using if-statements.
- Removing division and modulo by making sure the right operand is a power of two.
- Combining the computation of the R, G and B values to perform vector operations.
- Interleaving computation and data transfers from and to LS.

---

2 It is possible to access files in SPU programs, but this is slow, as all file I/O is done with the help of the PPE.
Other optimizations, like using subword parallelism as explained in Appendix D are possible, but are not implemented due to lack of time.

4.3 Mapping on GPU

A host system reads the input files and copies them to GPU memory and after the GPU is finished, it will copy the output from GPU memory and write it to the output file. After this the mapping of the YUV to RGB program is started, using only one block and one thread and no optimizations, as this requires the least number of changes.

After it turns out that the entire YUV to RGB conversion works correctly using only one block and one thread, parallelism is introduced. Every two rows of the image are processed by a different block and each thread can process several columns. More specifically, if there are \( n \) GPU threads and the image has \( m \) columns, then thread \( x \), where \( 0 \leq x < n \), will process all columns \( y \), where \( 0 \leq y < m \) and \( y \mod n = x \). The number of GPU threads is set to 128 in the end, which appears to give the best performance after all optimizations.

After the mapping is done, several optimizations are performed. The following list of optimizations were applied to the GPU program:

- Loop unrolling for small loops with a constant number of iterations.
- Unrolling iterations in loops to remove if-statements.
- Use of GPU min and max functions to compute the minimum and maximum instead of using if-statements.
- Using textures for indexing Cr420 and Cb420 input arrays, which are cached and allow clamping of coordinates as mentioned in Subsection 3.2.2.
- Explicitly put the Cr422 and Cb422 arrays on shared memory, as the compiler puts them in slow, not cached GPU memory.
- Removing division and modulo by making sure the right operand is a power of two. This required enlargement of some buffers.
4.4 COMPARISON

The GPU does not support subword parallelism, so it is not possible to optimize by combining scalars into vectors. One additional optimization that is performed is allocating pagelocked memory on the host, which allows for fast transfers from main memory to GPU memory and back. Memory allocated in this way on the host will not be paged out by the operating system and it is possible to asynchronously start a memory transfer from or to the GPU and in the mean time perform other work on the host system if desired.

4.4 Comparison

Table 4.1: Results of the YUV to RGB conversion program

<table>
<thead>
<tr>
<th></th>
<th>Original version</th>
<th>loop-merged version</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPE</td>
<td>39 ms</td>
<td>116 ms</td>
</tr>
<tr>
<td>Cell</td>
<td>29 ms</td>
<td>18.3 ms</td>
</tr>
<tr>
<td>GPU</td>
<td>48.3 ms</td>
<td>48.1 ms</td>
</tr>
</tbody>
</table>

Now that the YUV to RGB conversion program has been mapped onto both the Cell and the GPU, it is possible to compare the running times. As a reference for the version without parallelism, the PPE is used. All times, including the times for the mappings by Paul [Mey08], are obtained by running the program on a 352x288 image and can be found in Table 4.1. The times for the GPU are actually on a NVIDIA GeForce 8600 GTS, as the GeForce 8800 GT was not yet available during this mapping. The host system containing the GPU runs Windows XP Service Pack 2, has 2 GB of internal memory and has a dual-core Intel Core 2 CPU model 6420 running at 2.13 GHz.

The reason the loop-merged version takes so much time on the PPE is due to the file I/O. This version was to use as little memory as possible, and one way to do this was by reading the Y values and writing the RGB values separately. It also appears that the GPU is worse than the Cell version, but this is due to some overhead.

For Cell, the overhead consists of the following:

3The program has also been tested using a 1280x720 image and the output byte stream was exactly the same as the original program.
4About 85–90 ms can be saved when reading all Y values and writing out all RGB values at once.
• Reading input and writing output takes 4.3 ms.

• Starting one SPE takes 1.2 ms. Starting is synchronous, so the sixth SPE is started 6 ms after the first one is started.

• Stopping one SPE takes 0.8 ms in addition to the time needed for the SPU program to finish.

For the GPU, the overhead can be categorized in the following way:

• Initializing CUDA can take between 40 and 60 ms.\(^5\)

• Allocating and freeing dynamic (page-locked) memory on the host takes 1.8 ms.

• Reading input and writing output takes 2.1 ms.

• Copying data from and to the GPU takes 0.2 ms in total.

Table 4.2: Results of the kernels of the YUV to RGB conversion program

<table>
<thead>
<tr>
<th></th>
<th>Original version</th>
<th>loop-merged version</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPE</td>
<td>35 ms</td>
<td>30 ms</td>
</tr>
<tr>
<td>Cell</td>
<td>9.1 ms</td>
<td>2.7 ms</td>
</tr>
<tr>
<td>GPU</td>
<td>2.0 ms</td>
<td>1.9 ms</td>
</tr>
</tbody>
</table>

Looking only at the programs that run on the SPE and on the GPU, the running times will be less than the ones in Table 4.1. The times excluding the overhead can be found in Table 4.2. From these times, it can be easily concluded that using both the Cell and the GPU can give high speedups, although the overhead, especially for the GPU, can be quite high. However, if the conversion program would process several images instead of one, both the Cell and the GPU will outperform the original version, especially when the PPE, when using the Cell, or the host CPU, when using the GPU, preload the next image while the SPEs and the GPU are processing the current one. On the Cell there is still opportunity for optimization as mentioned in section 4.2, so the running time may be even lower. As for the GPU, the running time will also be smaller when using the GeForce 8800 GT\(^6\) because the 8600 GTS has only 32 stream processors available.

\(^5\)A peculiarity was discovered during the timing. When connecting to the system remotely using VNC, this time was closer to 40 ms; however, when timing locally on the system, the time was closer to 60 ms. To get consistent results, all times were obtained while running the GPU version remotely.

\(^6\)The kernel time for the original version is 0.97 ms and for the loop-merged version it is 0.88 ms.
Chapter 5

Mapping

This chapter describes the mapping of certain functionality of the Irrlicht 3D Engine. First the functions that are to be mapped onto the Cell and the GPU are described, after which the mapping on the Cell is described followed by the mapping on the GPU.

5.1 Mapped functionality

As Irrlicht is quite large, the C++ code for the software renderer contains more than 16000 lines\(^1\) and because the time that can be spend on the mapping is limited, it is decided that only the functions that take the most time will be mapped onto the Cell and the GPU.

To do this, the demo application is run and profiled, as to obtain the functions that take the most time during one run of the application. During the profiling there was interaction with the program by using the mouse and the keyboard. It also turned out that having no interaction did not have much impact on the profiling results\(^2\) when running the application for several minutes.

Using the profiling information it was decided to map the three functions that take the most

\(^1\)This count only includes the code that is exclusively used by the software renderer; it also includes empty lines and comments.

\(^2\)The time needed by the functions did differ, but the ordering of the top five functions that used most time did not change.
time when running the demo application. The reason for mapping these three functions is that the time needed for these functions together was over 70% of the total application time: 38% for the first function, 24% for the second one and 11% for the third. Furthermore, the function that took most time called the second most time consuming function, so mapping the first function also required mapping the second. Finally, using these three functions it would be possible for both the author and Paul Meys to perform the mapping: the first function, and therefore also the second function\footnote{Actually, only the calls made by the first were mapped.} are mapped by me, whereas Paul is to map the third most time consuming function.

5.1.1 Description of the mapped functions

\texttt{irr::video::CTRTextureLightMap2\_M4::scanline\_bilinear2\_mag()} is what the first function is called\footnote{\texttt{irr::video} is the namespace and \texttt{CTRTextureLightMap2\_M4} is the name of the class.} and it is used for drawing the walls, the floor and the roof. The function draws one line of a triangle in an off-screen surface\footnote{At the end of each frame, this surface is copied onto the screen using the \texttt{StretchDIBits()} API call under Windows and the \texttt{XPutImage()} API call under Linux.} for an example see the orange triangle in \textbf{Figure 5.1a}. It first checks how many pixels of the current line are covered by the triangle as can be seen in red in \textbf{Figure 5.1b}, then it checks the depth buffer for the first visible pixel, see \textbf{Figure 5.1c} and \textbf{Figure 5.1d}. If there are no visible pixels, this function does no further processing, otherwise it computes all visible pixels, see \textbf{Figure 5.1e}, and updates the depth buffer where necessary, see \textbf{Figure 5.1f}. Computing the visible pixels requires two textures, one for the actual texture that is to be drawn, see \textbf{Figure 5.2a} and one for a so called lightmap, which contains precomputed light values stored in a texture, see \textbf{Figure 5.2b}. The second function, \texttt{irr::getSample\_texture()}, is called to perform texture mapping for each pixel using bilinear interpolation and the final result is obtained by multiplying all values by four, see \textbf{Figure 5.2d}. Without this multiplication the image would be darker\footnote{A guess, but perhaps the reason for using darker textures is to obtain better compression when storing textures.}, which is shown \textbf{Figure 5.2c}.

\texttt{irr::video::CTRTextureGouraudNoZ2::scanline\_bilinear()} is how the third function is called. It draws one line of a triangle in an off-screen surface and is used for drawing the background of the world, which can be thought to be infinitely far away. Unlike the
5.1. MAPPED FUNCTIONALITY

(a) Entire triangle

(b) Pixels that are to be drawn

(c) Depth buffer and interpolated depth values of triangle

First visible pixel

(d) First visible pixel

(e) Updated output values

(f) Updated depth buffer

Figure 5.1: Drawing one line of a triangle
first function, it uses nearest neighbour interpolation for texture lookups and does not use nor
update the depth buffer.

The functions that call the first and third most time-consuming functions both split a triangle
into two parts and calculate the pixels from top to bottom. The first part is made up of the vertex
with the highest and second highest $y$-coordinate and the second part is made up of the vertex
with the second highest and third highest $y$-coordinate$^7$ see the grey line in Figure 5.1a. This is
done to ensure that the slope which is used to calculate the start of the $x$-coordinates is correct
and also to use the correct texture coordinates for texture mapping.

$^7$In the case that two vertices have the same $y$-coordinate, there is only one sub-triangle to process.
5.2 Mapping on the Cell

Some measurements are performed before the actual mapping. The first function is timed while displaying the start screen, which can be seen in Figure 5.2d and is used as a reference point, and the average time needed for the function to complete is 8.5 $\mu$s on the PPE. The overall performance of the application is measured using the number of triangles drawn per second, which is around 39 thousand.

```
Source listing 5.1 For-loop containing a call to the first function
for y = yStart to yEnd do
    set parameters for function
    call irr::video::CTRTextureLightMap2_M4::scanline_bilinear2_mag()
    update parameters
end for
```

When the entire first function is offloaded onto one SPE, all relevant data must be put into the LS of this SPE. Putting this data without performing any operations on one SPE already takes over 6 $\mu$s. Rather than mapping only the function itself, the for-loop containing the function, which is described in Source listing 5.1, is mapped. The rationale behind this is that the communication time for the small number of updated variables for the next line of pixels can be quite large and this overhead will be smaller if there is less communication. Measuring the time of the for-loop results in an average of 40 $\mu$s; putting relevant data into the LS of one SPE without doing anything takes around 9 $\mu$s on average.

The first version of the mapping of the for-loop onto the SPE takes on average 153 $\mu$s and the application can only draw 14 thousand triangles per second. This is expected as each line of the depth buffer and the off-screen surface is transferred from main memory to LS and back. With a working version on the SPE, it is time to perform optimizations.

5.2.1 Optimizations

The first optimization is the use of a software managed cache for the textures. The idea is that whenever location $(x, y)$ is accessed in a texture, it is likely that it is accessed again for the computation of the next pixel. Even if this is not the case, then it is likely that one of

---

8 The PPE sends pointers to the SPE, after which the SPE starts a DMA transfer.
9 The Cell SDK contains a software managed cache API, see section E.1 for more information.
the locations \((x + 1, y)\), \((x, y + 1)\) or \((x + 1, y + 1)\) is accessed again\(^{10}\), so having a copy of the texture value in the LS rather than retrieving each texture value from main memory is likely to have performance benefits. This cache is configured using trial and error and by collecting statistics and optimized such that the cache hit ratio is over 99%\(^{11}\). There are indeed performance benefits: the for-loop on one SPE takes only 44.7 \(\mu s\), with the demo application drawing 35 thousand triangles per second.

Obviously there are other optimization strategies besides using a software cache. The second optimization is the use of subword parallelism. Subword parallelism is possible as there are some operations on different scalar values that could be combined into vector operations. The for-loop on one SPE takes 34.8 \(\mu s\) on average with the application drawing 41 thousand triangles per second after this optimization.

### Source listing 5.2 Integer multiplications

```c
for (y = yStart; y <= yEnd; ++y)
{
    z = width * y;
    Do other things...
    Use z
    Do other things...
}
```

### Source listing 5.3 Integer multiplications replaced

```c
for (y = yStart, z = width * yStart; y <= yEnd; ++y, z += width)
{
    Do other things...
    Use z
    Do other things...
}
```

Another optimization is to use less integer multiplications if possible, because the SPE only supports 16-bit integer multiplications natively\(^{12}\), causing 32-bit integer multiplications to take multiple instructions. One place where this optimization is possible is the one given in Source listing 5.2, which can be replaced by Source listing 5.3. The for-loop on the SPE takes 34.4 \(\mu s\) on average after this change, but the number of triangles being drawn did not change.

The optimizations continue with copying data back to the PPE only if the data is changed. If the entire line of pixels is not visible according to the depth buffer, no data is copied back from LS to

---

\(^{10}\) Triangles are drawn from top to bottom and from left to right

\(^{11}\) Although it was possible here, in general it may not be possible to get a cache hit ratio that is above 99%, if only because the amount of LS is limited.
main memory either. Furthermore, only pixels\(^{12}\) between the first and the last pixel in a line are transferred to the LS and transferred back to main memory if needed. After this optimization, the for-loop takes 31.9 \(\mu s\) on average and the number of triangles being drawn increases to 43 thousand triangles per second.

![Integer Part](image1.png) ![Fractional Part](image2.png)

Figure 5.3: Fixpoint number

Following this optimization is the conversion from fixpoint math to floating point math. The software renderer makes use of fixpoint math to avoid floating point operations. Each fixpoint number is 32-bits large and has 9-bits for the fractional part as can be seen in [Figure 5.3]. The use of fixpoint numbers requires shifting and masking for operations like multiplication\(^{13}\) and extraction of the integer or fractional part. The SPE can do floating point operations quite efficient, so the conversion to floating point math should increase the performance, also because the SPE supports a fused multiply-add instruction. It turns out that the for-loop takes 22.3 \(\mu s\) on average after this optimization is done and the number of triangles per second increases to 52 thousand.

The above mentioned optimizations were all done using one SPE. Optimizations using multiple SPEs, where each SPE is assigned an ID starting from 0, are the next step. The idea is that each triangle is processed by all \(n\) SPEs, where SPE \(x\) is to process all lines \(y\) of the triangle, where \(y \mod n = x\). Using only two SPEs, with the PPE sending the pointers to the relevant data to all SPEs in sequential order as can be seen in [Figure 5.4a], results in an average time of 19.6 \(\mu s\) and a total of 53 thousand triangles to be drawn per second.

An alternative would be to have each SPE send the necessary pointers to the next SPE, except for the last one, as can be seen in [Figure 5.4b]. The first SPE will obtain the pointer from the PPE. A reason why this can be faster is that the libspe library used to send messages from

\(^{12}\)Actually, it is possible that more pixels are transferred as to obtain correct alignment and size for DMA transfers.

\(^{13}\)Fixpoint numbers use a 32-bit integer type and multiplication thus takes multiple instructions.

\(^{14}\)On average, there are 15 lines per triangle.
the PPE to SPEs is quite slow. Another reason is because the SPEs can issue a DMA transfer for sending the pointers as messages and, without blocking on this first transfer, it can then issue the request for a DMA to receive the necessary data from main memory. Using two SPEs results in an average time of 14.3 $\mu$s and a total of 63 thousand triangles per second for the entire application.

Table 5.1: Mapping on the Cell

<table>
<thead>
<tr>
<th>(a) Mapping of first function</th>
<th>Triangles (Thousands per second)</th>
<th>Time ($\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPE</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>1 SPE</td>
<td>52</td>
<td>22.3</td>
</tr>
<tr>
<td>2 SPE</td>
<td>63</td>
<td>14.3</td>
</tr>
<tr>
<td>3 SPE</td>
<td>67</td>
<td>11.6</td>
</tr>
<tr>
<td>4 SPE</td>
<td>68</td>
<td>10.4</td>
</tr>
<tr>
<td>5 SPE</td>
<td>69</td>
<td>9.8</td>
</tr>
<tr>
<td>6 SPE</td>
<td>70</td>
<td>9.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) Mapping of third function</th>
<th>Triangles (Thousands per second)</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPE</td>
<td>39</td>
<td>3</td>
</tr>
<tr>
<td>1 SPE</td>
<td>41</td>
<td>1.99</td>
</tr>
<tr>
<td>2 SPE</td>
<td>42</td>
<td>1.00</td>
</tr>
<tr>
<td>3 SPE</td>
<td>43</td>
<td>0.70</td>
</tr>
<tr>
<td>4 SPE</td>
<td>43</td>
<td>0.51</td>
</tr>
<tr>
<td>5 SPE</td>
<td>43</td>
<td>0.41</td>
</tr>
<tr>
<td>6 SPE</td>
<td>43</td>
<td>0.34</td>
</tr>
</tbody>
</table>

Measurements for up to six SPEs are performed using the just mentioned mapping on two SPEs. The results are summarized in Table 5.1a and Figure 5.5 and we can see that it does not scale.

\[^{[1]}\text{Having the PPE wait for a mailbox message from an SPE is about 5.5 } \mu\text{s slower than checking a volatile variable in main memory and having the SPE change it using an atomic operation.}\]
5.2. MAPPING ON THE CELL

5.2.1 Mapping on the Cell well

For completeness sake, the results for the mapping of the third function by Paul Meys are given in Table 5.1b.

5.2.2 Alternative mapping

The approach in the previous subsection uses very fine-grained parallelism. Rather than processing each triangle by each SPE, an alternative approach would be to process each triangle by a different SPE. There are some challenges when using this approach though.

First, not every triangle uses the same textures, so the number of textures to be stored will increase by the number of SPEs. Storing these textures should be no problem in Irrlicht.

\[\text{It is noticeable that a function which takes quite some time to complete has bigger speedups when increasing the number of SPEs, as the overhead of copying data is only a small part of the total time.}\]
CHAPTER 5. MAPPING

(a) Two triangles
(b) Rasterization

(c) Incorrect rendering
(d) Correct rendering

Figure 5.6: Drawing two triangles in parallel

(a) Two triangles
(b) SPE1 locks first line and SPE2 stalls
(c) SPE1 locks second line and SPE2 locks first line
(d) SPE1 locks third line and SPE2 locks second line
(e) SPE1 locks last line and SPE2 locks third line
(f) SPE1 is done and SPE2 locks last line

Figure 5.7: Illustration of locking lines
5.2. MAPPING ON THE CELL

A bigger problem is that both the offscreen surface and the depth buffer can be updated on the same position by different SPES, as each SPE has its own copy of the current line in its LS. For an example, see Figure 5.6. In Figure 5.6a, two triangles are to be drawn, each by a different SPE. Figure 5.6b shows the rasterized pixels of the triangles, where the grey colour indicates overlap. If the yellow triangle is rendered in parallel with the red one and both SPES use their own copy of the depth buffer in their LS, then the rendering can be incorrectly done as shown in Figure 5.6c. A correct rendering is shown in Figure 5.6d.

To overcome the second problem, there has to be some form of synchronization either between the SPES or on main memory. Perhaps it is possible to use atomic operations or to use a Mutual Exclusion (mutex) object.

This alternative approach has been tried by locking and unlocking lines[17] in the offscreen surface and the depth buffer and having each SPE process a different triangle, as illustrated in Figure 5.7. The PPE dispatches the triangles to the SPES in a cyclic order and it turns out that this approach is slower than the initial approach. Explanations for this are the cost of sending messages to the SPES, the locking and unlocking of lines using atomic operations and the requirement of the SPE to wait for the MFC to finish copying before unlocking a line.

A third approach would be dividing the screen by the number of SPES and have each SPE render only one part. The advantage of this approach is that no two SPES will render the same line, thus no locking is needed. A disadvantage is, in case of large triangles, that multiple SPES may still need to take part in rendering one triangle, requiring extra copies of the parameters for other SPES. Also, the order in which the triangles are rendered is not specified; it could be the case that all triangles in one part of the screen are rendered first, followed by triangles in another part of the screen. It is possible that one SPE will be doing a lot of work, while the other SPES are idling. This approach has not been tested, mainly due to the large amount of time already spent on this mapping. If the triangles are issued in a more or less random order, then this approach may work well.

[17] One array containing height 32-bit elements is used as the mutex object, where a value of 0 denotes an unlocked line and 1 denotes a locked line.
5.3 Mapping on the GPU

Like in the previous section, the for-loop containing the function will be mapped onto the GPU. Measuring the time of the for-loop on the computer containing the GPU results in an average of 11.5 $\mu$s, with the demo application averaging around 137 thousand triangles per second.

Transferring all necessary data each time the for-loop is called, but not performing any actions on the GPU, results in only 22 thousand triangles being drawn. It would not pay off to offload the function to the GPU if the transfers already slow things down a lot. Because of this, instead of starting the mapping onto the GPU, first the transfers between system memory and GPU memory are minimized. These transfers are necessary because the GPU has no access to system memory, but only to GPU memory, as illustrated in Figure 5.8.

If only the data that changes for every triangle is sent to the GPU, and the GPU does not execute the function yet, the number of triangles is around 161 thousand per second with the for-loop containing function taking around 9.0 $\mu$s. It should be clear that the transfer for this data is necessary each time, as the parameters are unique for each triangle.

The next step is to minimize the transfers of textures to the GPU. In order to do this, each texture will be send at most once to the GPU; the next time the texture is used, the pointer in the GPU memory is reused. Source listing 5.4 contains pseudocode to do this. As the GPU actually knows the concept of textures, the texture will be bound on the GPU every time it is

---

18 See section E.2 for a comparison between the PPE and the Intel Q6600 CPU.
19 Keep in mind that the GPU is not performing any work yet.
20 The actual implementation uses the C++ std::map container that maps a pointer in system memory to a pointer in GPU memory and returns a NULL pointer if the pointer on the GPU has yet to be allocated.
5.3. MAPPING ON THE GPU

Source listing 5.4 Minimizing transfers for textures

```plaintext
if texture is not in ListOfTextures
    Allocate memory on GPU
    gpuPtr = allocated memory on GPU
    Copy texture data to gpuPtr
    ListOfTextures[texture] = gpuPtr
end if

Use ListOfTextures[texture] as texture on GPU
```

needed, instead of copying texture parameters\(^{21}\) to the GPU each time the texture to be used changes. After this, the number of triangles per second is around 148 thousand, while running an empty for-loop on the GPU takes around 9.0 µs on average.

Source listing 5.5 Minimizing transfers for offscreen surface and depth buffer

```plaintext
if first triangle in frame or previous draw function != this function
    Copy surface and depth buffer to GPU memory
end if
Perform work...
if last triangle in frame or next draw function != this function
    Copy surface and depth buffer from GPU memory
end if
```

Following that, the number of transfers for the offscreen surface and the depth buffer are minimized. This is done by only copying the data to GPU memory, if another function potentially changed the values and only copying the data back right before another function will potentially change values in either the surface or depth buffer or both. Pseudocode for this can be found in Source listing 5.5\(^2\). The number of triangles drawn by the application is down to 144 thousand on average, while the duration of the for-loop containing this function is around 9.1 µs. Again the for-loop does not perform any work on the GPU yet.

Source listing 5.6 For-loop on GPU

```plaintext
for(y = yStart; y <= yEnd; ++y)
{
    Do stuff...
    abc += y;
}
```

With the transfers minimized and because there is room, albeit relatively small, for improvement, the mapping process on the GPU is started. Trying to run the function after the initial

\(^{21}\)For example, the width and the height of the texture.

\(^{22}\)The checks for the copying are actually done by a parent function. If a lot of triangles are to be drawn by this function, then it can be easily seen that the checks, which will mostly return false, are unnecessary most of the time and will unnecessarily slow down the program.
mapping on the GPU results in the following error message: “Unspecified launch failure”, after the computer does not respond for a few seconds. Either some memory access in the mapped function is wrong or the launch times out\footnote{The release notes for the Windows version of the CUDA toolkit mention a time-out of five seconds if the GPU is also used for displaying the screen to the user. This time-out is caused by a watchdog timer in Windows that kills seemingly unresponsive processes occupying the GPU.}. To reduce the running time of the function, the for-loop for drawing each line of the triangle is removed; the number of blocks is set to the number of iterations in the for-loop and each block is used to draw each line instead. See also Source listing 5.6 and Source listing 5.7.

Furthermore, the textures are changed to use the texture units on the GPU. Again this is to reduce the running time, as texture lookups on the GPU are cached. Besides reducing the running time, this part is one of the few that accesses memory, so it may have been the culprit in causing the error message.

After these two changes, the mapped function actually runs, although it is a bit slow. Using only one thread, the number of triangles drawn per second averages 27 thousand, while the function takes around $86.2 \mu s$ on average.

### 5.3.1 Optimizations

With the function mapped onto the GPU, it is time to perform optimizations. Whether these optimizations will have a large impact on the running time remains to be seen, as the mapped function is already much slower compared to the one running on the CPU.

The first optimization is to use shared memory for storing the computed parameters of the current line on the triangle that is to be drawn. The reason for using this optimization is that shared memory is very fast if there are no bank conflicts and because it supports a non-programmable broadcast mechanism if the same address is accessed by multiple threads in a warp\cite{NVI07}. Using this optimization, the number of triangles drawn by the application is around 32 thousand per triangle, whereas the running time gets down to $71.6 \mu s$. 

Source listing 5.7 No for-loop on GPU

```
abc += blockIdx.y - yStart;
Do stuff...
```
The second optimization uses so-called coalesced memory accesses, that is accessing global memory in a particular way to obtain higher bandwidth as explained in section E.3 when loading parameters for the triangle from global memory into shared memory; these parameters are then used to compute the parameters for the current line. When using 32 threads for the coalesced memory access and using only one thread for the rest of the mapped function, the number of triangles stays the same, whereas the running time decreases to 70.3 $\mu$s.

Source listing 5.8 Texture containing unsigned int elements

```cuda
texture<unsigned int, 2, cudaReadModeElementType> textureRef;

unsigned int val = tex2D(textureRef, x, y);
r = (val & 0x00FF0000) >> 16;
g = (val & 0x0000FF00) >> 8;
b = (val & 0x000000FF);  
```

Source listing 5.9 Texture containing uchar4 elements

```cuda
texture<uchar4, 2, cudaReadModeElementType> textureRef;

uchar4 val = tex2D(textureRef, x, y);
r = val.z;
g = val.y;
b = val.x;  
```

After careful examining of the source code, the next optimization changes the textures to use the `uchar4` data type instead of the `unsigned int` data type. This removes explicit masking and shifting to obtain the red, green and blue values; see also Source listing 5.8 and Source listing 5.9. This optimizations results in a decrease of the running time to 67.3 $\mu$s, whereas the number of triangles increases to 34 thousand per second.

With the previous optimization done, it is possible to do a fixpoint to float conversion, as was also done during the mapping on the SPEs. By using the `uchar4` data type and by setting the third template parameter of the texture declaration to `cudaReadModeNormalizedFloat` instead of `cudaReadModeElementType`, it is possible to obtain floating point numbers in the range $[0.0, 1.0]$ when performing texture lookups. Like in the mapping on the Cell, this gave a performance increase: 37 thousand triangles per second on average for the application and an average running time of 60.2 $\mu$s.

The next optimization is to use coalesced memory accesses for finding the first visible pixel.

\[^{24}\text{A structure containing four elements of type unsigned char: x, y, z and w.}\]
\[^{25}\text{In this case, the integer value x is mapped to floating point value } \frac{x}{255}.\]
using the depth buffer, see Figure 5.1d, for computing the updated values in the depth buffer, as shown in Figure 5.1f, and for computing the output values, see Figure 5.1e. Converting the latter two to coalesced accesses is relatively easy, as there is no dependency on the previous or the next pixel. Converting the first is trickier, as finding the first visible pixel uses a linear search and is done sequentially. This is solved by reading the depth buffer values by all threads into shared memory and having the first thread search for the first visible pixel. With this optimization, the number of triangles in the demo application increases to 48 thousand per second and the running time averages 43.4 μs.

As coalescing gives higher bandwidth, it makes sense to increase the number of threads to see if there are any benefits. It is possible that there are no benefits, as threads have to wait for each other as they can potentially overwrite the value in shared memory, before the first thread could use the old value for finding the first visible pixel. Increasing the threads with 32 also results in higher performance, namely 49 thousand triangles and 43.0 μs, however using 96 threads results in a performance decrease. In the rest of the mapping, 64 threads are used.

The optimization that is performed after this uses the texture units on the GPU. When using cudaReadModeNormalizedFloat in 2D textures, it is possible that results from texture lookups are obtained using bilinear filtering. So using the texture units on the GPU, it is not necessary anymore to do four texture lookups and the linear interpolation in the source code of the GPU program. When using the bilinear filtering, the addressing mode should be set to wrap, as that is used by the Irrlicht engine. To use a custom addressing mode, the texture lookups must be done using normalized coordinates, which requires either floating point division by the width and the height, or floating point multiplication with the precomputed inverse. Using the texture units the running time is only 42.4 μs on average, whereas the number of triangles for the application does not change.

The last optimization consists of removing the sequential part of looking for the first visible

\footnote{Using the \texttt{__syncthreads()} intrinsic.}

\footnote{Recall that a warp consists of 32 threads.}

\footnote{Out-of-range texture coordinates will be wrapped to a valid range.}

\footnote{Addressing textures with values between \([0, 1]\) instead of \([0, width - 1]\) and \([0, height - 1]\).}

\footnote{There was a mistake in the mapping which gave wrong results when using normalized textures. The reason for this is that \texttt{height} by \texttt{width} \texttt{sizeof(uchar4)} textures were allocated instead of just \texttt{height} by \texttt{width} textures. The confusion for this is that when copying textures to GPU memory, \texttt{height} \texttt{width} \texttt{sizeof(uchar4)} bytes must be specified.}
5.3. MAPPING ON THE GPU

pixel in the depth buffer by the first thread. Instead of storing the depth buffer value in shared memory, the threads set a value in an array stored in shared memory whenever the pixel is visible, that is whenever the comparison with global memory succeeds. The array contains as many elements as the number of threads per block and is of type int, to avoid shared memory bank conflicts. Initially, all values in the array are the same: the number of threads per block. If a thread determines that a pixel is visible, the value in the array location corresponding to that thread is set to the thread ID. Finally, multiple threads will determine the minimum value in the array together using a so called reduction operation, see section E.4 [Har07] [Fer04]. If this minimum value is less than the number of threads per block, and thus corresponds to a valid thread ID, that thread found the first visible pixel, otherwise the next set of values in the depth buffer is tested. This optimization uses a lot of synchronization, but is still expected to be faster than using only one thread to compute the first visible pixel. This turns out to be the case, although the gain is quite low: an average of 51 thousand triangles per second for the demo application and the average running time of the function is 40.9 µs. Results for several threads are given in Table 5.2a and Figure 5.9. For completeness sake, the results for the mapping by Paul have been summarized in Table 5.1b.

Table 5.2: Mapping on the GPU

<table>
<thead>
<tr>
<th></th>
<th>Triangles (Thousands per second)</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>137</td>
<td>11.49</td>
</tr>
<tr>
<td>32 threads</td>
<td>50</td>
<td>41.1</td>
</tr>
<tr>
<td>64 threads</td>
<td>51</td>
<td>40.9</td>
</tr>
<tr>
<td>128 threads</td>
<td>50</td>
<td>41.4</td>
</tr>
<tr>
<td>256 threads</td>
<td>48</td>
<td>43.0</td>
</tr>
</tbody>
</table>

As all known optimizations are performed, it can be seen that offloading this function to the GPU is quite slow. It turns out that every time the GPU function is called, the number of blocks is around 15 on average, whereas at least two blocks per group of eight SPs are recommended.

---

31Reads in global memory are still coalesced.
32Ranging from 0 to the number of threads per block minus one.
33The number of values tested in parallel is normally equal to the number of threads per block, but can be less for the first time, to fulfil the coalescing requirements, and the last time, as each line has a finite number of pixels.
to hide idling during synchronization and to hide the latency of memory accesses [NVI07].
This approach is not a good one for this particular function, unlike the mapping of the third
function by Paul. In the third function the triangles have an average height of 480, so there are
480 blocks.

Furthermore, it turns out that running an empty function on the GPU, including the transfers
of the variables making up a triangle[^34] and including the parameters for the entry function
call[^35] takes 37.8 μs, so attempting to be faster than the CPU is quite impossible using the “one
triangle per function call approach”. If the communication overhead is ignored however, we
can see that the computation time is around 3 μs.

[^34]: Without these transfers, the time is down to 31.1 μs.
[^35]: If the parameters, for example pointers to GPU memory, are excluded, and the transfers to the GPU are ignored,
the startup time of the function can be found and appears to be around 21.8 μs.
5.3.2 Alternative mapping

An alternative approach is to process multiple triangles at the same time. Each triangle will be put into a queue and the entire queue of triangles will be processed once there are \( n \) triangles in the queue\(^\text{36}\).

In this approach, it is not possible to replace the for-loop that draws each line of the triangle by the number of blocks. The reason for this is that the number of lines per triangle can differ, as can be seen in Figure 5.10. Obviously, this can be worked around by taking the maximum height, which is \( \max(y_{End_i} - y_{Start_i} + 1) \) for \( 1 \leq i \leq n \), and do nothing if the current block number is larger than the height of the triangle, see Source listing 5.10.

![Figure 5.10: Each triangle can have its own height](image)

Source listing 5.10 Skipping lines exceeding the height of a triangle

```cpp
// Start of GPU program
if (blockIdx.y > yEnd[i] - yStart[i] + 1) return;
y = yStart[i] + blockIdx.y;
...
```

As for the number of triangles, this can also be given as an argument in the number of blocks, seeing as the number of blocks can be a 3D-argument. This also requires sending the start- and the end-values for the y-coordinate for each triangle, and each triangle can potentially use a different texture. The use of different textures is a limitation for mapping on the GPU: empirical research shows that the number of active textures on the GPU is limited to eight at a given

\(^{36}\)In the case of the last few triangles, it is possible that less than \( n \) triangles are processed.
In other words, the maximum number of triangles that can be processed in parallel is four, which should give up to 60 blocks on average for each invocation of the function.

**Source listing 5.11** Determining whether output should be computed

```c
if (val >= z[i])
{
    z[i] = val;
    ...
    output[i] = someValue;
}
```

**Source listing 5.12** Using atomic operations to compare a value against the depth buffer

```c
if (val >= atomicMax(z[i], val))
{
    ...
    output[i] = someValue;
}
```

It should be clear that when rendering multiple triangles simultaneously, it is possible that some triangles overlap, as is the case in Figure 5.10. Luckily, the NVIDIA GeForce 8800 GT supports atomic operations, as was mentioned in subsection 3.2.4, which can be used for comparing and setting the value in the depth buffer. So the original code, which is shown in **Source listing 5.11**, can be replaced by **Source listing 5.12** if only the atomic operations on the GPU would support floating point numbers.

Unfortunately (most) atomic operations are not supported when using floating point numbers, but given the fact that the depth value in this implementation is not negative and given the fact that the representation of the floating point numbers excluding exceptional values is monotonic increasing, it is possible to work around this problem.

Using the `__float_as_int()` intrinsic, it is possible to obtain the bit representation of a floating point number, after which it is possible to use the desired atomic operation. As the depth value is at least 0 in this implementation, taking the maximum works for almost all cases. The only exceptions are NaNs. If the input contains at least one NaN, which should result in

---

37 Attempting to use more results in error messages by CUDA.
38 Recall that each triangle needs a texture and a lightmap.
39 Unlike the two’s complement representation of integers, floating point numbers use a sign-and-magnitude representation.
40 Infinity and NaNs.
41 Numbers with higher values have a larger exponent or, if the exponent is the same, a larger significand.
42 The result of any operation yielding an NaN on the GPU is equal to the bit pattern 0x7FFxFFFF [NVI07].
false in the comparison given in Source listing 5.11, the output will be wrong. Luckily, the computation for the value used in the comparison appears to yield no NaNs so using the atomic operation should not be a problem in this specific case.

After all these changes, it turns out that passing the start- and end-values for the y-coordinates and the texture width and height as arguments to the GPU program takes quite an amount of time, as each block will need access to these variables. Transferring these parameters into GPU memory asynchronously instead of passing them as arguments also requires a significant amount of time. Processing up to four triangles with this approach takes significantly more time than the original approach: the transfers are the bottleneck.

Since the GPU can actually outperform the CPU in today’s graphics, there must be some explanation for these results, which are quite bad. The first difference between the mapping and the actual hardware acceleration of the graphics pipeline lies in the part of the graphics pipeline that is mapped. On the actual GPU the entire pipeline is accelerated, whereas in the mapping, only parts of the rasterization and texture mapping stage are run on the GPU. This also means that in the latter case there are many transfers needed from and to the GPU in order to obtain correct results, obviously killing performance. Furthermore, in the mapping only programmatic access is allowed to the SPs, whereas the GPU contains ROPs among other units, which work independently from the SPs. It is also the case that CUDA is quite new, whereas the GPU has supported hardware acceleration for quite some time now, meaning that the latter case has had more time for performance enhancements. As for the software renderer, it is optimized for normal CPUs, which may not be the ideal way when mapped on a GPU.

Although the results for the first function are somewhat disappointing on the GPU, it should be noted that the third function performs quite well. The software renderer is still seen as a good representation of the graphics pipeline, especially when the transfer times are ignored. The

---

43The source code appears to have no operations that can yield NaNs and the result on the GPU appears to be correct when using the atomic operation.

44Recall that normalized texture coordinates are used, requiring division of texture coordinates by the width and the height.

45In particular, it may be necessary to transfer the output image from system memory to GPU memory and back more than once. At the end of the frame, when all triangles are rendered, the entire image has to be send to the GPU again so that it can be displayed on a screen.

46For example fixpoint numbers are used instead of floating point arithmetic.

47Recall that in case of the first function, the transfer time are 37.8 µs, whereas the total time with 64 threads is 40.9 µs. Also note the speedup for the third function, even when the transfers are included.
justification for ignoring the transfer times is the following: if the entire graphics pipeline is mapped, then the data that is transferred can be computed using the triangle’s vertices.

With the results of the preparation in the previous chapter and the mapping in this chapter, the advantages and disadvantages of both architectures can be assessed, completing the preparation for a proposal of a new architecture.
Chapter 6

Proposal

This chapter contains the proposal for a new architecture, based on the results from the previous section. First the two architectures are evaluated, describing their advantages and disadvantages, followed by the actual proposal.

6.1 Evaluation of Cell and GPU

First it should be mentioned that programs that can be parallelised entirely have a huge advantage when using either the Cell or the GPU architecture. This can be easily concluded from the results when mapping the YUV to RGB program as can be seen in section 4.4.

The SPEs can access main memory by using their MFC, allowing them to initiate transfers to their LS. The GPU on the other hand has a large amount of video memory available, but unfortunately, all transfers must be initiated from the host system. It is not possible to dereference pointers to system memory on the GPU.

Another advantage of the Cell architecture is that each SPE can run a different program if desired, allowing functional parallelism. In case of the GPU, this is not possible, however this effect can be simulated by using if-statements on the block and/or thread ID, possibly enlarging the total size of the GPU program. Both architectures allow data parallelism; in case of the Cell

\footnote{This may be a limitation of CUDA; perhaps there is an undocumented way to do this.}
processor, it requires sending the same program to all SPEs, whereas on the GPU one program is sent together with the amount of blocks and threads per block.

Each SPE has an MFC which can retrieve data from main memory, or from the LS of other SPEs. On the GPU, memory accesses will stall the current thread(s) and other threads, if there are threads that are not blocked, can run instead. On the Cell, this means that one SPE can queue several memory transfers from main memory to its LS, whereas on the GPU, a high number of concurrent threads is needed to lower the amortized cost of memory accesses.

Each LS in an SPE can only hold 256 KB of instructions and data, which the SPU can access. The GPU has several kinds of memory, like shared memory, which can hold 16 KB of data and is very fast. Texture memory and constant memory are cached, other accesses to video memory are not cached and can be slow. Having several kinds of memory can be an advantage, as there are more optimization possibilities, however, it can also be a disadvantage, as it is necessary to exploit these kinds of memory explicitly in order to obtain higher performance.

SPEs can communicate with the PPE or other SPEs via mailboxes or signals. The GPU allows synchronization between threads in one block, but there is no global synchronization between the threads in multiple block. Reason for this is that it may be possible that not all blocks are running, for example there are not enough resources to run more than a given amount of blocks.

The GPU has texture units which work independently from the SPs, allowing the SPs to perform other computations while a texture lookup is performed. Of course, it has to be the case that there are enough computations available in order to do so.

Both the SPEs and the GPU do not support 32-bit integer multiplies natively, requiring multiple instructions to perform the multiply operation. In the case of the graphics pipeline, this is not a big issue though, as the number of integer multiplications is quite low.

Branches can be quite expensive on the SPEs, if they are predicted wrongly. In order to remove these potential stalls, for example while looking for the first visible pixel, either branches should be removed, or so-called branch hints can be used. For the GPU branches can be quite cheap if all threads in the same warp take the same branch. If this is not the case, all threads must process multiple branches, masking out the branch that should not be taken.
A nice instruction on the SPU is the ability to specify a power-of-two scale factor when converting an integer to single precision floating point or back. In the first case the resulting floating point number is the integer divided by the scale factor and in the second case the resulting integer is the floating point number multiplied by the scale factor. An example where this can be used is converting colour represented by a floating point number between 0.0 and 1.0 to a colour represented by an 8-bit integer.

On the software side, it is nice to know that the SPEs can be programmed in C or C++, whereas the GPU supports only a subset of C, most notably the lack of recursion. During the mapping, this was not a limitation, as there was no recursion, however in the general case, this may be a problem. In case of tail recursion, this problem can be worked around by converting the recursion to a loop; in other cases a goto may be used. When using a goto statement instead of a recursive function, care should be taken not to assume copy-by-value semantics for the variables used as arguments in the original function. In particular, arguments are not shadowed out by the next invocation of a recursive function call in this workaround.

In the case of the GCC version for the SPU compiler, which is used throughout this project, it is unfortunate that there is no optimization level that can vectorize scaler code automatically. For the GPU this is not necessary, as it does not support subword parallelism.

The simulation tools in the Cell SDK do not support programs with a Graphical User Interface (GUI), which is unfortunate in the case of mapping the demo application to the Cell, as these simulation tools support cycle accurate measurements. On the other hand, the CUDA toolkit does not contain a simulator, although it is possible to emulate the GPU program on the host system. The emulation has some disadvantages as it is quite slow and does not catch all errors.

Finally, it should be mentioned that for functions which take very little time to compute, the communication overhead is relatively high in both the Cell and the GPU. Minimizing transfers can therefore become a requirement of the mapped program in order to achieve maximum performance. If the transfer latency can be made very low, then it is possible to concentrate on

---

\(^2\) Actually, this requires the floating point number to be a little smaller than 1.0 in order to map the full range to 0 and 255 instead of 0 and 256.

\(^3\) Each GPU thread is mapped to an actual thread in the OS.

\(^4\) In emulation it is possible in the GPU program to dereference pointers that point to data in system memory.
the mapping of the program instead of also worrying about the transfers.

6.2 Ideas for the proposal

The new architecture should incorporate a more or less general purpose processor, like the PPE in the Cell architecture, and an as of yet unspecified number of PEs. The reason for this is to eliminate copies between memory corresponding to the general purpose processor and memory used by the PEs. These copies can be quite expensive as seen during the mapping on the GPU.

![Figure 6.1: Division of output image over the number of PEs](image)

One of the problems with drawing multiple triangles concurrently are potential race conditions, where two or more PEs may write to the same location in memory at the same time. One way to combat this problem is by requiring that one memory location can only be written to by one PE or, which is an even stronger requirement, one memory location can only be accessed by one PE. Given that the implementation of the graphics pipeline draws each line of a triangle separately, it is possible to divide the height of the output image over the number of PEs and have each PE process one or multiple lines of the output image.\(^5\) An advantage of this approach is that it scales better than the one used during the mapping, if the number of lines per triangle is rather low. Two possible ways to perform this division are given in Figure 6.1.

\(^5\)Note that this is different than the approach taken during the mapping, where the lines of the triangles are split over the SPEs and the SPSs instead of lines of the image.
Memory latency can be quite high for off-chip memory. To accommodate this problem, it may be necessary to cache all texture lookups, either by having one cache per PE, one cache for all PEs or one cache shared by several PEs. The first approach will require as many caches as there are PEs, requiring a high cost when increasing the number of PEs. Given that each PE has its own cache, this can result in high performance. The second approach on the other hand may require one large cache, such that it will not have to deal with cache misses all the time. As multiple PEs use the cache, there may be a contention problem, which can result in high latency. The last approach is a tradeoff between the first and second approach; the number of PEs sharing one cache and the size of the caches is yet to be determined.

Another attempt to decrease the high memory latency, is for each PE to have one or more scratchpad memories, which is local to that PE. For example, the part of the depth buffer corresponding to the lines processed by one PE, will never be modified by another PE. The size needed for part of the depth buffer is proportional to the size of the output image and inverse proportional to the number of PEs. Instead of a separate cache for textures, it may also be possible to use part of this scratchpad memory to cache textures.

The output image can also be split into several local scratchpad memories. However, at the end of each frame, the entire image needs to be transferred to the screen. This transfer can induce some latency, as the scratchpad memory cannot be both read from and overwritten at the same time. A solution would be to introduce double buffering, requiring two local scratchpad memories for part of the output image. Although this requires more memory, it is possible to send the frame to the screen and at the same time compute (parts of) the next frame. Perhaps it is even possible to use a memory controller for sending the image from the local scratchpad memories to the screen directly, bypassing the off-chip memory completely. In this case each PE should process several lines consecutively as shown in Figure 6.1a, enabling it to use one large block transfer for sending to the screen, rather than sending several smaller lines of the output image which are computed as shown in Figure 6.1b. Note that sending data to the screen has a strict deadline. If a refresh rate of 60 Hz is assumed,

---

6 Recall that the software cache for textures during the mapping on the Cell had a hit ratio over 99%. In the first function there are 64 sets of 256 byte cache entries for the first texture and 8 sets of 16 byte cache entries for the second texture.

7 Assuming single-ported memory.

8 Unless displaying two frames that are somehow interleaved at the same time is actually wanted.
it means that the transfer to the screen must be completed within around 16.67 ms. Assuming a maximum resolution of 1920x1080 pixels, with each pixel occupying 32 bits for storage, a sustained bandwidth of at least 474.6 MB (around 3.982 Gbit) per second is required. As for the computation of the frame, it is desired but that this finishes before a screen refresh.

The scratchpad memories may be connected through a Communication Assist (CA) to the PE and to the memory controller. This CA can be used to make double buffering more transparent: pointers to the output image will point to both scratchpad memories, but as every memory operation goes through the CA, the CA can ensure the data is written to or read from the correct scratchpad memory in every frame.

All PEs must process all triangles to determine if the triangle is part of the output image that the PE must compute. Optionally, the general purpose processor will determine which triangle must be sent to which PEs for processing. However, this can result in a lot of work for the general purpose processor while on the other hand it can dispatch the input to all PEs using some sort of broadcast mechanism and continue either with the next triangle or do other computations for the application. Therefore dispatching all triangles to all PEs may be a better solution.

A FIFO-buffer can be used for sending the input data of all triangles to the PEs. As it may be the case that some PEs process the triangle and other PEs decide they do not have to do anything, the input of the FIFO-buffer should also be buffered in the PEs. This way, it is possible for some PEs to continue on the next triangle instead of stalling until all PEs are finished with the current triangle. The size of the FIFO-buffer and the input buffer in each PE is yet to be determined.

Obviously all instructions for the PEs need to be stored somewhere. Putting data and instructions in the same memory would be an option, however, this can result in access conflicts between data or instructions. Multi-ported memory can help, although the cost will be proportional to the number of ports. An alternative is for each PE to have a separate instruction memory, storing all the instructions for the entire pipeline. However, as all PEs are likely to perform the same type of work, this seems like a waste of space, as all memories store the same instructions. Perhaps it is better to use one instruction cache per PE and use one instruction memory for all PEs from which the cache can retrieve the necessary instructions.

As for the number of threads that run on one PE, this is limited to one at first. A large number
6.3. PROPOSED ARCHITECTURE

of threads on the GPU is only used for latency hiding, which is only necessary for the texture lookups if the output image is stored in the local scratchpad memories. If multiple threads are desirable, it is possible to use barrel processing like the GPU does [FSYA07]. Perhaps logic can be added to the PE to make it dual-threaded, like on the PPE, instead of using barrel processing. Connecting multiple PEs to the same CA for SIMD is another possibility, where the combination of scratchpad memories, instruction cache, CA and PEs will be called a tile.

Figure 6.2 summarizes several of the points mentioned in the previous paragraphs, although it may be the case that the actual proposed architecture differs significantly.

Figure 6.2: Ideas for the proposed architecture

6.3 Proposed architecture

The general purpose processor in the proposed architecture can be any kind of CPU core, as long as there is a C++ compiler for that CPU. A CPU core supporting connections to FIFO-buffers\[9\]

\[9\]It is not necessary to use a hardware FIFO; fast memory with two pointers (and an bit to denote whether the FIFO is empty) suffices.
is preferred over one that requires modification to add support for these kind of buffers. The
general purpose processor will use these buffers to send triangle data to the tiles for processing.

Assuming that it is desirable to store up to 100 triangles\textsuperscript{10} in the FIFO at a given moment in
time, a buffer size of 4 KB is needed, assuming an average of 10 arguments per triangle\textsuperscript{11}.

The output image will be split into multiple consecutive parts which are kept in the local scratch-

pad memory, as it keeps the latency low, especially since the pixels in the output image may be
written to more than once. Assuming that 1080p HD is the largest supported screen size, the
output image consists of 1920x1080 pixels of 4 bytes that must be stored and this is equal to
8100 KB. Given that double buffering is necessary for performance reasons, two scratchpad
memories per tile are required for the output image.

As the splitting of the output image is done per line, where the storage for each line takes
7.5 KB, at most 1080 different tiles are possible. This also means that the size of the scratchpad
memories for the output image must be multiples of 7.5 KB. It is not necessary that all tiles store
the same amount of lines. As an example, if the top and bottom part of the image is known to
have a very low amount of triangles, it is possible to store more lines of the image in the first
and the last tile compared to the other tiles in order to balance the work over the PEs. In fact, it
may be possible to do this without changing the size of the scratchpad memories if the output
image is less than 1920x1080 pixels.

For the depth buffer, which has the same amount of pixels as the output image and which stores
single precision floating point numbers, an additional scratchpad memory per tile is needed.
This scratchpad memory can be made 1 KB larger so it can also be used to cache textures. This
amount is chosen, because the texture cache working set for the NVIDIA GeForce 8 GPU is 8 KB
per 8 SPs and because it is assumed that NVIDIA has a lot of experience with optimizing texture
lookups\textsuperscript{12}. This scratchpad memory can be made another 1 KB larger for a circular buffer that
stores the triangle data retrieved from the FIFO-buffer and this amount is arbitrarily chosen to
be a quarter of the total triangles that can be stored in the FIFO-buffer.

One CA per tile is desired for connecting the PE (or PEs) to the three scratchpad memories. This

\textsuperscript{10} An arbitrary number that does not seem to be too small or too large.

\textsuperscript{11} For example, 4 arguments for homogeneous coordinates, 2 arguments for texture coordinates and 2 arguments
for other data, like pointers.

\textsuperscript{12} Using more memory for caching may be better as it can increase the hit ratio.
CA is responsible for coordinating memory accesses to the two memories containing part of the output image, for performing the texture caching, for communicating with the memory controller, and for retrieving the triangle data from the FIFO-buffer.

The instruction memory is to contain the instructions for the PE programs and should be large enough to store as many (frequently used) programs as possible. If the SPU program for the first function is an indication for the required size, then one program is about 31 KB. On the other hand, the GPU program for the first function is about 10 KB. Setting the size of the instruction memory to 256 KB allows the storage of 8 – 25 PE programs, assuming the sizes of the SPU and GPU program are more or less representative for the, yet to be decided, ISA of the PE. Note that the Irrlicht engine has about 23 different rendering functions in the software renderer, so being able to store up to 25 PE programs should be enough for now, as not every renderer function is always used in every game. Note that even though an instruction memory is used, it may be possible to translate 3D graphics API-calls runtime to PE programs, perhaps by providing a (software) library that does this.

Source listing 6.1 Reloading program into cache

```
// Initialization
...

while (condition)
{
  // Lots of statements
  ...
}
```

The instruction cache in a tile should be able to store at least one program, otherwise it may have to request different parts of the same program over and over again. For example, if the test for the condition and the loop body in Source listing 6.1 do not fit in the cache, then it may be necessary to reload the part that is not in the cache anymore after every iteration. In fact, it may be preferred to store at least two programs, so that it is possible to preload the next program while the current one is running. Given the sizes in the previous paragraph and assuming that the size of the PE program will be between the SPU and GPU programs, an instruction cache size of 41 KB per tile seems like a good choice. As for the interconnect between the instruction caches and the instruction memory, this is still an open issue, as a broadcasting mechanism

---

13The CA can send the output image to the screen via the memory controller while the PE is busy with computations.
will not work well if several PEs have to load different programs. Perhaps it is better to stream instructions to all tiles, instead of requiring the PEs to load them, but this depends on the utilization of the general purpose processor. If streaming takes up all processor time, then it will not be possible for a game to be played as it should be.

The memory controller connects the general purpose processor, the CAs of all tiles, the off-chip memory, the instruction memory and the screen. It is responsible for transferring the output image via the CAs to the screen, which requires a bandwidth of 3.982 Gbit per second as mentioned in section 6.2. Furthermore, it will process request to the off-chip memory from the general purpose processor and the CAs, for example to store or retrieve texture data and preferably it does this with high bandwidth and low latency. Finally it will load the PE programs into the instruction memory from off-chip memory when instructed by the general purpose processor, although later on it can be decided to store fixed PE programs in instruction memory as opposed to allow loading dynamic PE programs.

The off-chip memory itself is quite important, both for the data used by the general purpose processor and also to store textures. Given that a 32-bit OS and CPU can handle only 4 GB, having more than this amount makes no sense\(^4\). Using this size will require a separate address space for the instruction memory and the scratchpad memories, which makes programming a bit harder. Using a shared address space is preferred, which implies that less than 4 GB is available, and for this reason the size of off-chip memory is set to 3 GB, as this memory stores data used by the CPU and the tiles\(^5\). For example, when using textures, mipmap mapping can be used for rendering.

\(^4\) Assuming tricks like segmented addressing are not used.
\(^5\) Unlike a GPU which has its own memory separated from the host.
be used to speed up rendering elements that are far away. Mipmaps are made by halving the dimensions of a texture and other mipmaps, which is illustrated in Figure 6.3, and require only \[ \sum_{i=1}^{\infty} \left( \frac{1}{2} \times \frac{1}{2} \right)^i = \frac{1}{3} \] more space compared to the original 2D texture. Furthermore, all objects, which make up the triangles that are to be drawn each frame, will be buffered in this memory as opposed to reloading them from external I/O each time they are used. Given the fact that the graphics pipeline is the main aspect, it may be preferable to use GDDR3 or newer versions like GDDR4 and GDDR5.

Another element of the architecture is the PE. As the graphics pipeline relies heavily on floating point arithmetic, having a single precision floating point unit in the PE is actually a necessity. Furthermore, support for the floating point multiply-add operation is also added. As for integer arithmetic, this is not (yet) required for the application, so it is left out. Support for pre-emption is not necessary either, as it makes no sense to process a triangle while the drawing of the previous one is not completed yet.

The PE will support guard operations so small if-statements can be mapped to them instead of using branches. In particular, statements that are conditionally executed when a pixel is visible can be translated using guards instead of branches and the result of this is that the pipeline is not flushed when the branch is mispredicted. Predicting the visibility can be quite hard, as it depends on the triangles that are drawn and also on the order they are drawn. Using the additional guarded operations as opposed to branching is preferred in this case.

As for the register width in the PE, it is necessary to process single precision floating point numbers, so registers must be 32 bit wide. The SPU has a large amount of registers and one of the reasons is speculated to be the shared instruction and data memory, which is single-ported. In the case of the GPU, a large number of registers is needed so it can run a lot of threads. Because the number of registers for the first function can be obtained when compiling on the GPU, this number is used as a starting point for the number of registers. In the case of the optimized version on the GPU, it needs 24 registers per thread. Given that shared memory is used for storing data on the GPU, it is expected that some more registers are needed and using

---

\[ ^{16} \text{The first function uses almost 30 different textures, with the largest textures having 128x128 32 bits elements.} \]

\[ ^{17} \text{So for this function, almost 2.5 MB is used for textures.} \]

\[ ^{18} \text{For example a harddisk.} \]

\[ ^{18} \text{Although the scratchpad memory storing the depth buffer can also be used if necessary.} \]
the next power-of-two gives us 32 registers, or a register file containing 32 32-bit entries.

As can be seen from the results during the mapping on the Cell architecture, the use of SIMD in the form of vectors containing four single precision floating point number results in a speedup of about 1.3 in the first function. This is too small to justify the additional programming efforts for adding vector operations and to justify the additional costs by increasing the register width or by adding special SIMD registers to all PEs. However, if guarding on each scalar value in a vector is supported, SIMD can be used to obtain a speedup. In this case it is possible to process multiple pixels at once and mask out some pixels which are not visible using guards. As the first function on the GPU benefits from the use of 32 threads, 32 scalars will be put into one SIMD-register, resulting in 128 byte SIMD-registers. The number of scalar registers is halved to 16 and the number of SIMD-registers is set to 16 as well.

The number of PEs per tile is set to one, at least for now. Reason for this is that it is not desired for multiple PEs to compete for access to one of the (single-ported) scratchpad memories, although the option of multiple PEs per tile is kept in mind. Instead of additional PEs, it may be possible to incorporate Special Function Units (SFUs) which implement functionality of the graphics pipeline. If there is one SFU for every function in the graphics pipeline, it may be possible to remove the instruction memory and instruction caches. A block diagram of the proposed architecture, with no SFUs for now, can be seen in Figure 6.4.

As for the number of tiles, this is a tradeoff between performance and size. When there are a lot of tiles, and thus a lot of PEs, each tile has less work to process compared to when the amount of tiles is low. On the other hand, more tiles require more memories, more PEs and additional CAS. Table 6.1 contains a summary of the sizes of the memories for several numbers of tiles, assuming one PE per tile and assuming a 1920x1080 resolution, where the division of the output image is done per line.

If the Cell is taken as a reference, the number of tiles can be set to eight. On the other hand, if the GPU is taken as a reference, 96 or 128 tiles can be used instead. To decide the number of tiles, a very rough estimation of the minimum clock speed of a PE is used.

Assume that the application must be able to draw 7 million triangles per second, which is over

---

19 The first function also benefits from 64 threads, but having 256 byte registers seems to be a bit of overkill.
### 6.3. PROPOSED ARCHITECTURE

#### Table 6.1: Number of tiles compared to size of memories

<table>
<thead>
<tr>
<th>Number of Tiles</th>
<th>Size of LMEM0 and LMEM1 per PE (KB)</th>
<th>Size of LMEM2 per PE (KB)</th>
<th>Total size of instruction cache (KB)</th>
<th>Total size of local memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8100</td>
<td>8102</td>
<td>41</td>
<td>24302</td>
</tr>
<tr>
<td>2</td>
<td>4050</td>
<td>4052</td>
<td>82</td>
<td>24304</td>
</tr>
<tr>
<td>3</td>
<td>2700</td>
<td>2702</td>
<td>123</td>
<td>24306</td>
</tr>
<tr>
<td>4</td>
<td>2025</td>
<td>2027</td>
<td>164</td>
<td>24308</td>
</tr>
<tr>
<td>5</td>
<td>1620</td>
<td>1622</td>
<td>205</td>
<td>24310</td>
</tr>
<tr>
<td>6</td>
<td>1350</td>
<td>1350</td>
<td>246</td>
<td>24312</td>
</tr>
<tr>
<td>7</td>
<td>1162.5</td>
<td>1164.5</td>
<td>287</td>
<td>24426.5</td>
</tr>
<tr>
<td>8</td>
<td>1012.5</td>
<td>1014.5</td>
<td>328</td>
<td>24416</td>
</tr>
<tr>
<td>9</td>
<td>900</td>
<td>902</td>
<td>369</td>
<td>24318</td>
</tr>
<tr>
<td>10</td>
<td>810</td>
<td>812</td>
<td>410</td>
<td>24320</td>
</tr>
<tr>
<td>16</td>
<td>510</td>
<td>512</td>
<td>656</td>
<td>24512</td>
</tr>
<tr>
<td>32</td>
<td>255</td>
<td>257</td>
<td>1312</td>
<td>24544</td>
</tr>
<tr>
<td>64</td>
<td>127.5</td>
<td>129.5</td>
<td>2624</td>
<td>24608</td>
</tr>
<tr>
<td>96</td>
<td>90</td>
<td>92</td>
<td>3936</td>
<td>24612</td>
</tr>
<tr>
<td>128</td>
<td>67.5</td>
<td>69.5</td>
<td>5248</td>
<td>26176</td>
</tr>
<tr>
<td>160</td>
<td>52.5</td>
<td>54.5</td>
<td>6560</td>
<td>25520</td>
</tr>
<tr>
<td>192</td>
<td>45</td>
<td>47</td>
<td>7872</td>
<td>26304</td>
</tr>
<tr>
<td>224</td>
<td>37.5</td>
<td>39.5</td>
<td>9184</td>
<td>25648</td>
</tr>
<tr>
<td>256</td>
<td>37.5</td>
<td>39.5</td>
<td>10496</td>
<td>29312</td>
</tr>
<tr>
<td>288</td>
<td>30</td>
<td>32</td>
<td>11808</td>
<td>26496</td>
</tr>
<tr>
<td>320</td>
<td>30</td>
<td>32</td>
<td>13120</td>
<td>29440</td>
</tr>
<tr>
<td>352</td>
<td>30</td>
<td>32</td>
<td>14432</td>
<td>32384</td>
</tr>
<tr>
<td>384</td>
<td>22.5</td>
<td>24.5</td>
<td>15744</td>
<td>26688</td>
</tr>
<tr>
<td>416</td>
<td>22.5</td>
<td>24.5</td>
<td>17056</td>
<td>28912</td>
</tr>
<tr>
<td>448</td>
<td>22.5</td>
<td>24.5</td>
<td>18368</td>
<td>31136</td>
</tr>
<tr>
<td>480</td>
<td>22.5</td>
<td>24.5</td>
<td>19680</td>
<td>33360</td>
</tr>
<tr>
<td>512</td>
<td>22.5</td>
<td>24.5</td>
<td>20992</td>
<td>35584</td>
</tr>
<tr>
<td>768</td>
<td>15</td>
<td>17</td>
<td>31488</td>
<td>36996</td>
</tr>
<tr>
<td>1024</td>
<td>15</td>
<td>17</td>
<td>41984</td>
<td>48128</td>
</tr>
<tr>
<td>1080</td>
<td>7.5</td>
<td>7.5</td>
<td>44280</td>
<td>26400</td>
</tr>
</tbody>
</table>
10 times more than the demo application is able to draw when using hardware acceleration\(^{20}\) and assume that the distribution of triangles over the tiles is roughly the same. It is known that the SPU and GPU programs for the first function are about 31 KB and 10 KB. If the size of the SPU program is taken as a reference and because an instruction is four bytes [IBM07d], the number of instructions is around eight thousand per program. Assuming that the PE is able to issue one single precision floating point instruction in one cycle\(^{21}\) and assuming that the number of instructions issued at runtime is 10 times larger\(^{22}\), there are 80 thousand cycles per triangle.

\(^{20}\)The application can draw around 650 thousand triangles per second, but this is at a 800x600 resolution instead of 1920x1080. This was tested on the PC containing the GPU.

\(^{21}\)Like the GPU.

\(^{22}\)Assuming no stalls on memory operations, but including the branches.
6.3. PROPOSED ARCHITECTURE

However, the SIMD-registers are eight times wider than on the Cell, so the number of cycles will be closer to 10 thousand. Using this very rough estimate, a clock speed of 8.75 GHz per PE is needed, when the number of tiles is equal to eight. For 96 tiles the clock speed can be set to a little less than 730 MHz, whereas for 128 tiles less than 550 MHz suffices. As it may be desired to keep the number of tiles low, 64 tiles is also an option, for which a clock speed of less than 1100 MHz per PE would still be enough. For now, the number of tiles is set to 64, but keep in mind that this is based on a very loose estimation.

As for the bandwidth between the different objects in the proposed architecture, the numbers seen in Figure 6.4 are used. The connection to the screen needs a bandwidth of at least 3.982 Gbit per second, as was mentioned earlier. As the output image has to be obtained from the local memories, this kind of bandwidth is also needed between the the CA and local memories and it is rounded to 4 Gbit per second. Since the memory must be accessible while transferring data to the screen, the bandwidth between the CA and the memory controller is doubled and the bandwidth between the memory controller and the off-chip memory is set to multiples of 4 Gbit per second. To be symmetric, the bandwidth between the CPU and the memory controller is also set to 4 Gbit per second and the same holds for the connection between the PE and the CA. The FIFO-buffer needs to stream seven million triangles per second, which requires a bandwidth of 2.24 Gbit per second, assuming that the data for each triangle is about 40 bytes. The connection between instruction memory and the memory controller is limited to 2 Mbit per second, which means that the entire instruction memory can be filled in about one second. For the bandwidth between the instruction memory and the instruction cache, it is assumed that there are 12 different programs, which are 20.5 KB on average and which have to be loaded in \( \frac{1}{5} \) of the time available for a frame. This means that the bandwidth must be about 605 Mbit per second between the instruction memory and the instruction cache. Finally, the bandwidth between the instruction cache and the PE is doubled compared to the bandwidth between the instruction memory and instruction cache to obtain 1.21 Gbit per second.

With this the main outline for the new architecture has been decided, based on the evaluation of the Cell and the GPU and it is assumed that this architecture will have high performance for at least the rasterization stage of the 3D graphics pipeline.
Chapter 7

Conclusion and Future Work

This chapter contains the conclusions of the work presented in this report, based on the evaluation of the architectures by mapping the graphics pipeline and based on the proposal of the new architecture. Besides the conclusion, suggestions for future work are also given.

7.1 Conclusion

The main part of this work consisted of getting familiar with and mapping parts of the 3D graphics pipeline onto the Cell Broadband Engine and the GeForce 8 Series Graphics Processing Unit in order to gather ideas for possible improvements. Given the fact that there are a lot of optimizations possible, and some of them are actually necessary, it should be no surprise that this part took the most amount of time. Debugging these programs is also time-consuming, as this consisted mainly of looking carefully in the source code for mistakes.

All known optimizations are applied in some form for both the Cell and the GPU, as detailed in subsection 5.2.1 and subsection 5.3.1. It is believed that all optimizations have been applied correctly to the mapped programs. One obvious optimization that has not been applied due to the lack of time is the mapping of the entire graphics pipeline as opposed to mapping single

\footnote{An example of a bug: everyone “knows” that \texttt{for(int x = a; x < b; ++x) c += d;} can be rewritten to \texttt{c += d * (b - a);}. However, this will give wrong results if \texttt{a} is larger than \texttt{b}.}
functions. However, it is believed that these functions are representative for the (rasterization stage of the) graphics pipeline, as they are the most time-consuming ones.

It is very important that the mappings of the programs are optimized for both the Cell and the GPU, as the ideas for the proposal are taken from the evaluation of these mappings. From the experience gained during this project, it can be said that mapping programs onto the Cell and the GPU is relatively easy. However, without any optimizations, these programs will usually run with (very) low performance and can actually be slower than the original program on some reference CPU. On the other hand, it will be an advantage if it is easy to write efficient programs for the proposed architecture. Unfortunately, there was not enough time to decide on the programming methodology for the new architecture.

All components for the proposed architecture have been described, although some parts are described with more details than other parts. The purpose for the components has been described, together with the rationale for using the components, whenever this was deemed necessary. Furthermore, the bandwidth between all components has been set, based on certain assumptions, like the (maximum) resolution of the output image.

A very rough estimate has been made for the number of tiles and the clock speed for each PE, assuming that a tenfold performance improvement compared to current GPUs is desired. There was not enough time to build a model in order to verify these numbers, so it is assumed that this architecture will perform better than current GPUs.

Finally, it should be noted that this architecture is based on the rasterization stage. The vertex processing stage is ignored entirely\textsuperscript{2} even though it is part of the graphics pipeline. For example, in the vertex processing stage it is not yet known onto which part of the screen a triangle will be projected, but in our architecture there are no direct communication links between different tiles for sending triangle data, which may be suboptimal. Also, the entire evaluation is based on a software implementation, but it is believed that this is less of a problem, as this implementation contains a quite complete set of the most frequently used functionality in the graphics pipeline.

In short, an architecture has been proposed which is believed to obtain high performance when

\textsuperscript{2}It must be processed by the CPU, while it may be possible to offload this part to (some of) the tiles as well.
used for the graphics pipeline even though the architecture itself may be less than optimal for the vertex processing part of the 3D graphics pipeline.

7.2 Future Work

As always, there is room for improvements, so suggestions for future work are given in this section.

A (simple) model should definitely be made to verify whether the rough estimation of the number of tiles and the clock speed of the PEs will yield the desired performance improvements. This model may then be simulated in order to perform the verification.

The architecture should be described in more detail, open issues should be addressed and potential ambiguity should be resolved, so that it is possible to make an implementation of the architecture in the future, if this is desired. This will require a feasibility study, for example to check whether the bandwidths given in the previous chapter are realistic numbers. The model and simulation may be used as part of this feasibility study.

The vertex processing stage should be taken into account as well. As mentioned in the previous section, direct communication links between different tiles (CAS) may be necessary. An alternative is to have separate tiles with PEs or SFUs for processing the vertices, similar to separate vertex and pixel shaders in previous generation GPUs, which can be up to 7% slower compared to the unified shader model [MGR05]. Another alternative is to add a second FIFO-buffer, which contains the vertex data that is to be processed, and allowing the tiles instead than the CPU to put the triangle data in the first FIFO-buffer.

The way the texture cache works may also be taken into account. Lookups in a 2D texture at location \((x, y)\) have a high chance to be followed by lookups at locations \((x + i, y + j)\) for \(-1 \leq i \leq 1\) and \(-1 \leq j \leq 1\). This property may be exploited to obtain higher performance.

The way to program the architecture has not been decided yet. As mentioned in the conclusion, it may be preferable to make programming high performance application easy, as opposed to using all kind of tricks to obtain optimal performance.

\[^3\]In particular games, but other applications that use rendering may also benefit from this architecture.
Finally, it may be desired to compare this proposed architecture to other (new) architectures. An example architecture that can be compared is the GPU with 800(!) Stream Processors introduced by AMD recently, late June 2008. Or perhaps it is more desirable to compare to architectures from research projects as opposed to commercial offerings.
Appendix A

Sample PPE Source Code

#include <stdio.h>
#include <libspe.h> // Library for interfacing with the SPE

/* SPU program:
   embedded into the PPE executable,
   contained in a dynamically loaded library using dlopen()/dlsym(),
   or loaded using spe_open_image() */
extern spe_program_handle_t spu_program;

/* Number of SPEs that should be started,
   should not exceed the total available */
#define NUMBER_OF_SPES 2

int main()
{
    speid_t speThread[NUMBER_OF_SPES];

    // Start SPEs
    for(int i = 0; i < NUMBER_OF_SPES; ++i)
    {
        speThread[i] = spe_create_thread(0, &spu_program, NULL, NULL, -1, 0);
        if(speThread[i] == 0)
        {
            fprintf(stderr, "Cannot create SPE thread\n");
            return -1;
        }
    }
}
// Optionally do other things on PPE

// Wait for the SPEs to finish
for(int i = 0; i < NUMBER_OF_SPES; ++i)
{
    int status;
    spe_wait(speThread[i], &status, 0);
}

return 0;
Appendix B

Sample CUDA Code

dim3 blocks(16, 2, 1); // Number of blocks can be specified in three dimensions
dim3 threads(32, 4, 2); // Same for number of threads

int system_mem[1024]; // Resides in system memory

__device__ int gpu_mem[1024]; // Resides in GPU memory

__constant__ int gpu_constant[1024]; // Resides in GPU constant memory

// Distinct copies in each group of eight stream processors
__shared__ int gpu_shared[1024];

// Function existing on GPU only
__device__ void gpu_device()
{
    // gpu_device(); // No recursion possible
}

// Entry function for GPU
__global__ void gpu()
{

    int threadx = threadIdx.x; // Retrieve part of thread index
    int blockz = blockIdx.z; // Retrieve part of block index

    // Call GPU function
gpu_device();
}
int main()
{

    // Call GPU function with number of blocks and threads
    gpu<<<blocks, threads>>>() ;

    // Optionally do other things

    // Wait until the end of the GPU function
    cudaThreadSynchronize();

    return 0;
}
Appendix C

Compiling Irrlicht using GCC

The Irrlicht download provides a makefile that can be used for compiling it under Linux as a static library, which after compiling be embedded into programs using the Irrlicht engine. This makefile uses GCC, but does not enable any optimizations at all, which can result in programs running slower than necessary.

Source listing C.1 Violation of the strict aliasing rule

```c
int swapIncorrect(int arg)
{
    short* p = (short*) &arg;
    short p0 = p[0];
    short p1 = p[1];
    p[0] = p1;
    p[1] = p0;
    return arg;
}
```

Enabling optimizations by adding the “-O3" switch results in incorrectly working applications however. This is caused by violations of the so-called strict aliasing rule, as can be seen from the warning${}^{1}$. An example that gives this warning can be seen in Source listing C.1.

The strict aliasing rule states that a compiler can assume that pointers to objects of incompatible types will never refer to the same memory location${}^{2}$ or in other words, they cannot alias${}^{3}$. Because of this rule, the compiler can optimize certain kinds of programs, like the

---

1“warning: dereferencing type-punned pointer will break strict-aliasing rules”.
2char* is an exception.
APPENDIX C. COMPILING IRRLIGHT USING GCC

Source listing C.2 Advantage when using strict aliasing

```c
void add(int* values, short* constants, int n)
{
    int i;
    for(int i = 0; i < n; ++i)
        values[i] += constants[i&3];
}
```

one given in [Source listing C.2][1] In this program the compiler can assume that modifying the values array will not result into modifications of the constants array; it is only necessary to load the values in the constants array once.

Source listing C.3 Working around the strict aliasing rule

```c
typedef union
{
    int a;
    short b[2];
} intshort;

int swapCorrect(int arg)
{
    intshort tmp;
    tmp.a = arg;
    short p0 = tmp.b[0];
    short p1 = tmp.b[1];
    tmp.b[0] = p1;
    tmp.b[1] = p0;
    return tmp.a;
}
```

Using a C-union, as shown in [Source listing C.3][2] it is possible to work around this problem. This is used as a solution in the Irrlicht engine, after which it is possible to enable optimizations when compiling Irrlicht under Linux using GCC.

Another problem that was encountered, was the fact that the colours of the demo application were incorrect when running on the PS3[3] using the software renderer. This was traced to a bug in big endian mode and after fixing this, the colours came out correctly.

---

1Admittedly, this sample is a bit contrived as the constants array does not contain ints but shorts.
2An X-server was already installed on the PS3, so running the demo was as easy as plugging in a mouse, starting the X-server and starting the application.
Appendix D

Subword parallelism for the YUV to RGB program

The vertical scaling can potentially be sped up by computing four values at once as can be seen in Figure D.1. The horizontal scaling can also potentially be sped up, see Figure D.2. The
shifting and extracting scalar values do require some computation, so it may be the case that attempting to speed this up will result in a slowdown. These two kinds of optimizations were not implemented in the preparation.

![Diagram of subword parallelism for computing RGB](image)

**Figure D.3**: Subword parallelism for computing RGB

One of the optimizations that is implemented, is the one shown in Figure D.3, where the three RGB values are computed using vector operations.
Appendix E

Extra information for mapping

E.1 Software Managed Cache API

The Cell SDK contains an “SPU Software Managed Cache Library” which allows a programmer to designate part of the LS as an explicitly managed cache. To use this API the programmer has to define parameters including the data type, the size of a cache line, the number of sets that can be in the cache, whether the cache is 4-way or 1-way and whether the cache is readonly. Furthermore the name must be defined and it is possible to have the library collect statistics about the cache.

After defining the parameters, the cache-api.h file must be included to use the cache. It is possible to use multiple software caches by redefining the parameters and including the same header file again.

With the software cache set up, the cache_rd, cache_wr and cache_flush function calls can be used to read from, write to or flush the cache.

Finally, the programmer must be aware that the cache on one SPE is not coherent with the cache from another SPE. Furthermore, after using the above mentioned function calls, the current DMA tag, which is for example used to wait for any or all DMA transfers with a certain tag, may be changed and it may be necessary to explicitly reset the tag after using the cache.

1The latter two function calls only make sense if the cache is not read-only.
E.2 Comparison between the PPE and the Intel Q6600 CPU

The number of triangles per second on the PPE is about 3.5 times less when compared to the CPU. This large difference resulted in this simple comparison between these two processors to see whether the number of triangles per second represent the performance of the processors well enough.

Source listing E.1 Increasing all elements in an array

```c
#define SIZES 100000000
int data[SIZES];
unsigned long long i;

for(i = 0ull; i < SIZES; ++i)
{
    ++data[i];
}
```

The C++ program in Source listing E.1, which increases all 100 million elements in an array, has been compiled and run on both the PPE and the CPU with optimizations on and off and with the loop body unrolled ten times. The times taken for this program can be found in Table E.1.

From the timings it can be seen that branches are quite expensive on the PPE and that the Intel Q6600 CPU has a higher performance, despite having a lower clock frequency.

Table E.1: Timings of program increasing all elements in an array

<table>
<thead>
<tr>
<th></th>
<th>Not unrolled</th>
<th>Not unrolled</th>
<th>Unrolled 10×</th>
<th>Unrolled 10×</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Not optimized</td>
<td>Optimized</td>
<td>Not optimized</td>
<td>Optimizations</td>
</tr>
<tr>
<td>PPE</td>
<td>5.85 s</td>
<td>3.11 s</td>
<td>4.28 s</td>
<td>2.76 s</td>
</tr>
<tr>
<td>Q6600</td>
<td>1.20 s</td>
<td>0.36 s</td>
<td>1.06 s</td>
<td>0.36 s</td>
</tr>
</tbody>
</table>

Table E.2: Timings of program increasing one variable

<table>
<thead>
<tr>
<th></th>
<th>Not unrolled</th>
<th>Not unrolled</th>
<th>Unrolled 10×</th>
<th>Unrolled 10×</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Not optimized</td>
<td>Optimized</td>
<td>Not optimized</td>
<td>Optimizations</td>
</tr>
<tr>
<td>PPE</td>
<td>24.89 s</td>
<td>0.00 s</td>
<td>18.94 s</td>
<td>0.00 s</td>
</tr>
<tr>
<td>Q6600</td>
<td>3.44 s</td>
<td>1.28 s</td>
<td>2.56 s</td>
<td>0.17 s</td>
</tr>
</tbody>
</table>

The same experiment has been performed, except that instead of increasing all 100 million elements in an array, one variable is increased a 1000 million times. The times taken for

---

*This can be calculated from the numbers in section 5.2 and section 5.3.*
this program can be found in Table E.2 and here again it can be seen that branches are quite expensive. Furthermore, the compiler plays an essential role when performing optimizations: increasing a variable 1000 million times can be substituted by a single precomputed addition.

From both experiments, it can be safely concluded that the PPE lacks in performance, even when doing simple tasks. This is to be expected as the computationally intensive tasks are meant to run on the SPEs instead.

In the case of more complex programs on the PPE, so called microcoded instructions can also give a performance penalty [IBM07], as these must be decoded into multiple simple PowerPC instructions and require at least 11 cycles. Using a general compiler for the PowerPC ISA which has no knowledge of these instructions should be avoided for this reason; instead a compiler which prefers non-microcoded instructions is recommended.

### E.3 Coalesced memory access

To maximize the total bandwidth when accessing global memory on the GPU, so called coalesced memory accesses should be used [NVIO7] [Har07]. This means that thread 0 in a half-warp should access memory aligned to $16 \times \text{sizeof(type)}$, whereas thread $y$ in the same half-warp should access the same memory offset by $y$ as shown in Figure E.1. It is not necessary that all threads in the same half-warp access memory which is also shown in Figure E.1.

![Figure E.1: Coalesced memory accesses](image)

For future GPUs it may be necessary that coalescing only works on entire warps, as opposed to half-warps in current GPUs.
Coalescing works for 32-bit, 64-bit and even 128-bit memory accesses, although the bandwidth for the 32-bit accesses is highest and for 128-bit accesses is lowest. However, coalesced accesses still have higher bandwidth compared to non-coalesced accesses, of which examples can be seen in Figure E.2.

E.4 Reduction operation

A reduction can be used when the operation in question is both associative and commutative⁴ and can best be explained using a picture. Figure E.3 shows how a reduction is performed, assuming a power of two as the number of threads and as the size of an array. In this case, all threads $x_0$, where $0 \leq x_0 < 16$, will first compute offset $x_0$ and $x_0 + 16$ in the array and store the result in location $x_0$. After that, all threads $x_1$, where $0 \leq x_1 < 8$, will compute $x_1$ and $x_1 + 8$ in the array. This continues until thread 0 computes the elements 0 and 1 in the array and stores the result in location 0 in the array.

⁴For example addition or multiplication.
Figure E.3: Reduction operation
Bibliography


S_TACT=105A0X16&S_CMP=LP


[GCC] *GCC, the GNU Compiler Collection*, http://www.gnu.org/software/gcc/

[GTE] *eVGA Production Specification Sheet – e-GeForce 8800 GT*,


[RAM] RAMP – Research Accelerator for Multiple Processors,
http://ramp.eecs.berkeley.edu/


[See08] Peter Seebach, Linux development on the Playstation 3, Part 1, 2008,


[Som07] Rys Sommefeldt, AMD R600 Architecture and GPU Analysis, 2007,
http://www.beyond3d.com/content/reviews/16

[TRI] The TRIPS Project – Tera-op Reliable Intelligently adaptive Processing System,
http://www.cs.utexas.edu/~trips/


[XLC] IBM Software – C and C++ compilers,