UNIVERSITY OF OSLO
Department of Informatics

Efficient Linear Algebra on Heterogeneous Processors

Id

Master's thesis

Trygve Fladby

2nd May 2007

Path: architecture.tex
Last Changed Rev: 150
Last Changed Date: 2007-04-11 23:05:14 +0200 (Wed, 11 Apr 2007)
Path: backmatter.tex
Last Changed Rev: 78
Last Changed Date: 2007-02-07 17:45:15 +0100 (Wed, 07 Feb 2007)
Path: cg.tex
Last Changed Rev: 129
Last Changed Date: 2007-03-30 09:24:29 +0200 (Fri, 30 Mar 2007)
Path: frontmatter.tex
Last Changed Rev: 78
Last Changed Date: 2007-02-07 17:45:15 +0100 (Wed, 07 Feb 2007)
Path: future.tex
Last Changed Rev: 136
Last Changed Date: 2007-04-02 22:09:30 +0200 (Mon, 02 Apr 2007)
Path: introduction.tex
Last Changed Rev: 139
Last Changed Date: 2007-04-04 18:12:27 +0200 (Wed, 04 Apr 2007)
Path: lu.tex
Last Changed Rev: 147
Last Changed Date: 2007-04-11 20:34:23 +0200 (Wed, 11 Apr 2007)
Path: maintext.tex
Last Changed Rev: 150
Last Changed Date: 2007-04-11 23:05:14 +0200 (Wed, 11 Apr 2007)
Path: parallel.tex
Last Changed Rev: 140
Last Changed Date: 2007-04-04 21:40:11 +0200 (Wed, 04 Apr 2007)
Path: thesis.tex
Last Changed Rev: 135
Last Changed Date: 2007-04-01 21:12:07 +0200 (Sun, 01 Apr 2007)

Abstract

This thesis use the extreme powers of the GPU for linear algebra. Selected linear algebra algorithms, more specifically the LU and the conjugate gradient algorithm for solving linear systems, has been ported to execute its main computational load on the graphics processing unit available on most computers. The main contributions in the thesis is more efficient pivoting in the LU-algorithm, where a minimum of data is copied, and gathering of the inner products for simultainous readback and reduction on the GPU.

Prefeace

This thesis is a part of the GPGPU project [**?**] at Sintef ICT that aims to use the GPU as a high performance computational resource. This is individual work, but multiple topics from the work have been discussed with and are influenced by the thoughts of other students in the group. The algorithms provided in this thesis have been tested in WindowsXP on a computer with an Intel Pentium IV 3.00GHz processor, 2GB ram, and a NVIDIA 7800GT graphics card.

Attached paper: At the end of the thesis a white-paper is attached. This is the result of the co-operation between Brodtkorb, Sætra and me on applying the LU-algorithm to a cluster of computers. My special contributions include design of the local LU-algorithm, that is basically just a reimplementation of the LU-algorithm deduced in this thesis, and the implementation of the local pivot and eliminate procedure. It is difficult to take credit for details in the paper, because it has been rewritten and changed by all of us many times, but at least the pivot and eliminate sections are originally written by me.

Acknowledgements: First of all I would like to thank my two supervisors Knut-Andreas Lie and Trond Runar Hagen for providing assistance and feedback throughout the period of the last one and a halve years at Sintef. Secondly I would like to thank André Rigland Brodtkorb that have worked with almost the same topics as me, but also focused on providing an interface to MATLAB, for many meaningful and insightful discussions and providing me with a setup to get started with LaTeX, and Martin Lilleng Sætra that works with utilizing a cluster of GPUs, for working with Brodtkorb and me in a joint effort of porting the LU-algorithm to a cluster of GPUs. The result of this co-operation is the white-paper attached to the back of this thesis. I would also like to thank my other fellow master students at Sintef; Hanne Moen and Lars Moastuen for discussions and a friendly social environment. It is also important for me to thank Tor

Dokken for some insightful thoughts, Jon Mikkelsen Hjelmervik for the idea of the top500 project, Sintef ICT for providing us with offices with a view and the department of Informatics at the University of Oslo for providing us with computers and large screens, and everyone else that has played a role in the process of making this what it is. Finally, I would like to thank my family, especially my mother and father for being extra patient with me over the last months.

Trygve Fladby
April 2007

Contents

Chapter 1

Introduction

Graphics processors have traditionally had a very predetermined set of available commands, controlled by graphics APIs like OpenGL and DirectX. During recent years the graphics chips have rapidly evolved into fast and programmable stream processors to an affordable price. Simultaneously the rise in "gigahertz" has started to hit the wall on conventional processors and we can see an emerging multi-core trend. Executives from Intel have announced they are five to eight years away from producing 80-core chips and Intel is currently experimenting with new core designs [1]. In this thesis the GPU is utilized as a multi-core co-processor to allow us to experiment with stream-programming and the process of splitting tasks into tiny data-parallel pieces that are assigned to multiple cores. More specifically the field of research in this thesis is numerical linear algebra applied on heterogeneous multi-core processors. In this thesis these processors are the CPU and the GPU, but the basic ideas should fit with other stream processors as well.
There is already much work done in the field of General Purpose Computing on the GPU, but there is still much to do. Researchers have been able to demonstrate a set of algorithms that proves that the GPU can be suited as a computational resource. In this thesis I have investigated existing functionality related to linear algebra on the GPU, and implemented some algorithms, for best possible performance.

1.1 Organization of the thesis

In the thesis I will first go through the basics of parallel programming, before I follow-up with categorizing the graphics processor into the class

[1] http://informationweek.com/news/showArticle.jhtml?articleID=196901935

of Heterogeneous processors. Then I continue with more details related to the GPU, before I end with what I have done; implementation of the conjugate gradient method and the LU-algorithm on the GPU.

Chapter 2

Parallel programming

Traditional software is mainly written in a serial fashion targeting a single CPU. Unfortunately, the speed of computations in serial programs has to date more or less been limited by the doubling of the number of transistors every 18 months (Moores Law). This issue has been overcome on larger computations by splitting up the task and adapting it to simultaneously execute on multiple CPUs. Lately, we have also seen a multi-core revolution on standard computers. The catch is that programmers must leave the idea of traditional serial programs and adjust to a parallel programming model, choose parallel algorithms or redesign serial algorithms, in order to take advantage of the extra speed. This chapter will give an overview of common issues, concepts and terminology related to parallel programming. Some of the background material on general terminology is based on Wikipedia article [Wik07c] and article [Wik07b].

2.1 Types of parallelism

There are in general three different ways a problem can be parallelized. Task-, instruction- and data-based parallelism. The way parallelization is implemented in a program can typically be divided in the two categories, implicit and explicit parallelism. Implicit parallelism is that the system, the compiler, or some other control mechanism partitions the problem and sends tasks to processors automatically, as opposed to explicit parallelism where this is for the programmer to determine. Below there will be a closer inspection of the three different ways of parallelization.

Task parallelism: Task parallelism is to identify sections of code that can be executed independently on multiple cores or CPUs. The main problem

might be that there are often a limited number of independent tasks available that can be performed simultaneously, so it might be hard to scale the program beyond a few cores. Another issue is balancing the load, so that processors do not have to idle.

Instruction parallelism: Instruction based parallelism is widely used on computers today. The compiler groups several simple instructions together and tries to optimize the program to execute the instructions simultaneously.

Data parallelism: Data parallelism is when array or stream elements are distributed to each processor, so that each processor owns a portion of the stream and executes instructions on a sub-stream. This model scales extremely well when there is little dependency between elements in a stream. See in Listing 2.1 how this would be expressed as a loop in a sequential program.

Listing 2.1: Inner loop data parallelism

```
1 float a[n], b[b], c[n];
  for(int i = 0; i < n; i++)
    c[i] = a[i]+b[i]
```

Algorithms that for efficiency reasons can be vectorized for use with Matlab or Python map very well to this level of parallelization and because all sub-arrays usually are about the same size, this type of parallelization will balance the load well to multiple processors.

2.2 Communication

Like serial, parallel algorithms need to be optimized for memory usage and cpu- time, but unlike serial algorithms it is also necessary to optimize for communication between processors. There are four patterns in communication that tend to show up during the design phase in many algorithms. These patterns will be outlined, but first there will be an overview of the two ways parallel processors can communicate.

Shared memory: Multiple processors share a global address space. Changes to one memory location affected by one processor will immediately be available to all other processors in the domain. Because there is a risk that

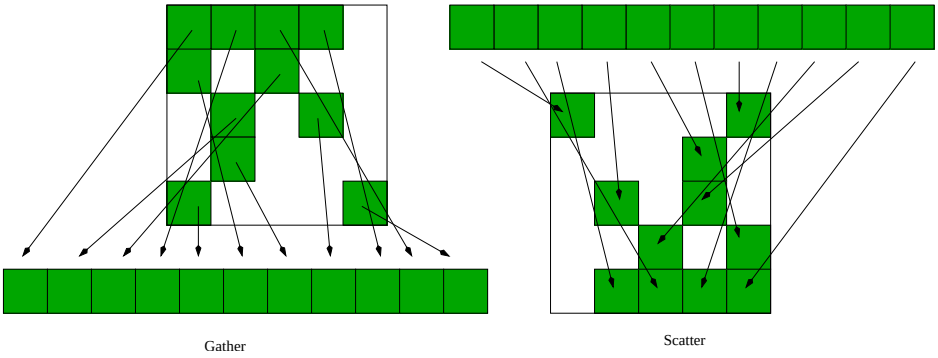


Figure 2.1: Gather and Scatter

different threads will read and write, or two threads will write simultaneously to the same location, it is necessary to synchronize the threads to avoid race conditions and errors in the result.

Message passing: Processors have their own memory and communication is done by passing messages, which may incur some added time, latency, to initiate communication. Often it is desirable to group transfers between processors to reduce the significance of latency at the cost of bandwidth.

Broadcast and Reduction: Broadcast is to distribute relatively few values to all processors available and is limited only by propagation of values in the processor grid. The opposite is called, reduction which is to reduce a set of values to fewer, for example to find the highest, lowest, or the sum of the elements in an array. Reduction should be relatively fast and parallelizable until there are fewer values than processors left, since it always is possible to read values that reside in cache or local memory.

Gather and Scatter: The gather operation takes a list of addresses and writes the values in these addresses into an ordered array. Scatter takes an array and a list of pointers to scattered locations, and scatters the values to these locations. These operations are in other words opposite of each other. See Figure~**??**. Unfortunately, these operations may lead to cache misses and increased communication, if the data structure is not carefully planned.

Synchronous and Asynchronous Communication: There are two more important terms related to communication. The first is synchronous communication which requires the processors to wait for each other to finish with current jobs before data can be transferred. The other is asynchronous communication which allows tasks to transfer data independently from one another.

2.3 Parallel architectures

After stating the differences between data-, instruction- and task-based parallelism and the differences of communication patterns, we discuss the classification of parallel architectures.

Flynn's Taxonomy: Flynn's Taxonomy is one of the widest used classifications of parallel computers. The most interesting categories today are SIMD, Single Instruction Multiple Data and MIMD, Multiple Instruction Multiple Data. Most modern CPUs are able to execute multiple instructions on multiple data simultaneously, MIMD. SIMD is a vector processor which performs the same instruction on hundreds of elements in a stream simultaneously. In addition to the GPU which is SIMD, modern CPUs also have SIMD in their instruction set (Intel SSE) and the IBM Cell processor has multiple vector processors.

Chapter 3

Heterogeneous processors

If you look at it, by the time you put dozens of cores on a chip, they won't be the same kind that you can put three or four on a chip today

–MANNY VARA
Technology strategist with
Intel's RD labs

There are two types of architectures for numerical computation in parallel environments. There is the more traditional homogenous multi-core style, where the code is set to run on equal parallel processors or cores, and the heterogeneous computing environment where there are several different architectures working together. Obviously it is easier to port existing codes to homogenous computers than heterogeneous because all cores can run the same code, but when a new homogenous computer is installed, there may be some tweaking to get the code to run optimally with respect to maximum cache re-usage and special optimized instructions. For instance a lot of effort has been put into getting ATLAS to Auto Tune itself for speed and getting compilers to translate code to the fastest instruction set available.

Heterogeneous multi-core computing is the utilization of several architectures, were some are designed for a special purpose. The algorithms are reimplemented to make the best usage of the special purpose architectures and handle constraints in the architecture. The advantage of such architectures is that they are designed for speed in one special application

and not for usage in everything between desktop applications and high-performance computing. The IBM Cell Broadband Engine Architecture is a heterogeneous multi-core architecture on a chip that has gotten a lot of attention lately as the processor in Playstation 3. This processor has one main processor (PPE), and eight Synergistic Processing Units (SPU) made for acceleration of high-performance applications. AMD has also outlined plans for "Accelerating Processing Units", that is, multi-core chips that include any mix of dedicated processors [1]. Natural choices for such processors could be a GPU, media accelerators, something similar to, or licensed, ClearSpeed [2] [3] acceleration technology for high-end computational systems, and Ageia [4]PhysX physics accelerator. All are parts that are delivered on extension cards today and together with a CPU provide a heterogeneous programming environment. One advantage of such an integral system will be low cost of communication between the processors.

Even though on-chip heterogeneous processors yet are less common, heterogeneous environments have been common for years. The most computationally demanding consumer applications are 3D rendering and consequently about all commodity computers have a dedicated graphics processor unit (GPU). Because of increased flexibility in the architecture in recent years, General Purpose computations on the GPU (GPGPU) has become an important part of the research as a data-parallel processor in heterogeneous multi-core systems. Therefore the work in this report is also based on this system.

[PPM06] presents parallel matrix multiplication in a CPU and GPU heterogeneous environment. One challenge they point out is to balance the load of multiplication between the CPU cores and the GPU, and they consider to research further on developing an automatic performance tuning library. In chapter 5 of this report there is also done some work on utilizing both the CPU and GPU simultaneously. However, load balancing will not be an issue, since different parts are task-parallelized between the CPU and the GPU in a way that favors the most effective features in both architectures.

The benefits from utilizing heterogeneous processors are reflected in

[1] http://techreport.com/onearticle.x/11438
[2] http://www.clearspeed.com/
[3] http://www.reghardware.co.uk/2006/03/15/amd_clearspeed_opteron_maths_co-pro/
[4] http://www.ageia.com/

Table 3.1: This is an overview of available processors and their peak FLOPS. Prices are collected from a Norwegian webshop in April 2007

Processor type	Cores	GFlops	Price (NOK)
Intel Core 2 Extreme Quad Core QX6800 2.93GHz 8MB	4	94	9950,-
Cell Broadband Engine	8+1	204.8	N/A
NVIDIA GeForce 8800	550	128	5049,-

the performance table 3.1. Of course it shcould be mentioned that the standard CPU, is also useful. Special purpose architectures can can only work on for example computations. Not general tasks, like handling logic and running an operating system.

Chapter 4

General-Purpose Computing on Graphics Processing Units

The GPU (Graphics Processing Unit) is a parallel coprocessor designed to do high performance visualizations in graphics applications. One of its main applications today is games; which has made it a commercial success story. Cutting-edge features demanded from the gaming industry and relatively cheap price due to its widespread popularity has recently made it interesting for general purpose computing (GPGPU). Throughout this chapter the concepts from last chapter is mapped to the GPUs parallel architecture and its peculiarities are illuminated in order to be able to implement efficient linear algebra. However, there is much secrecy around details in this architecture, so a general rule to speed is that things that look like graphics to the GPU should be fast.

4.1 Textures

An array in graphics memory is called a texture. The GPU can read from one input buffer, and write to another output buffer, but not use the same buffer as input and output, respectively, read-only and write-only. Therefore one run in the pipeline requires at least two textures. The graphics APIs offer one-, two- and three- dimensional textures, but the render texture has to be two-dimensional. The length of one-dimensional textures is also limited (4096 for NVIDIA G70).
Relatively short read-only vectors can be stored as one-dimensional textures, but if the elements in the vector should be updated by the GPU or the vector is to long, it is necessary to store the vector in two-dimensional texture layout. The vector can for example be packed like in Figure 4.1.

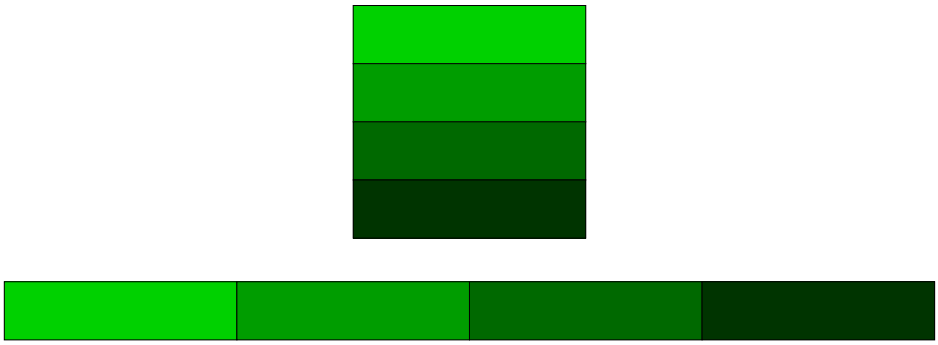


Figure 4.1: Packaging of a vector into a 2d-matrix

Dense matrices map very naturally to the two-dimensional layout of textures. However, to maximize speed it is important to consider if the most natural way is the fastest. Since a texture is designed to keep color images, each position in a texture can hold up to four components (red-, green-, blue-, alpha- channels). Because of limited cache, further discussed in a later section, it may be vital to consider utilizing these channels in a column-, row- or block-wise fashion in order to improve the chance of cache hit in the direction of your data. Of course, the gain in performance has to be evaluated against the cost of repacking. The processors are also pipelined in such ways that work on four-wide arrays are just as fast as working on a scalar value, making the computation four times faster as long as it is not bandwidth limited. This does not apply for the new generation G80 graphics card from NVIDIA, that is a scalar architecture.

4.2 The architecture

The graphics processor is a SIMD architecture capable of doing lots of arithmetic operations in 32bit floating point precision. To utilize this for general purpose, every step in the parallel algorithm has to be converted to render operations. For general purpose usage the most interesting parts of the render pipeline are the two programmable processors, the vertex- and fragment- processors, and one special purpose device called the rasterizer. The vertex processor is designed to work on vertices, while the fragment processor is designed to do per-pixel operations. Since there in most applications are more pixels than there are vertices in the geometry, the fragment processor is more powerful, so the heaviest calculations should be done in this part of the pipeline. Due to its capability to render independ-

ent pixels at high speed it is great for data-parallel applications.

4.2.1 The programmable pipeline

The programmable graphics pipeline is shown in Figure 4.2. First, two streams are sent to the vertex processor. In GPGPU the first stream describes the computational range of the problem and the other stream is texture coordinates and describes the computational domain. In addition, the vertex processors can read textures. The vertex processors can do calculations on all these streams, to change the range, domain or something else that linearly changes over the grid. Then the result is sent to a very efficient special purpose device, called the rasterizer in which geometry is turned into fragments. The rasterizer linearly interpolates the texture coordinates, so they can be used as addresses to look up values in a texture, and sends them to the fragment processor, thus the rasterizer can be recognized as an address interpolator. The next step is the fragment processor which is designed to take the computational load, before the results are written to a texture.

The fragment program can be compared with the "inner loop" on a CPU that iterates over elements in a stream. The counter in the "inner loop" that indexes the arrays is replaced with the results from the rasterizer hardware. The fragment processor on NVIDIA G70 is able to output up to four values per target in four targets and one depth value.

There are still some steps in the rendering pipeline that may have some occasional interest. The depth buffer can be used to find a subset of values, where one of them is the maximum [HHL+05], or it can be used for some other branching purpose where a set of values has to be checked. This mechanism can also be used for something called early z-cull that eliminates calculations on elements that are not affecting the result. Another feature is the occlusion query that is designed to count the number of rendered pixels and return the answer, without stalling the pipeline. This feature can potentially be used to check equality of vectors [HB05]. There are still some post-processing operations left, like blending, but on NVIDIA G70, these operations are not supported for 32bit floating point.

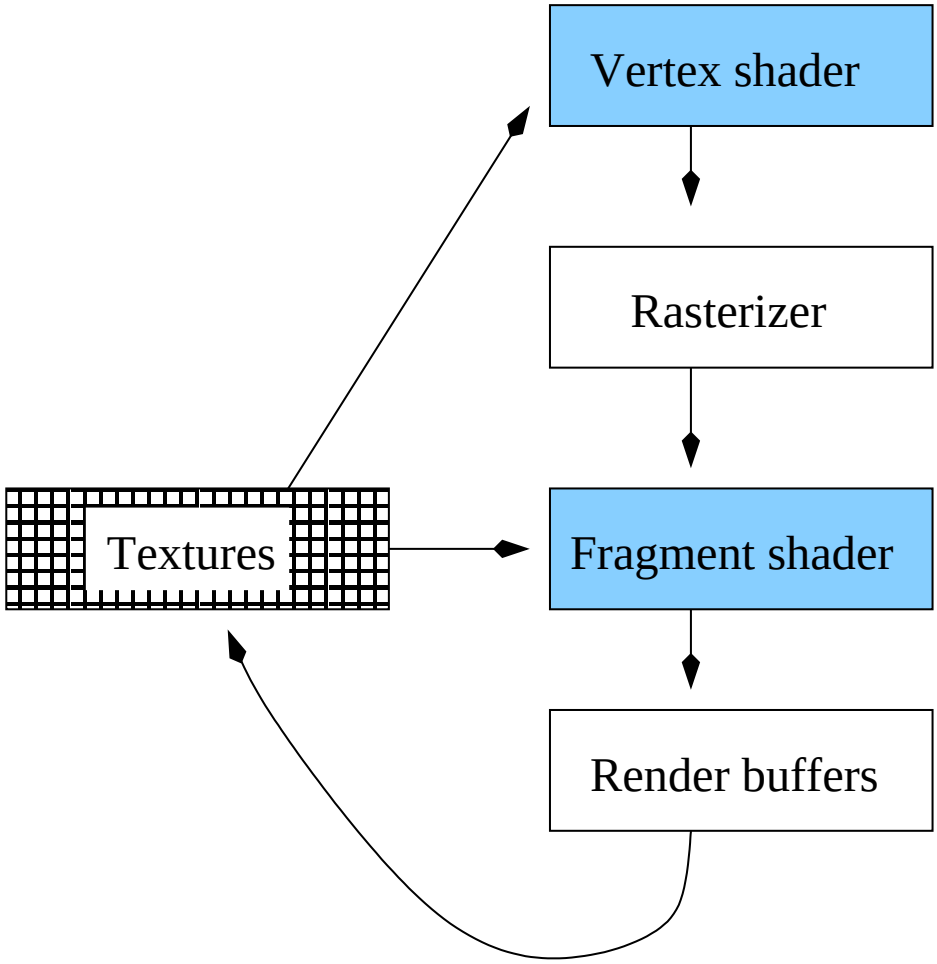


Figure 4.2: The programmable pipeline: Shader model 4 hardware, introduced with DirectX 10, may also specify a geometry shader, to be executed between the vertex and fragment shader.

4.2.2 Memory management

There are different design goals for texture memory on the graphics chip and main memory in a computer. CPUs have natively little parallelism in the instruction set so it is essential that memory references are returned from main memory with as little latency as possible while texture memory is designed for high throughput only. The GPU is pipelined in such a way that non-dependent operations are done if some are available, while high latency out-of-cache memory is fetched, to hide the additional cost of texture read. Therefore most high-end graphics chips are equipped with ddr3 or ddr4 memory, while high-end main memory is the well established ddr2.
To further lower the latency and compensate for bandwidth, the CPU has some extensive hierarchical caching mechanisms. On the GPU the cache is designed for a completely different purpose. The cache is designed to accelerate texture filtering and is therefore caching relatively low amounts of data in two directions. Locality is therefore important when GPU algorithms are optimized for cache. For algorithms that read large amounts of data once, the speed on the CPU will, in spite of caching, fall back to memory speed, so in bandwidth hungry applications the GPU easily outperforms the CPU.

4.2.3 Shader Model 4

Shader model 4 were introduced with DirectX 10. The main difference is that the shader execution units are unfor, which means that the processors are shared between the vertex processing stage and the fragment processing stage, allocating resources where it is most needed.

4.3 Communication

The graphics processors are designed to work on small independent parts of a stream, thus there is no communication between the processors. Each time a processor needs values computed on another processor, it has to finish execution of the kernel and render the results to shared memory. The enormous bandwidth can compensate for the lacking capability to store intermediate results in cache on large data sets, but since cache on the CPU is faster than texture memory, very cache-friendly algorithms on the CPU may still be faster. On the upside, there is little need to worry about synchronization issues because each processor writes to its own restricted

part of the memory.

For communication between the graphics chip and the main board there is both a bandwidth and latency challenge. The bandwidth over the PCI-Express x16 bus is up to 4GB/s in each direction, but in practice it can be under half that speed, which is severely slower than the internal memory transfer speed on the graphics chip. To hide the overhead from first transferring data to texture memory and then read it back, it is necessary in between the two operations to iterate over the transferred data set many times. The GPU is able to do independent work while a new texture is uploaded; efficiently hiding both the latency and eventual bandwidth limitations from streaming of the texture over the bus, but readback from the render target stalls the pipeline, because it is required that the rendering of the image is completed before the data can be transferred, introducing latency from both later restart of the render pipeline and transferring memory over the bus, eventually making the performance suffer.

During traditional readback, neither the CPU nor the GPU is able to do anything else than wait, first for the other side to catch up, if it is slower, and then for the data to be transferred. As long as the render target is read back, it is impossible to do anything, but there is an extension called Pixel Buffer Object that may help in some cases. PBO is actually an extension to Vertex Buffer Object (VBO), which is there to instantiate geometry from rendered data by copying it into a buffer that can be used by the vertex processor. The PBO extension includes some additional targets for data from the render target allowing asynchronous readback. The PBO extension allows for copying of data in high-bandwidth graphics memory. After this is done, the GPU is free to start again with more rendering. On the CPU the, readback call is non-blocking and returns immediately, hence other important work can be done while the data is read back with help from the DMA [Elh05]. When the data for sure is in system memory, it can be accessed and processed. Traditional synchronous readback is still faster if there is no work to be done while waiting for memory to be transferred.

Many parallel algorithms are actually designed to reduce the latency from memory transfers. Especially techniques designed to lower latency on clusters with message passing are interesting. This topic will again be discussed in chapter6, but the essence is that the algorithm is rewritten to send less often, but in larger blocks.

4.4 Communication Patterns

In chapter 2 a few patterns of communication were mentioned. Those were broadcast, reduction, gather and scatter. This section will describe how these patterns adapt to the GPU.

Broadcast: Broadcast adapts very well to this architecture. If the value to be broadcasted is on the CPU, it can be sent as a uniform parameter, which is a "runtime constant" that is distributed very fast to all GPU- kernels. If the value reside in texture memory, each fragment program can look it up.

Reduce: A lot of effort has been put into detection of the most efficient way to reduce a set of values to a smaller set or only to one scalar, either by summation or searching for the element with e.g. highest value on GPUs. This operation can frequently be found in linear algebra algorithms as inner products, and is generally popular in algorithms because of its cache friendly sequential access behavior. There are multiple procedures for how to do this on the GPU. Reduction can be performed by alternately render and read from two buffers (ping-pong) in multiple passes, and render fewer values in each pass [BP04]. [BP04] also proposes to read back before there are fewer elements left than the degree of parallelism offered by the graphics processor. [HHL+05] finds the maximum in an "all-reduce" operation exploiting the depth buffer, and reads back to the CPU in the end for final calculations. [GGHM05] runs a sequential fragment program at a single texel location, and reads back that texel in search for a pivot. [GGHM05] also provides a figure showing that this step has little performance impact on the underlying algorithm for partial-pivoting, even though there is no parallelism in this approach.

Gather: Gather can easily be implemented on a GPU. Two textures can be used such thats one is an address texture and the other one contains the values. This can for example be used to represent sparse matrices as in GPU GEMS 2 [**?**].

For each value there will be two texture fetches, where the second is depends on the result of the first. This may lead to poor performance. First of all, if there is no non-dependent work left for the fragment processor to do while it fetches memory, the pipeline will stall and clock cycles are wasted. In addition, texture fetches will take more time, because of poor random access performance [Buc05][Page 510]. Dependent texture fetch also breaks the possibility for the graphics chip to prefetch larger segments

of memory, altdoug if the dependent pixel is in cache e.g. a neighbouring pixel it should be fast.

Scatter: Scatter is much harder to implement, since all fragment addresses are pre-computed, before execution of the kernel. [Buc05] has done some work on a couple of cases were scatter can be seen as gather and a way to scatter values by also outputting memory addresses, and then sort by these addresses in multiple passes.

If only a few values are to be scattered in a large texture, [Buc05] proposes to render points, with the vertex processor, but the snag is poor usage of the rasterization hardware, and higher chance for collisions. Still, in chapter 5 this method will be used.

4.5 Programming

When the graphics chip is programmed for general purpose, the legacy from graphics APIs, which is designed for just graphics rendering, is noticeable. Knowledge of OpenGL or DirectX is necessary to avoid hollow error messages like "OpenGL: Invalid operation". New technologies targeting general purpose usage has emerged, that help abstracting away the most excessive difficulties.

GLSL and Cg: The graphics APIs have support for shader languages that is used to program the graphics processors. GLSL and Cg are examples of high level "C-like" languages that provides an interface to the parallel hardware. After compilation, kernel operations are executed in parallel on entire streams. A GLSL shader is shown in Listing 4.1. The languages offers natively vector data types with up to four components that can be reordered arbitrary at no cost with an operator called swizzle, and some built-in functions, like the dot- product.

Shallows: Shallows is a C++ library built upon OpenGL that abstracts away the OpenGL calls for the user, but still allows users to control OpenGL directly if that is wanted under some circumstance. The library provides easy creation of the environment necessary to run vertex and fragment shaders designed in GLSL or Cg with textures as input and output, and also easy access to error checking functionality.

Glift: Glift [?] is a STL like generic template library for the GPU for algorithms and data structures. The library is programmed in Nvidia C for graphics and C++, and offers a large amount of reusable data structures. A library like this is a good idea, because like STL, it will be able to save considerable time for development. A challenge right now is to maintain the library and include new algorithms as they get introduced in a high rate.

CUDA: Computed Unified Device Architecture, Cuda, is an extension to the C-programming language designed by NVIDIA for the G80 generation of graphics processors. CUDA extends the C-language with a few extra function type qualifiers and some other notion that allows the compiler to determine which functions that shall execute on the GPU, and which functions that shall execute on the CPU. CUDA offers a few libraries, where the cuBLAS library, that includes BLAS functionality for the GPU, is specially interesting for the scope of this thesis.

4.6 Branching on the GPU

As for inner loops on the CPU, branching should be minimized on the GPU. When a branch is taken by some, but not all fragments, many processors will execute both branches. For the Nvidia GeForce 7 series a conditional branch will process 880 pixels. I have unfortunately not found any other places than this article [1]. It should be mentioned that branching is getting better on all new generations of graphics chips, but still; if it is possible the branches should be moved to the CPU, or the preprocessor, or alternatively the expression should be rewritten, so that there is no need for branching.

4.7 Computational superiority

Through the former sections we have clarified some of the things to consider when programming the GPU for general purpose, and tasks that fit the programming model can benefit from for example higher-bandwidth memory. In this section the computational superiority of the GPU is illustrated. The flexibility of the CPU comes with a cost. A huge part of the

[1] http://www.extremetech.com/article2/0,1697,2053310,00.aspthatcanverythenumber

Listing 4.1: Fragment shader

```
uniform sampler2D matrix;

void main ()
{
  //Read vector from the texture
  vec4 top = texture2D(matrix, gl_TexCoord[2].st);

  //Read a float from the texture
  float multiplier = texture2D(matrix, gl_TexCoord[1].st).x;

  //scalar-vector multiplication
  top = top * multiplier;

  //Read a new vector
  gl_FragColor = texture2D(matrix, gl_TexCoord[0].st);

  //Subtract the vectors and save the result to the pixel
  gl_FragColor -= top;
}
```

transistors on the CPU is devoted to control mechanisms to direct communication or branching in the software. On the GPU, branching has over the years become better, but there is still far less transistors used to control the processors on the GPU, with the result that multiple processors are treated uniformly. To compensate for slower main memory and the demand for low latency in each memory fetch, the CPU uses almost half the transistors on the chip for cache. However, the GPU is designed to work on large streams of data, which will not fit in the cache anyway, so there is only a little cache available. The transistors saved on reduced cache-, and control-mechanisms on the GPU is put into the Arithmetic Logic Unit. In applications that have a high rate of arithmetic operations, like the Black-Scholes [Wik07a] PDE, the GPU can really outperform the CPU. In a Peakstream based implementation [Pea06] there was a speed-up of 28x on an ATI R580 GPU versus a dual Intel Xenon processor. See figure 4.3 for a rough schematic comparison of the distribution of transistors on the CPU and the GPU.

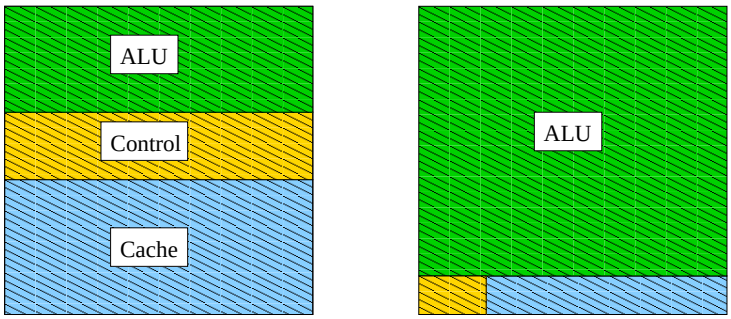


Figure 4.3: Schematic comparison of the distribution of transistors on the CPU and the GPU

Chapter 5

LU factorization

During 1951 a programme for solving simultaneous linear algebraic equations was used for the first time

–MICHAEL WOODGER
The History and Present Use of Digital Computers at the National Physical Laboratory (1958)

My LU-implementation is designed to solve a dense linear system of equations, $Ax = b$, on the GPU. The algorithm set for this task is the LU-algorithm. First there will be a presentation of the algorithm, before related work is introduced. Then there will be a presentation of this implementation followed by an analysis of the results from the implementation.

5.1 LU- decomposition

LU- decomposition is a factorization technique utilizing basic row operations to obtain triangular forms that easily can be solved with substitution, $A = LU$. Because it inherently breaks down into a factorization phase and a substitution phase, the two phases will be discussed separately, starting with the former. The Crout- and the Doolittle algorithms are the two most common LU- decomposition algorithms, but because the differences are minor and Doolittle is chosen for later implementation for simplicity reasons, the theory in this chapter will apply to the Doolittle method. The difference is that with Doolittle there are ones on the diagonal in the lower

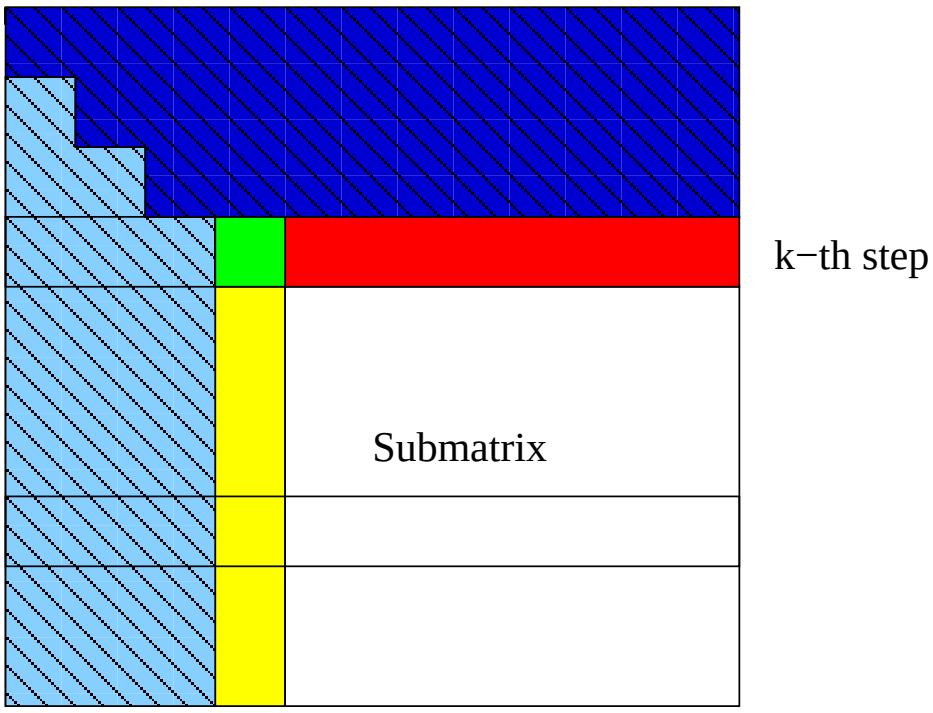


Figure 5.1: LU-factorization

triangular matrix, and with Crout the ones are with the upper diagonal matrix.

5.1.1 LU- factorization

The factorization phase is two-split. First multipliers are calculated, before the remaining submatrix is updated. The multipliers are the multiple of the entry at the upper left position in the k-th step subtracted from the first column of the rows below. At each position in the submatrix the product of the corresponding entries in the k-th row and the multiplier column is subtracted from the underlying submatrix, in concordance with figure3. Once Doolittle is finished there is a unit lower triangular matrix L, and a upper triangular matrix U, related to A by the matrix multiplication, $A = LU$. Both matrices can be stored in one matrix as is done in the vectorized algorithm 5.1. Notice the predictable indexing and data-parallelism qualities.

Listing 5.1: Vectorized LU

```
1 for k=1:N-1
      A(k+1:N,k) = A(k+1:N,k) / A(k,k)
      A(k+1:N,k+1:N) = A(k+1:N,k+1:N) - A(k+1:N,k) * A(k, k+1:N)
  end
```

5.1.2 Substitution

When the factorization is complete, forward and backward substitution may be done in order to solve the system, $LUx = b$. First $Ux = y$ is set, and $Ly = b$ is calculated by forward substitution.
Then $Ux = y$ is calculated by backward substitution and the solution to

Algorithm 1 Forward substitution:

for $k = 1$ to $N - 1$ **do**
$y_k = b_k - \sum_{i=1}^{k-1} L_{ki} y_i$
end for

the system is obtained.

Algorithm 2 Backward substitution:

for $k = n$ to 1 **do**
$x_k = \frac{y_k - \sum_{j=k+1}^{n} U_{kj} x_j}{U_{kk}}$
end for

5.1.3 Pivot

If the upper left entry in k-th step is zero, $a_{kk} = 0$, the algorithm will attempt to divide by zero and break down. If the absolute value of the entry is small, the multiplier may get large and errors in the submatrix will be enlarged caused by finite precision in floating point arithmetic. The example below borrowed from [Hig96][Chap. 1] illustrates this phenomena well.

$$A = \begin{pmatrix} \epsilon & -1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ l_{21} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{pmatrix}, 0 < \epsilon \ll 1$$

$u_{11} = \epsilon$, $u_{12} = -1$, $l_{21} = \epsilon^{-1}$, $u_{22} = 1 - l_{21}u_{12} = 1 + \epsilon^{-1}$. If ϵ is sufficiently small, u_{22} evaluates to ϵ^{-1} When L and U are multiplied and subtracted from A, there is an error in the result.

$$A - \hat{L}\hat{U} = \begin{pmatrix} \epsilon & -1 \\ 1 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 0 \\ \epsilon -1 & 1 \end{pmatrix} \begin{pmatrix} \epsilon & -1 \\ 0 & \epsilon^{-1} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

However, if the rows in pear. There are several strategies to keep the multipliers small, in order to avoid accumulation of large errors. The most common strategy is partial pivoting which searches for the largest absolute value entry, the pivot, in the k-th column in the k-th step and does a corresponding row interchange. The risk of breakdown is simultaneously eliminated.

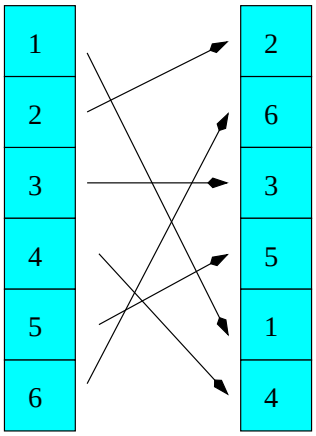
$$pivot_k = max_{k \leq i < n} |a_{ik}|$$

On the cpu row interchanges can efficently be represented with a permutation array, representing the reordering. See Figure 5.2(a). Initially the permutation array contains an ordered sequence of numbers from zero to n. Each time rows are interchanged the corresponding positions in the permutation array are interchanged. Then p(k) is used to represent row k. This will ensure minimal overhead. Reordering the rows is essentially the same operation as premultiplying A with a permutation matrix and calculate $PA = LU$. To solve this system, the right hand side also has to be premultiplied with P, so the system to be solved is $LUx = Pb$.

5.2 Related work

LU- decomposition is earlier implemented on the GPU by [GGHM05], and this effort has functioned as a guideline to what to give full attention to, and what is less important to focus on through the work described later in this chapter. This section summarizes some of [GGHM05]'s work and outlines ideas from this implementation that is brought further. The section is split so that the parts that directly relate to the LU- algorithm outlined in a later section get more space.

[GGHM05] implements and tests both the Gauss-Jordan and the LU-decomposition algorithms, and the LU- decomposition is tested with partial pivoting, full pivoting and without pivoting. It provides an analysis of texture accesses and arithmetic operations for each fragment and the total number of updates in the two algorithms and concludes that LU is faster



$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

(a) (b)

Figure 5.2: Rowinterchange: In the left figure the representation of row interchanges is viewed and to the right the corresponding permutations matrix is shown.

than Gauss-Jordan on the GPU.

One of the sweetest things about this implementation is that due to the GPU?s massive data-parallelism design the LU-algorithm shows as a very clean and clear non-complex algorithm. It is basically just the vectorized algorithm of the standard LU-algorithm and because the vectorized version favors a sequential access pattern for memory as the GPU does for texture accesses the two seems like a perfect match.

The implementation features a very clean and intuitive mapping of matrices to textures that matches the two dimensional data layout of GPUs. The matrix is directly mapped to a one-channel 2D texture and in each pass the lower right (N-k) * (N-k+1) matrix, that matches the pattern of a large quadrilateral, is updated. This allows for high utilization of the graphics pipeline.

It is easy to keep track of the number of passes that is done, so the top row and multiplier column in k-th step is during update of the remaining submatrix found with help from texture coordinates. This quality of the implementation is dubbed index pair streaming. The advantage with this model over computed index pairs is that this allows for memory to be

prefetched by the fragment processor. To utilize available cache the fragment processor fetches memory in large 2D blocks. Since the index pairs are a column and a row the fetches of these two parts in a fragment should match the spatial locality of these blocks and get fetched from cache and therefore put little stress on the bandwidth, allowing for a faster execution. Texture cache and block sizes are kept secret by graphics vendors, but since the rasterization engine can be used, the design is cache oblivious and should therefore perform well on a GPU.

Both partial and full pivoting are implemented and tested. Both methods sweeps the matrix for a pivot to increase the stability of the linear set, but full pivoting is significantly slower than partial pivoting and is therefore a less often wanted strategy. The partial pivoting implementation run a fragment kernel in a single texel that loops over the texture and write the address of the highest pixel to the target. Then that texel is read back to the cpu.

Since pointers based on dependent-texture fetch are very inefficient on GPUs, the algorithm runs a copy fragment program for row interchange. The two rows are rendered to the correct position in the target, and then rendered back to the source. This approach is more efficient, because of the high bandwidth on graphics memory. Since rows are actually swapped, the main drawback of pointer interchanges on CPUs, namely thrashing of cache on frequent row interchanges, is eliminated.

For every pass in the LU algorithm with partial pivoting, one single pixel find pivot program will run, two instances of the copy fragment program will run to interchange rows, one normalize program to calculate multipliers, a new instance of the copy program to render the multipliers back to the source buffer and then a row operation program that updates the remaining submatrix, before proceeding to next pass.

Only data stored in texture memory is used, so the bandwidth will be fully utilized during every stage in a pass, and results from benchmarking shows that the algorithm is a little faster than ATLAS for partial pivoting. Another interesting result is that the performance impact from readback and row interchange in LU with partial pivoting is relatively small compared to LU with no pivoting.

The High Performance LINPACK benchmark (HPL) [APC04] implementation of LU is designed to benchmark supercomputers. Much of the

work done in this implementation does not relate to implementing LU on a single node e.g. block cyclic distribution of data, but one particular optimization is very relevant. The lower triangular factor L is applied to the right-hand side b, as the factorization steps forward. This trick is the key to one of the optimizations done in my implementation.

5.3 The Implementation

Much of the effort put in this implementation is inspired by the LU on GPU work of [GGHM05] and the details of this will be reflected in this text. For speed, flexibility and stability reasons the algorithm that is implemented and tested is the vectorized LU- algorithm 5.1 with partial pivoting. Even though parts of the design have much resemblance with the [GGHM05] implementation, there is one major difference in the concept. While [GGHM05] does almost everything on the GPU, this implementation spreads tasks to both the CPU and the GPU based on the degree of data- and task- parallelism in each step in the algorithm, to optimize for best utilization of the strengths in both architectures and simultaneous execution of tasks. The other main difference is that this algorithm in addition to factorization also performs forward and backward substitution to obtain a final solution from solving a linear system.

5.3.1 Design goals

During the design phase of this implementation I sat the following guidelines to focus on, for optimum usage of the graphics hardware. These guidelines are partly based on Galoppo et. al. [GGHM05]'s analyses of what lead to most speed, and partly some consequences that may occur, when other optimization techniques are applied.

Index pair streaming

Memory locations in the LU- algorithm can be read in a very predictable pattern, allowing the vertex processor and the rasterizer hardware to compute where to find values. Since these addresses are computed outside the fragment processor, the graphics processor can pre-fetch blocks of memory to the fragment processor, which should lead to faster execution. Galoppo et. al. [GGHM05] streams the addresses for the top row and the multiplier column to the vertex program and the rasterizer interpolates the result and send the addresses to the fragment processor. In their

implementation of the algorithm it yielded a 25% speedup over computed index pairs.

4-wide vectors

Since the GPU is designed to work on four component color vectors, there will be a theoretical computational speedup bound to packing the matrix such that every calculation is done on four component vectors. The extent of this optimization is hard to predict, since there is an increased chance that the algorithm will be bandwidth limited because more memory is read simultaneously, and the algorithm has relatively few independent arithmetic operations that can hide the cost of texture fetches. Galoppo et. al. [GGHM05] observed close to peak bandwidth on NVIDIA Ultra 6800 on their one-component texture algorithm.

There may also be an increased amount of wasted work in the algorithm since the LU eliminates one row and one column in each pass, and not a multiple of four values, but this may depend on the packing of the matrix. This issue will be discussed further later.

If the 4-wide vectors are put either along the rows or along the columns in the texture, there will most probably be an increase in speed, either because there is 4 times as many top row elements read from cache or there is 4 times as many multiplier column elements read from cache as index pairs when a block is processed.

Even though there will not be an increase of a multiple of four in speed, the bandwidth will be fully utilized more often, since it is a better chance for bandwidth limitations when four times as much data are processed in a block.

Simple matrix representation

Repacking a matrix for more efficient representation on the GPU in main memory can potentially be costly due to cache misses. If the algorithm packs four components along the rows or the columns the access pattern will match either Fortran or C style arrays and can be mapped directly to 2D texture memory. If the matrix size is not a multiple of four it can easily be padded in the process of copying it to driver controlled memory.

R	G	B	A	R	G	B	A	R	G	B	A	R	G	B	A
				(1,1)(1,2)(1,3)(1,4)				(1,5)(1,6)(1,7)(1,8)				(1,9)(1,10)(1,11)(1,12)			
				(2,1)(2,2)(2,3)(2,4)				(2,5)(2,6)(2,7)(2,8)							
				(5,1)(5,2)(5,3)(5,4)											

Figure 5.3: Packing of components in a texture

More effective solution for row interchanges

On the CPU, rows are swapped with pointers in memory. A "pointer texture" is not very efficient on the GPU because it will lead to dependent texture fetch. [GGHM05] solves this by copying rows to the right location, utilizing the high bandwidth texture memory, to copy 4*4*matrix-width bytes of data in each pass to get the rows to the correct location. On a CPU the maximum copying at this step in each pass is 3*4 bytes if an int array is used to represent the pivots, but [GGHM05] argues there may be an added cost from lots of cache misses during update of the submatrix on the CPU.

"Pointer textures" is a non-solution, but there is a lot of copying in the [GGHM05] implementation that may affect performance. A hybrid approach where less data is copied could potentially increase performance, but since data must be copied between two buffers (ping-pong) this guideline was a hard challenge to overcome.

5.3.2 Matrix representation

A matrix is represented as a texture on the GPU. A matrix matches the two dimensional layout of texture memory very well. In this implementation the width of the matrix is divided by four and mapped to a texture with four-wide sub-arrays in a row-wise fashion. See figure 5.3 for illustration. If the matrix width is not a multiple of four it has to be padded to the right width. The reason why the matrix is packed in this style, is that it allows for a minimum amount of data to be copied for row interchanges.
Four component textures is chosen because most graphics processors are built to work on four- color textures (Red, Green, Blue, Alpha). This is a design choice done to utilize as much GPU power as possible on all available hardware when working on the update submatrix part. In addition

to the matrix there is one special purpose texture column to the left in the texture, used for multipliers and intermediate results.

5.3.3 Flow in the implementation

The implementation is divided into two shaders that execute in each pass; one for each data-parallel operation in the vectorized algorithm 5.1. The first shader calculates all the multipliers in the k-th pass and stores the results in the left most column in the texture. The other updates the submatrix based on the previous calculated multipliers. When a pass is finished the buffers are swapped and the procedure is done over again.

5.3.4 Row interchanges

The algorithm is in particular designed for partial pivoting. The challenge is to interchange rows with minimum impact on performance. In this approach to the problem, rows are scattered to the correct location during calculation of new values. Since the algorithm has to ping-pong between buffers on the GPU, the k-th row in k-th step should after it has been updated with respect to the corresponding multiplier and the pivot row, be located at the pivot row's index in the other buffer. The pivot row could then be copied to an "upper-diagonal" texture to be saved in anticipation of backward substitution. If the k-th row is rendered to the "pivot row location" in the other texture during the row-operation and multiplier calculation part of the LU-algorithm, simultaneously as the other rows that are not interchanged are processed, the only cost not eliminated is copying the pivot-row to the "upper-diagonal" texture once in each pass.

The figure 5.4 tries to illustrate the procedure. Red is the color of the k-th row, blue are rows that are just updated and rendered to the target buffer in the right figure, but not moved to another index and yellow is the color of the pivot row. The source buffer is the left-most column in the figure and a pivot is selected in it to be used in calculation by the other elements. During both calculation of the multipliers and row operations in the lower right sub-matrix, the blue elements are updated and rendered to the corresponding indices in the right buffer. Simultaneously instructions to execute the same computational kernel on the red element, specified by texture coordinates, from the source buffer and render it to the index corresponding to the yellow element from the source buffer, specified by vertex coordinates, but in the target buffer, are queued to the graphics

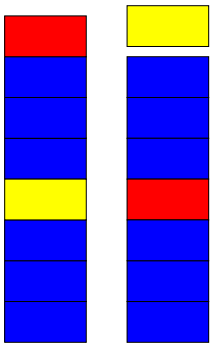


Figure 5.4: Rows are read from the left figure, permuted, and written to the right figure

pipeline. When the instructions have finished executing, the updated red element has been scattered to the correct location and all elements that is needed for continues reduction are ready for next step in the algorithm. This operation should be fast because the location is after all just geometry. The yellow pivot row will not be used further in the factorization algorithm and will have to get copied to another buffer, utilizing the high bandwidth of graphics memory, to not get overwritten in the next pass. Although this is a copy operation, relatively few bytes are copied.

The observant reader may have discovered that something does not add up. The k-th row is rendered directly to it's new location as it is updated in the factorization process, and there is no additional step to maintain the order of the rows in the lower triangular matrix based on earlier calculated multipliers, as rows in later passes are interchanged (hatched area in k-th row and pivot row in figure 5.1), thus leading to columns with dissimilar permutation. Like in the high performance LINPACK (HPL) [APC04] benchmark algorithm the multipliers are applied to the right hand side b as the factorization progresses with the result that the system $Ly = b$ has been solved when the algorithm ends. Since previous multipliers are of no further usage for the proceedings of the algorithm on the GPU, they are read back to the CPU in the dissimilar permutation order they are calculated in. This trick allows for the shader that calculates multipliers to also interchange the indices in intermediate results from solving $Ly = b$. The exact details of this procedure are discussed in the next sub-section.

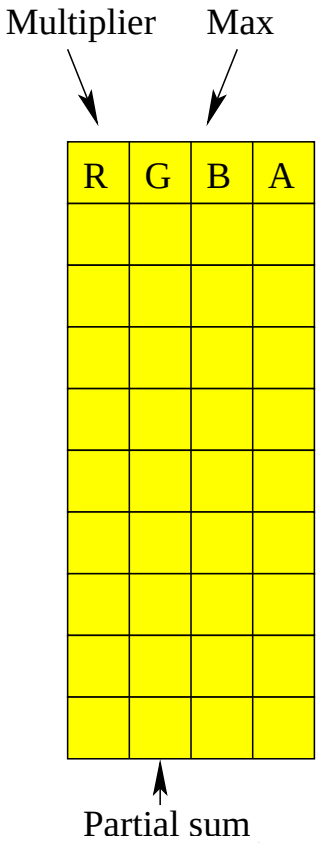


Figure 5.5: Overview of how values int the left-most column in the texture is placed.

5.3.5 Calculation of multipliers

Multipliers are calculated and rendered to the left-most four-wide vector column in the texture, but two more operations are also done in the process. Just as in the HPL algorithm [APC04] forward substitution is done as the factorization of the system progresses. In the substitution algorithm 1 the most computationally demanding operation is the inner product that is done for each row in the lower triangular matrix. In each pass of the algorithm on the GPU we can do a multiplication between the multiplier column and the result that is calculated from the last pass, $y[k-1]$ from algorithm 1, that is streamed to the GPU as a float uniform, and add it to the partial inner products from rows from last pass.

The last operation done in this shader is the reduction of the first column

in the remaining sub-matrix, and the result is rendered to the same pixels as the multipliers. The purpose of this reduction is that the texture column including the multiplier, partial inner products and reduced column is read back to the CPU, so the CPU can, while the rest of the sub-matrix is reduced, calculate a new y[k], and find a new pivot. See Figure 5.5 for the organization of the results in the texture row.

During execution of this shader two pixels are read in each fragment. The first is the last multiplier pixel where the partial sum from forward substitution and the first column from the sub-matrix are stored. The sub-matrix column from the multiplier row is used to calculate new multipliers. Then the next column in the sub-matrix is read to allow for calculation of new elements with a new pivot among them.

On the CPU the pixel column from the texture can be seen as a two dimensional array with a multiplier column, a search for pivot column and a partial sum column where the element found in the same row as the new pivot, is used to calculate a new $y[k]$ that in next pass is sent to the GPU, to continue inner product calculation. The location of the pivot is used to calculate new texture coordinates to decide how the next rows are swapped in graphics memory, and the pivot itself is streamed as a uniform float so that during calculation of multipliers the GPU will not have to look it up in the texture. The multipliers can either be droped or further permuted with pointer swaps on the CPU to create a lower triangular texture.

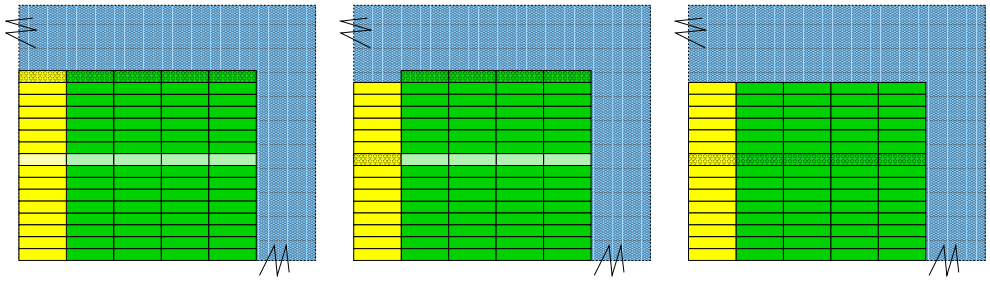
After multiplier, find pivot elements, and partial sum columns are calculated this column in the texture is copied back to the source texture, so the multipliers can be used to reduce the sub-matrix.

5.3.6 Reduction of sub-matrix

Reduction is the simplest procedure. Addresses to the multiplier column and the pivot row are streamed to the GPU with the index-pair streaming technique [GGHM05] and a the sub-matrix is reduced with respect to the multipliers and the pivot row. In every fourth pass the computational range of the sub-matrix is shifted one position to the left. This has to do with more efficient caching and will be further explained in next section when the algorithm is seen as a whole. When the reduction procedure has finished, the row that next will be the pivot row is copied to a U texture.

Listing 5.2: LU implementation

```
(CPU) Search for pivot row in main memory
bind buffer0 as source and buffer1 as target
(GPU) Copy pivot row to the U- texture.
(GPU) Swap rows and render multiplier column
for_each row
    (GPU) Copy multiplier column to source buffer
    (GPU) Copy multiplier column to readback buffer (PBO)
    (GPU) Swap rows and render reduced sub-matrix to target
    (CPU) Search for next pivot row (PBO)
    (CPU) Forward substitution (PBO)
    (GPU) Copy pivot row from gpu to U texture.
    Swap buffers
    (GPU) Swap rows and render multiplier column
end for
```



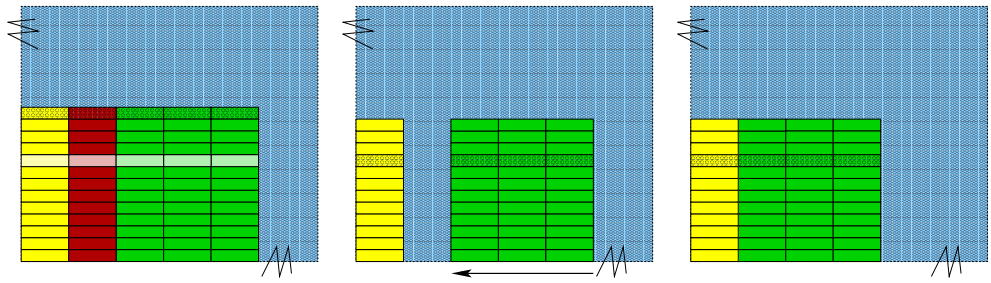
(a) (b) (c)

Figure 5.6: In the first figure the light row is selected for pivoting. In the next figure we can se that the shaded row in the multiplier shader is rendered to it correct location. In the last figure the submatrix is reduced and the shaded matrix is rendered to its corrrect location.

5.3.7 Overview of the implementation

Now, each part of the main algorithm has been explained. Here these parts are put into the bigger picture, and the relation between the parts is illustrated. First see listing 5.2 for the overall LU-factorization algorithm.

In figure 5.6(a) we have a matrix represented in a texture. The shaded area is the k-th row in k-th step in the texture and the lighter area is selected for pivot row. In figure 5.6(b) the multipliers are calculated and rendered to the correct location in the target texture. The new multiplier based on values from the shaded row, has been swapped and rendered to the pivot location. Then the rendered "texture-column" has been copied



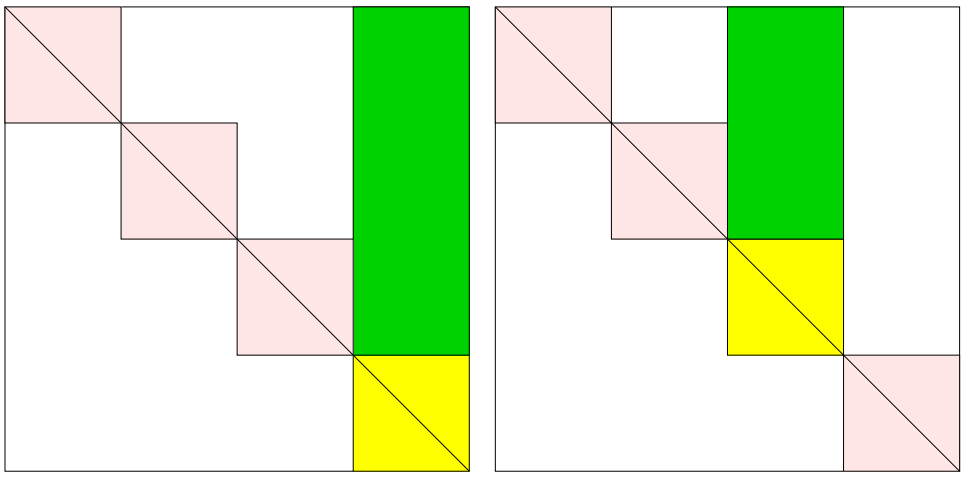
(a) (b) (c)

Figure 5.7: In this case there are no elements left in four-wide texture column, and the red row will collapse.

back to the source texture. In figure 5.6(c) a new sub-matrix has been calculated based on the multipliers and the sub-matrix from previous step. The lighter area was used as a "top-row" and the updated version of the shaded area has been rendered to the pivot position in the new sub-matrix. In the next step this result is used as input. There is one row less to consider in next step, but because of the packing the number of columns remains the same, until we have calculated four steps. In figure 5.7(a) we have selected a pivot row just as in figure 5.6(a), but the result from the reduction of the sub-matrix will require one column less than the input. In figure 5.7(b) we can see the red column is removed, but instead of rendering the output to the location shown, the whole remaining sub-texture is shifted one pixel-length to the left as in Figure 5.7(c). The reason the column-position is collapsed is to ensure that during multiplier calculation the first row in the sub-matrix always is read from cache. Performance will be analyzed further in next section.

5.3.8 Backward substitution

Backward substitution is done partly on the GPU and partly on the CPU. The matrix is dived into four blocks, where the CPU at all times calculates the substitution of the lower block, and the GPU does row vise inner products on the rows in the texture. This is clearly illustrated in the Figure 5.8(a) and Figure 5.8(a).



(a) (b)

Figure 5.8: The inner products in the yellow box is processed on the CPU, and the inner product in the green box is processed on the CPU. This is to save readbackoperation. In total the matrix is split in four parts.

5.3.9 Synthetic performance analyses

The two shaders executed in each pass is the multiplier shader and reduce sub-matrix shader. Both shaders use texture coordinates to lookup values in a texture and not computed indices, thus pre-fetching of texture memory should be possible. This section contains:fddfdf

Multiplier calculation The result from forward substitution in last step and the pivot value are broadcasted as a uniform to the GPU from the CPU, so the values will not have to get looked up in the texture. The shader still requires lookup of four fp32 values. Two of these values share a pixel, and since the sub-matrix is shifted to the left every fourth step the next value to be read will reside in the neighbour pixel, thus the lookup of this value should be from cache. The last value is the value next to the pivot in the pivot row for all fragments, so the fetch of this value should also be from cache. The arithmetic instructions done in this shader is a scalar multiplication, a scalar subtraction and a two-wide vector multiply add (MAD) operation.

Reduction of the sub-matrix The Nvidia SIMD GPU does one instruction on hundreds of pixels simultaneously. Since the pivot row has the width of the sub-matrix and the multipliers has the height of the sub-matrix it is natural to assume that in either direction the render target is updated the "neighbour fragment shader" already has read at least one of the index pairs that is needed into cache already, which will ease on the available bandwidth. Each fragment program read two four-wide vectors and a scalar multiplier value. The arithmetic instructions available in each shader to hide the cost of texture fetch are one four-wide multiplication and one four-wide subtraction.

Search for pivot While the sub-matrix is being reduced the multiplier column is read back to the CPU and a new pivot is searched for on the CPU. It is relatively little data that needs to be searched through and since the search for pivot can be done in parallel to reduction on the GPU and the CPU has faster cache, it should lead to increased overall throughput.

When the remaining sub-matrix is getting sufficiently small the sub-matrix is read back and the final calculations are done from inside cache on the CPU. This is because it is harder to utilize the parallelism of the architecture beyond a certain point, and the cost of communication is getting relatively high.

Memory transfers In each pass the pivot row is copied to an upper diagonal texture, and the rendered multipliers are copied to the source. Of course there is a cost related to this, but both copies are done fully in high bandwidth texture memory.

Wasted work Since the rows are packed as four-component sub-vectors and the algorithm eliminates one column at a time it is almost impossible to avoid wasted work in the computations. The left-most column in the matrix, column k+1 in k-th step is always calculated twice. In the case where the k+1 column is rendered to the alpha- position of the texture during reduction of the sub-matrix a whole pixel-column in the texture is wasted computations and memory accesses.

Table 5.1: This is the results from benchmarking the LU-algorithm

Matrix size	GPU LU	ATLAS
128	0,093	0,002808
256	0,156	0,026375
512	0,344	0,063747
1024	0,797	0,400158
2048	2,531	2,685458
2560	3,922	5,133072
3072	5,953	8,669463
3584	8,422	13,674318
4096	11,907	25,856191

5.4 Future extensions

The implementation as it is has one big limitation. The input matrix size must be a multiple of four. This is alright because it merely is a case study, but if the implementation was to be used in production, this issue can easily be fixed by shifting each row in the matrix the appropriate number of positions to the right when the matrix is copied to a texture, at no extra cost by utilizing the texture stream functionality in pixel buffer objects.

Another interesting feature would be to expand the implementation to work with multiple right-hand sides, as about the only change necessary would be that the multiplier shader works on a quad instead of a line.

Mixed-precision iterative refinement is another feature that could be interesting to examine for LU on GPU. Factorization of A and substitution is performed in single precision at single precision speed, and the only operations performed in double precision are the calculation of the residual and consecutive update of the solution.

5.5 Benchmarking

For benchmarking I have chosen random data, because the same number of operations are done no matter which values that are put in. For simple cheching of the answer the rows in the matrix are summed to a b-vector. See Figure5.9 and Table5.1.

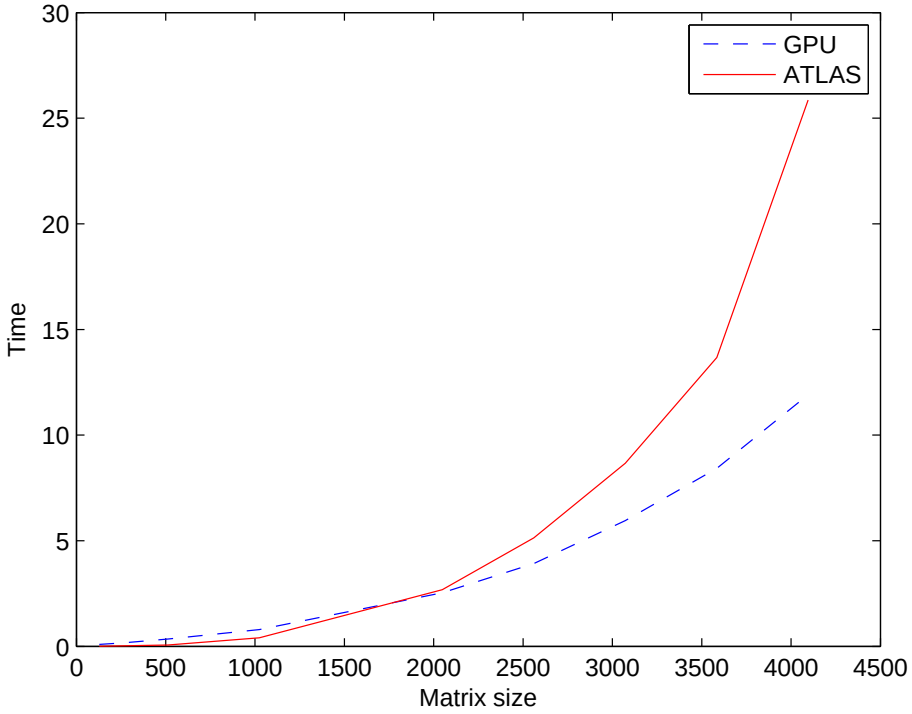


Figure 5.9: A graph showing the performance of the ATLAS Packing of components in a texture

Chapter 6

The Conjugate gradient method

The conjugate gradient method was originally proposed by Magnus R. Hestenes and Eduard Stiefel (1952)

6.1 Mathematical background

The Conjugate Gradient Method, (henceforth, CG), is an algorithm designed to iteratively solve a large symmetric positive definite linear system of equations on the form, $Ax = b$, where A is the matrix, b is a known vector and x is an unknown vector. The method is specially suited for solving of sparse systems, because the footprint in memory usually is smaller than for factorization methods. The need for efficient solving of this type of systems can be found in many important settings, including discretization of partial differential equations (PDEs). An initial starting vector x_0 is chosen and a new approximation x_{k+1} is computed from the previous x_k, by the formula $x_{k+1} = x_k + \alpha_k p_k$, repeatedly until the algorithm converges.

Choose a starting vector x_0:

$$(6.1)$$

$$p_0 = r_0 = b - Ax_0 \quad (6.2)$$

$$(x_0 = 0 => b - Ax_0 = b) \quad (6.3)$$

$$\gamma =< r_0, r_0 >; k = 0 \quad (6.4)$$

$$while \sqrt{\frac{p_k}{p_0}} > \quad k < kmax \quad (6.5)$$

$$t_k = Ap_k \quad (6.6)$$

$$\sigma_k = < p_k, t_k > \quad (6.7)$$

$$\alpha_k = \frac{\gamma_k}{\sigma_k} \quad (6.8)$$

$$x_{k+1} = x_k + \alpha_k p_k \quad (6.9)$$

$$r_{k+1} = r_k + \alpha_k t_k \quad (6.10)$$

$$\gamma_k = < r_{k+1}, r_{k+1} > \quad (6.11)$$

$$\beta_k = \frac{\gamma_{k+1}}{\gamma_k} \quad (6.12)$$

$$p_{k+1} = r_{k+1} + \beta * p_k \quad (6.13)$$

$$k = k + 1 \quad (6.14)$$

The Conjugate Gradient Method:
x_k is the approximated solution
α is step length
p_k is the search direction
r_k is the residual

A closer inspection of the algorithm above reveals that the most computational demanding work involved in each iteration of the method is the following:

1. One matrix-vector product
2. Two inner products
3. Three vector plus scaled vector additions

6.2 Related work

In this section related work will get introduced. Much work is done already. The conjugate gradient method has earlier been implemted on the GPU by

Bolz. et. al. [?] and also by Krüger et. al. [?].

6.2.1 Synchronization Overhead on Distributed Memory Multiprocessors

This subsection is based on the work of DAzevedo et. al. [?] for reducing the synchronization overhead on distributed memory multiprocessors. The reason this subsection is included is inner products. Inner products are a common operation in many algorithms and the CG- algorithm is not an exception. The standard formulation of the CG- algorithm requires two inner products, and the first inner product must be completed before the data is available for computation of the second inner product. Each time an inner product is calculated the result has to get read back to the CPU to interact as a scaling parameter or termination criteria, implying two stalls in the graphics pipeline and wasted time that could be used for calculation of the next iteration in the algorithm.

This is where the work presented by DAzevedo et. al. [?] can be used. They present ways to reformulate the CG method such that both inner products simultaneously can be calculated; hence one of the communication phases is eliminated, without compromising the stability of the original CG. The deduction of such a reformulation follows.

Rearangment of the method:
To derive the modified method, $p_k^T A p_k$ is expanded by substituting $p_k = r_k + p_{k-1}$

$$\begin{aligned} < p_k, t_k >= & < p_k, Ap_k > && (6.15)\\ = & < r_k + \beta_k p_{k-1}, Ar_k + \beta t_{k+1} > && (6.16)\\ = & < r_k, Ar_k > + \beta_k < r_k, t_{k-1} > + \\ & \beta_k < p_{k-1}, Ar_k > + \beta_k^2 < p_{k-1}, t_{k-1} > && (6.17)\\ & && (6.18) \end{aligned}$$

Symmetri of the coeffecient matrix and the matrix vector product (6.6) gives

$$< p_k, t_k >= \; < r_k, Ar_k > + 2\beta_k < r_k, t_{k-1} > + \beta_k^2 \sigma_{k-1} \qquad (6.19)$$

A property of the CG procedure (Orthogonality of residual vectors):

$$\frac{< r_k, r_{k+1} >}{< r_k, r_k >} = \frac{< p_k, Ap_{k+1} >}{< p_k, Ap_k >} = 0 \qquad (6.20)$$

$$r_k = r_{k-1} - \alpha_{k-1} v_{k-1} \quad (6.21)$$
$$< r_k, r_k > = < r_k, r_{k-1} > -\alpha_{k-1} < r_k, t_{k-1} > \quad (6.22)$$
$$\gamma_k = 0 - \alpha_{k-1} < r_k, t_{k-1} > \quad (6.23)$$
$$\beta_k = \frac{\gamma_k}{\gamma_{k-1}} \quad (6.24)$$

The result is the following algorithm:

(6.25)

$$r_1 = b, \gamma_1 =< r_1, r_1 >, p_1 = r_1 v_1 = A p_1 \quad (6.26)$$
$$\sigma_1 =< p_1, v_1 >, x_2 = (\gamma_1 / \sigma_1) p_1 \quad (6.27)$$

while $\sqrt{\frac{p_k}{p_0}} > k < kmax$(6.28)
$s_k = A r_k$(6.29)
$\gamma_k =< r_k, r_k >$ (6.30)
$\omega_k =< r_k, s_k >$ (6.31)
$\beta = \gamma_k / \gamma_{k-1}$(6.32)
$\sigma_k = \omega_k - \beta_k^2 \sigma_{k-1}$(6.33)
$\alpha_k = \gamma_k / \sigma_k$(6.34)
$p_k = r_k + \beta_k p_{k-1}$(6.35)
$t_k = s_k + \beta_k t_{k-1}$(6.36)
$x_{k+1} = x_k + \alpha_k p_k$(6.37)
$r_{k+1} = r_k - \alpha_k t_k$(6.38)
k = k + 1 (6.39)

The Conjugate Gradient Method:
x_k is the approximated solution
α is step length
p_k is the search direction
r_k is the residual

6.3 Implementation of the Conjugate Gradient method on the GPU

In my implementation of the Conjugate Gradient method I use the modified version deduced in the background material section of this chapter. The reason for this choice is communication. The GPU calculates inner products on four-wide sub-arrays, and the results from each of these sub-inner-products are added afterwards in a reduction shader, before the result is read back to the CPU. In the original CG there are two inner products which consequences in two reductions and two readback operations in each iteration in the algorithm. The modified version of CG allows for the inner products to be grouped for simultaneous calculation, and read back in one operation, thus the latency from communication in the implementation is minimized. In this implementation I have mainly focused on providing an efficient solution for matrices with some sort of sparse pattern, because with these types of matrices the CG-method really performs well, compared to factorization methods.

6.3.1 Splitting the work over multiple shaders

Parts that do not depend on output from work done to other locations in the computational range, are identified and grouped to be executed in the same shader, to minimize the number of overall render operations. In the algorithm above we can identify two groups in each iteration. Matrix-vector calculation and the multiplication part of the inner products can be put in one group and the four vector-plus-scaled-vector operations can be put in a second group. The grouping of operations requires multiple output targets, and this is supported through the Multiple Render Targets (MRT) extension on all commodity GPUs.

Matrix-vector and inner product shader: The matrix-vector and inner product shader calculates the matrix-vector product, $s_k = Ar_k$, and the inner products, $< r_k, r_k >$ and $< s_k, r_k >$, of four-wide sub-arrays. The accumulation of the results from the inner products is discussed in the next paragraph. Four rows in the matrix are processed in each kernel, which means that four rows are read from the matrix and multiplied with the matching elements in the vector, and written to the output texture. For a dense matrix, reading all elements in all columns in four rows is much work, but the implementation is mainly designed for sparse matrices. Another output texture, is used to output the results from the two inner

Listing 6.1: scaled vector-plus-vector

```
uniform sampler2DRect p;
uniform sampler2DRect r;
uniform sampler2DRect t;
uniform sampler2DRect s;
uniform sampler2DRect x;
uniform float alpha;
uniform float betha;

void main ()
{
  vec4 var1 = texture2DRect(t, gl_TexCoord[0].xy);
  vec4 var2 = texture2DRect(s, gl_TexCoord[0].xy);
  gl_FragData[1] = var2 + betha * var1; // t_k
  var1 = texture2DRect(r, gl_TexCoord[0].xy);
  gl_FragData[3] = var1 - alpha * gl_FragData[1]; // r_(k+1)
  var2 = texture2DRect(p, gl_TexCoord[0].xy);
  gl_FragData[0] = var1 + betha * var2; // p_k
  var1 = texture2DRect(x, gl_TexCoord[0].xy);
  gl_FragData[2] = var1 + alpha * gl_FragData[0]; // x_(k+1)
}
```

products, one scalar from each inner product.

Summing of the inner products: After the sub-parts of the inner products have been calculated, they are reduced by summation to fewer elements in a reduce shader, but when there are only a few values left the reduction is hard to parallelize further and the results are read back for final accumulation and usage on the CPU. Accumulating the inner products in this manner, see equations below, also has the advantage of minimizing the error bound in the final result as deduced in Higham [Hig96] [Chap. 3 p.70]. The shader reads blocks of neighboring values, and stores the result in an output texture.

$$s_1 = x(1:m)^T y(1:m) \tag{6.40}$$
$$s_2 = x(m+1:n)^T y(m+1:n) \tag{6.41}$$
$$s_n = s_1 + s_2 \tag{6.42}$$

Vector-plus-scaled-vector: The vector-plus-scaled-vector shader reads all vectors, and updates the values for next iteration. The domain where vectors are read from is linear and is calculated by the vertex shader, so the fragment shader just reads vectors and writes the result from the update. Five textures are read and four textures are written to, so the memory

bandwidth can be stressed, but compared with splitting up the work, there will be less texture read operations by stacking the vector update operations together. The implementation of this shader is shown in Listing 6.1.

6.3.2 Matrix and vector representation

The representation of matrices and vectors is a challenge. Vectors in texture memory must be represented as a two dimensional texture to be able to render to them. The matrix is static during iterations and should be represented in a pattern that leads to maximum throughput for matrix-vector calculations. Below the texture representation of the vectors and matrix are presented.

Vector: The vectors are originally 1D-data structures, but because 1D-textures have a limited length the vectors will have to get represented in 2D-texture memory, as in Figure 4.1. In the matrix-vector product shader it is necessary to consider the 2D-texture as a one-dimensional vector, since the shader processes all columns in four rows. To achieve this, the methods for translating 2D-1D addresses and 1D-2D addresses deduced in GPU GEMS 2 [**?**] are applied to be able to jump to the next row in the texture. In this implementation a total of nine equally sized textures are required to store vectors. The reason for nine textures is that the vector-plus-scaled-vector shader uses five vectors as input and four as output. Each vector is packed into multiple four-wide sub-arrays matching the channels available to store colors in, in the texture, so the number of pixels in the texture is the number of elements in the vector divided by four plus padding with zeroes in the end to get a full square 2D-texture.

Matrix: Representation of matrices for the CG-algorithm is a challenging task. To achieve the best possible speed it is necessary to consider sparse patterns in the matrices. Sparse patterns are non-zero patterns in matrices with many zero elements. Some matrices does not have a pattern, thus they require a more general storage structure. I have experimented with different types of matrix representations, including banded and random sparse matrices. For banded matrices the diagonals are counted and the distance between the diagonals are registered. Then the first four-wide pixel store the distance between the diagonals for the next four diagonals excluding the middle diagonal that has known distance. The next four pixels will contain the diagonals, and then the next pixel is used

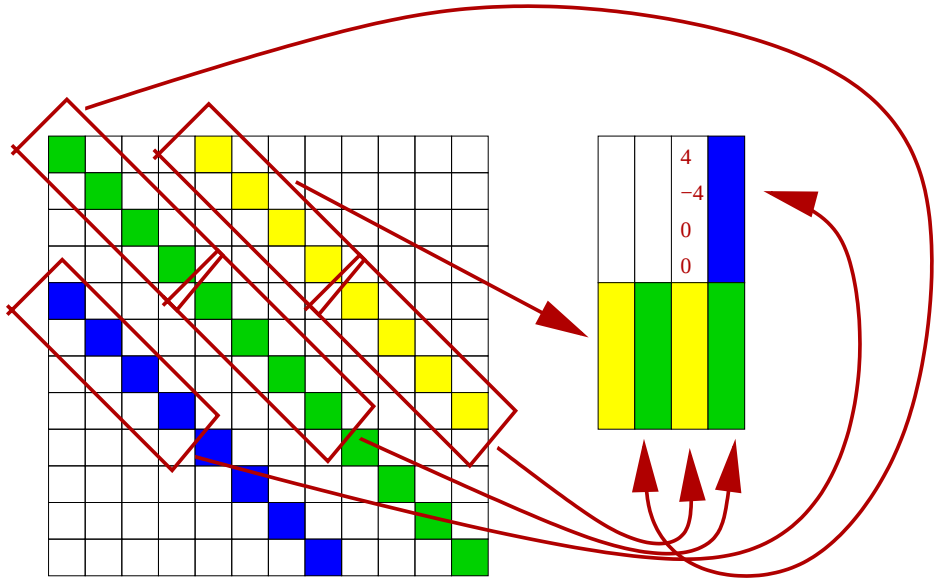


Figure 6.1: Packing of the matrix in the texture

for storage of the next distance pixel, and so on. Last the middle diagonal is stored. The series of the four-wide diagonals and their distance is squared and packed in a two-dimensional box with padding in the end if that is necessary to get a square box. Se Figure 6.1 for an illustration. To determine the end of the series, the last "distance pixel" will have to be zero-terminated, just like a text string. The same solution applies for random-sparse matrices. An element is seen as a four-row diagonal and stored in the same manner as diagonals. The main problem with this solution is that it requires branching in the shader, which can be slow if the branch diverges over the computational range.
To achieve more speed for diagonals, I implemented a solution where the distance between the diagonals are set in the shader source code, and the branching is done by the GLSL preprocessor. This move yielded approximately a five time speed-up compared to the other solution, but it does not apply to random-sparse matrices. The diagonals where this solution is applied are the diagonals closest to the middle diagonal, and they are stored first in the sub-boxes in the texture.

6.3.3 The overall algorithm

The algorithm is identical to that shown in the algorithm deduced in the background material section this chapter. The algorithm also includes an extra shader, which really is a reduced edition of the scale-vector shader, to initialize the calculations.

6.4 Benchmarking

The Poisson equation is used to test the efficiency of the algorithm. The equation is important in physics for simulation of fluids. The GPU-implementation followed my C++ reference implementation in speed carefully. I got the exact same timings. The lack of variations is a little bit strange, but I double checked that it is not the same implementation that was executed twice. Unfortunately, I think there is a bug that I have not discovered earlier, and therefore I choose not to relaese any tables with timings for this implementaion, because the results will most probably wrong.

6.5 Future extensions

A natural extension would be to continue with preconditioning in the GPU-algorithm. One solution could maybe be to use a method that does not require readback in every iteration, and use it as a preconditionar for CG on the CPU. Another thing is to continue to work with mixed precision. I have started with that, but for some reason I never got the answer to converge when I sent it back to the GPU, after it had been processed on the CPU for one iteration.

Chapter 7

Conclusions and future work

Results: In this thesis I have investigated functionality in the GPU, for its usage as a co-processor for accelerating numerical linear algebra. A selection of algorithms has been ported to execute on the GPU, including the LU-algorithm with partial pivoting and the conjugate gradient method. From the results from for example the LU- algorithm, we can see that we beat the highly tuned ATLAS implementation on the CPU, thus if we split the computational domain, and run parts of it on the CPU and parts on the GPU, we can be able to increase the speed of computations dramatically with little added cost. On the CPU years have been spent on altering linear algebra algorithms for optimal cache re-usage, but the data-parallel graphics processor performs likewise with the standard vectorized version of the LU-algorithm, even though there is some idling on both the GPU and the CPU when the column where the pivot should be searched for is read back, and the upper triangular rows has to be copied to its final location. On some new architectures, like the Playstation 3, the graphics memory can be directly accessed from the main processor, thus the read-back is less of a bottle-neck.

Contributions: There are two main contributions in this thesis. The LU-implementation includes a special pivoting strategy where the pivot row is rendered to its correct location during update of the sub-matrix. The other main contribution can be found in the conjugate gradient implementation. Gathering of the inner-product calculations on the GPU allows for simultaneous accumulation and readback of both inner products and that results in a reduction of about halve the cost of communication and accumulation. Although this strategy is old, it has not been applied on GPUs before.

Further research on the topics discussed in this thesis: In addition to porting other linear algebra algorithms to the GPU, there is still room for improvements in both implementations, for example a closer examination of pre-conditioning in the conjugate gradient method. More importantly new libraries that can access the GPU directly from outside graphics APIs, including CUDA, have been publicly available during the last period of this master thesis, and such APIs should be closer inspected. Especially CUDA, that includes a BLAS library.

Application in the future: We can generally see an increasing trend of multi-core processors and the data-parallel programming model utilized through this thesis provides a simple interface to the underlying parallel architecture. To utilize massively parallel processors in the future, there is a possibility that the data-parallel programming model will have to be used and the GPU is about as close as we can come to a massively parallel architecture today. The interface to the parallel architecture will probably change a lot over the next years, and the futuristic architectures will probably have more cache and be more flexible, but the same rules for performance as I have worked with in this thesis will most probably apply.

Bibliography

[APC04] J. Dongarra A. Petitet, R. C. Whaley and A. ClearyNetlib. Hpl algorithm, 2004. [Online; accessed 25-April-2007].

[BP04] Ian Buck and Tim Purcell. A toolkit for computation on gpus. In Randima Fernando, editor, *GPU Gems*, chapter 37, pages 621–636. Addison Wesley, 2004.

[Buc05] Ian Buck. Taking the plunge into gpu computing. In Matt Pharr, editor, *GPU Gems 2*, chapter 32, pages 509–545. Addison Wesley, March 2005.

[Elh05] Ikrima Elhassan. Fast texture downloads and readbacks using pixel buffer objects in opengl. Nvidia technical brief, 2701 San Tomas Expressway Santa Clara, CA 95050, August 2005.

[GGHM05] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[HB05] Mark Harris and Ian Buck. Gpu flow-control idioms. In Matt Pharr, editor, *GPU Gems 2*, chapter 34, pages 547–555. Addison Wesley, March 2005.

[HHL+05] Trond Runar Hagen, J. M. Hjelmervik, Knut-Andreas Lie, Jostein R. Natvig, and M. Ofstad Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory*, 13(8):716–726, 2005.

[Hig96] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[Pea06] Peakstream. High performance modelling of derivative prices prices using the peakstream platform, 2006. [Online; accessed 25-April-2007].

[PPM06] *Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment*, The University of Electro-Communications 1-5-1, Chofugaoka, Chofu-shi, Tokyo, Japan, 2006.

[Wik07a] Wikipedia. Blackscholes — wikipedia, the free encyclopedia, 2007. [Online; accessed 25-April-2007].

[Wik07b] Wikipedia. Parallel algorithm — wikipedia, the free encyclopedia, 2007. [Online; accessed 4-April-2007].

[Wik07c] Wikipedia. Parallel computing — wikipedia, the free encyclopedia, 2007. [Online; accessed 4-April-2007].

List of Figures

List of Tables

Listings

PLU Factorization on a Cluster of GPUs Using Fast Ethernet

André Rigland Brodtkorb, Martin Lilleng Sætra and Trygve Fladby

1st May 2007

Abstract In this white paper, we present a novel approach to solve linear systems of equations on a cluster using the PLU factorization. We use the graphics processing unit (GPU) as the main computational engine at each node, and a block-cyclic data distribution to solve the system. The local computation is a new way of solving the PLU factorization on the GPU. It utilizes the full four-way vectorized arithmetic found in most GPUs, and a new pivoting strategy. The global algorithm uses the message passing interface (MPI) for communication between nodes. We show that our algorithm is highly efficient on the local nodes, but bounded by the relatively slow network. A faster network will eliminate this bottleneck, and the speed of the local computations show promising results.

1 Introduction

This paper explores the field of general purpose computation on graphics processing units (GPGPU). We specifically target the PLU factorization of a large system of linear equations on a cluster of nodes. Solving large linear systems of equations using dense algorithms is used extensively as a benchmark for clusters and supercomputers. The High Performance LINPACK benchmark (HPL) [1] which computes the PLU factorization, is the standard way of benchmarking and ranking the fastest 500 supercomputers in the world [2]. This benchmark, however, has been criticized for neglecting the importance of faster inter-node communication. This is because the HPL benchmark can run the benchmark with different parameters that compensate for slow network communication by letting each node execute extra computations (e.g., look-ahead).

While the HPL benchmark uses the CPU to compute partial results on each node, we utilize the graphics processing unit (GPU) as the main computational engine to solve the same problem. The GPU is a massively parallel processor with vast amounts of processing power [3]. Current GPUs have a theoretical peak of 400 GFLOPS [4], compared to 90 GFLOPS [4] for current high-end CPUs. When comparing the price[1] per FLOP, the GPU comes out ahead as well with approximately $1.50 per GFLOP, compared to the CPU that costs approximately $18 per GFLOP.

During the last years, we have seen an enormous development in 3D-graphics. The demand for more powerful programmable graphics processing units (GPU) from for example the gaming industry has led to increased flexibility in the processors. The rapid evolution in speed and flexibility has made the GPU interesting for scientific purposes as well. The field of general-purpose computation on GPUs (GPGPU) has emerged as a new and exiting research area [3]. Even though the GPU is a far more powerful and cost-effective processor than the CPU, there is another price. While the CPU has complex logic for branch prediction, cache management, and instruction pipelining, most of the transistors on the GPU are used for pure floating-point operations. There is another architectural difference as well. The CPU is designed to operate on sequential code, such as word processing where each character is entered and processed sequentially. The GPU on the other hand, is designed to simultaneously compute all the pixels that together make up the screen image. In addition, the GPU could traditionally only be accessed via a graphics API, such as OpenGL [5] or DirectX [6]. The architectural differences, and the need to access the GPU through a graphics API require new algorithms and techniques to be employed when the GPU is to be used for general-purpose computing.

2 Background

The Top 500 project [2] was started in 1993 to provide a reliable basis for tracking and detecting trends in the field of high-performance computing. It is a list of the 500 most powerful supercomputers, which is updated twice per year. The ranking of the supercomputer sites is determined by how well they perform on the LINPACK benchmark. A parallel version of LINPACK named HPL [1] was introduced

[1] Prices are from the Norwegian web shop komplett.no 2007-04-23.

Listing 1: Example on a deadlock in an MPI-2 program

```
MPI_Init(&argc, &argv);

if(processId == 0) {
  MPI_Recv(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
  MPI_Send(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD);
} else(processId == 1) {
  MPI_Recv(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
  MPI_Send(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD);
}

MPI_Finalize();
```

by Dongarra, for this purpose. HPL is short for High-Performance LINPACK Benchmark for Distributed-Memory Computers. HPL utilizes the Message Passing Interface (MPI) and the Basic Linear Algebra Subprograms (BLAS). The algorithm used by HPL implements a two-dimensional block-cyclic data distribution. In addition a look-ahead strategy and bandwidth reducing swap-broadcast algorithm is used to increase performance. The complete operation count sums up to $\mathcal{O}(\frac{2}{3}n^3) + \mathcal{O}(n^2)$.

LU factorization on the GPU has previously been implemented by Galoppo et al. [7]. One of their main contributions was index-pair streaming, which uses texture coordinates to make a cache-oblivious algorithm. The index-pair streaming technique sets texture coordinates from the CPU in order for the GPU to pre-fetch data, in contrast to computing them on the fly on the GPU. This data pre-fetch resulted in about 25% speed increase [7]. They also reported their algorithm as faster than ATLAS, but the benchmark was highly synthetic.

To run our application in parallel on multiple nodes, we have utilized the Message Passing Interface 2.0 (MPI-2) [8]. MPI-2 is a C/C++ and Fortran interface for message passing between multiple processes spread over any number of nodes. It can be used in many different setups, e.g., supercomputers, distributed memory clusters, and shared memory clusters. Several implementations of MPI-2 exist, where we have chosen MPICH2 [9] for our application. The most important uses of MPI-2 in our application are the automatic generation of a block-cyclic Cartesian grid of processes and broadcast of data to groups of processes.

There are two concepts related to our use of MPI-2 that require some explanation; communicators, and blocking- and non-blocking calls. A *communicator* in MPI is a collection of processes. Many functions in MPI-2 take a communicator as argument and perform the requested operation on all processes in that communicator. A call to the broadcast function in MPI, for example, can look like this: `MPI_Bcast(buf, 10, MPI_FLOAT, 0, MPI_COMM_WORLD)`. This call will broadcast ten elements of the array `buf` to all processes in the `MPI_COMM_WORLD` communicator. The other processes in the communicator must also call the `MPI_Bcast` function to receive these elements. The `MPI_COMM_WORLD` communicator is a special communicator that contains all processes, and it is initialized automatically by MPI. When an MPI function is called on all processes within a communicator (or group) it is referred to as a collective operation. `MPI_Bcast` is a collective operation.

A *blocking* call will make the application wait for the call to complete before continuing execution. In this way you will know if the call has finished successfully or aborted due to some error. This also means that the application may get deadlocked, where two or more processes have called competing blocking functions that are circularly dependent on each other [10]. For example, if we have two processes that execute the code in Listing 1, it will result in a deadlock. Both processes are waiting for the other to send data, thus blocking program execution. A non-blocking call on the other hand, will not cause the application to wait for the call to return. In this way it is possible to call a function and continue executing the application before the function returns. Collective operations in MPI-2, however, are always blocking.

3 Algorithm

The LU factorization of a matrix A can be written as $LU = A$, where L and U are *lower* and *upper* triangular respectively. Using the Doolittle algorithm, we can construct the upper triangular matrix U using Gaussian elimination. The lower triangular matrix is constructed from the multipliers used to

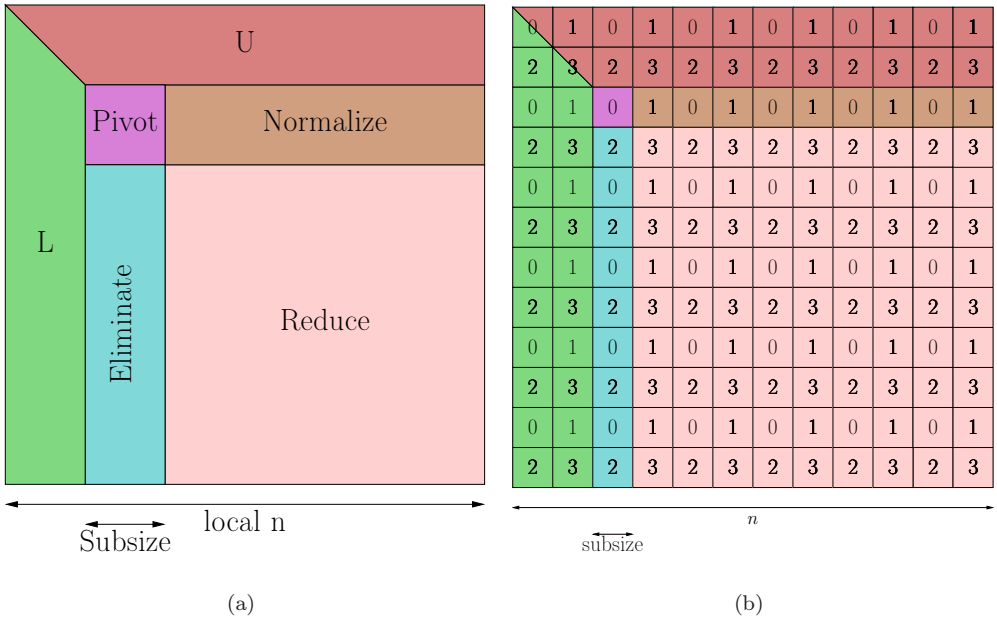


Figure 1: PLU decomposition on a cluster of nodes: (a) the four different parts of the LU factorization. (b) the block-cyclic distribution of data on four nodes, 0, 1, 2 and 3.

reduce A to an upper triangular form. For our algorithm to be numerically stable, we also permute the rows of A. This is known as partial pivoting, and ensures that the row we are eliminating with creates smaller perturbations of the result than would normally occur. With the permutation of the rows in A, our factorization takes the form $A = P^T LU$, where P is the permutation matrix that permutes rows of A.

Our algorithm has two layers, the global and the local computation. The global algorithm solves the PLU factorization of the matrix spread over all the nodes, shown in Figure 1(b), whilst the local algorithm is what each node needs to compute for the global algorithm to be correct.

Each node in the computation receives a block-cyclic part of the matrix, as shown in Figure 1(b). Then, all the processors compute what type of operation they need to compute. Our algorithm splits the computation into four distinct operations: pivot, normalize, eliminate and reduce, as shown in Figure 1(a). The operation computed on each node depends on the global position of the pivot operation. All processors that hold elements in the same row as the pivot operation need to compute the normalize operation, and similarly all nodes with elements in the same column as the pivot operation need to compute the eliminate operation. All remaining nodes need to compute the reduction operation. In Figure 1(b) this means that process 0 is the *pivot*, process 1 executes *normalize*, process 2 *eliminate*, and process 3 *reduce*. The pivot node shifts one down along the diagonal for each global pass.

3.1 Global algorithm

Computing the PLU factorization is an almost embarrassingly parallel operation. However, vanilla implementations demand a lot of data to be transferred between nodes, which is a very costly operation. In addition, many nodes would simply idle as we reach the end of the computation.

To reduce the idling, we distribute the matrix A block cyclically in the same fashion as the HPL algorithm [1]. Figure 1(b) shows this distribution, where all nodes have a part of the matrix to process throughout the whole factorization, except for the very last block. The last block is computed by the last node in an extra pass. For each pass in the global domain, we compute the result of one row of blocks, and one column of blocks. In the following, we refer to these as *block-row* and *block-column* respectively.

To lessen the amount and number of transfers between nodes, we use partial pivoting within in-core memory, thus eliminating the need to transfer rows between processors. It is trivial to create examples

where partial pivoting fails, but sufficient accuracy is attainable in practice. This also holds for our pivoting, which pivots in a subset of the regular pivot candidates.

In order to compute one pass in the global domain, we have to execute the four different operations *pivot, normalize, eliminate* and *reduce*. It should be mentioned that this data distribution, and splitting into different operations per node allows for multiple nodes, not only four as shown in this example. In the third pass of this algorithm, we have the following situation (see also Figure 2):

Pivot: The pivot position (process 0) must compute the PLU factorization of the current active pivot block in its local domain. The block size is subsize × subsize. In addition, it has to reduce the rest of the local matrix according to the computed L and U. These blocks belong elsewhere in the global domain (see Figure 1(b)). In each global pass, there is always only one pivot node.

Normalize: The normalize operation (process 1) needs to compute U according to the P and L computed by the *pivot* operation. It will also have to reduce all remaining elements in the local matrix, which again belong elsewhere in the global domain. There are $s - 1$ nodes that compute the normalize operation in each global pass, where s is the width and height of the processor grid.

Eliminate: Eliminate (process 2) calculates the multipliers needed to forward substitute one block by using the computed U's from *pivot*. In addition, it has to reduce the rest of the local matrix, according to the computed U. In each global pass, the number of eliminate nodes is also $s - 1$.

Reduce: The reduce operation simply reduces the local matrix according to the L and U computed in *eliminate* and *normalize* respectively. All remaining processes compute this operation, $s \times s - 2(s-1) - 1$ nodes.

As stated in the list of operations, the different processes depend on data from other processes. This dependency is not static, but varies with the operation the current node is set to execute. Figure 2 shows how the data is sent in the already used example. The nodes waiting for data cannot continue before they have received the data. This effectively limits the computational speed to the slowest node. The HPL [1] algorithm uses look-ahead to remedy this somewhat. As this chart shows, there is still quite a lot of idling for the four nodes. The pivot node, for example, computes its result and then waits until all other nodes have completed their computations.

3.2 Local algorithm

The local algorithm includes four stages *pivot, eliminate, normalize* and *reduce*, but first we will introduce the matrix representation. The data is row-wise represented in four-wide vectors [11]. This is to utilize as much computational power and bandwidth as possible, since most GPUs can execute one MAD instruction on four-long vectors per clock cycle. The advantage of this packing scheme is that it does not require restructuring of the data in main memory before it is sent to the GPU[2]. Another reason for this choice is that it fits well with the solution we have for pivoting. In addition to storing the matrix, we add an extra column leftmost in the matrix, as shown in in Figure 3(a). This column is used to speed up the calculation of the next pivot element, explained later. Because the result of writing to the same buffer as we read from is explicitly undefined in OpenGL, we have to use an extra texture. The two textures are used as one virtual matrix, but we alternate between reading / writing and writing / reading to the front and back textures, respectively. This technique is referred to as ping-ponging in the field of GPGPU.

3.2.1 Pivot

The pivot procedure computes the PLU factorization of A, but stops when one block-row and one block-column has been computed (see Figure 1(b)). It can roughly be split into two tasks: multiplier calculation, and reduction, each explained below. To compute a single row and column, we start by permuting the first column simultaneously as we compute the multipliers. Then, we reduce the rest of the matrix, whilst permuting the rows here as well.

To compute one column of multipliers, we read from the correct location in the source texture, and write to the leftmost column in the destination, as shown in Figure 3(a). The top element is rendered at

[2] Assuming its width is divisible by four.

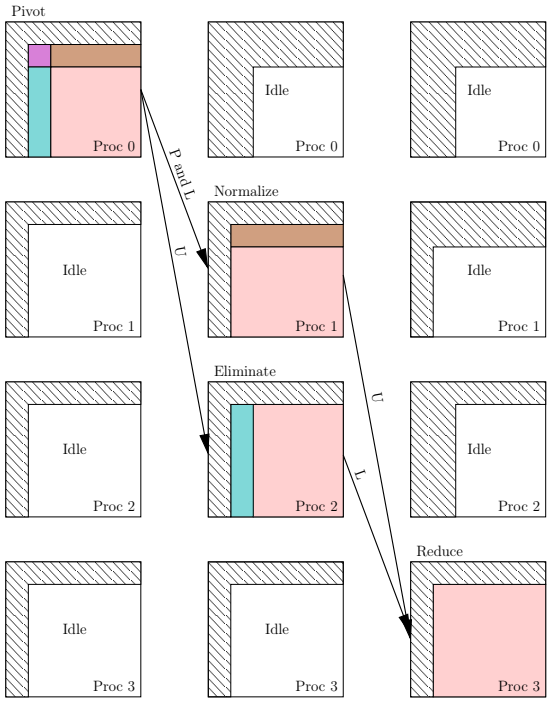


Figure 2: Data send patterns for PLU decomposition using four nodes in the third global pass (corresponds to the situation in Figure 1(b)). The shaded areas represent the part of the matrix we already have computed.

the position of the pivot element. Because the multiplier for the top row always is one, we do not need to compute it. In addition to computing the multipliers, we also compute the values of the column to the right of the pivot position and store in one of the other color channels (see Figure 3(b)).

When the computation is complete, we transfer the multipliers and the reduced next column to the CPU using a pixel buffer object (PBO). The PBO uses asynchronous read-back to the CPU, allowing both the CPU and the GPU to continue execution. When the whole leftmost column has been transferred to the CPU, the next pivot element is found by the CPU. Simultaneously as the data is copied, and the CPU searches for the pivot element, the GPU subtracts the multiplier times the top row throughout the rest of the matrix. The top and pivot row are also interchanged simultaneously in the same manner as in the first column. In addition, we employ the index pair streaming technique to increase performance [7]. When the computation is complete, the top row is copied to the CPU, again using a PBO. The algorithm continues until we have computed the whole block-row of U, and block-column of L.

3.2.2 Normalize

The normalize step computed on the local domain executes as follows: The L matrix from this global time-step's pivot node is uploaded to the GPU as a texture. Then, we execute a for-loop that sequentially computes one row of U at a time. First, the current top row and pivot row are swapped, simultaneously as we eliminate using the multipliers in L. Because we are using two buffers, we read back the pivot row simultaneously using PBO's, and store them in main memory. When all rows in the block-row have been computed, U is sent to all nodes in the same column for the reduction operation.

3.2.3 Eliminate

The elimination procedure calculates multipliers. Normalized rows (U) are sent from the current time-step's pivot node, and the multipliers are calculated using these. The elimination step follows much of the same procedure as the pivot step, but it is a simpler case since there is no complications with row interchanges. This is again because the pivot node only pivots within in-core memory.

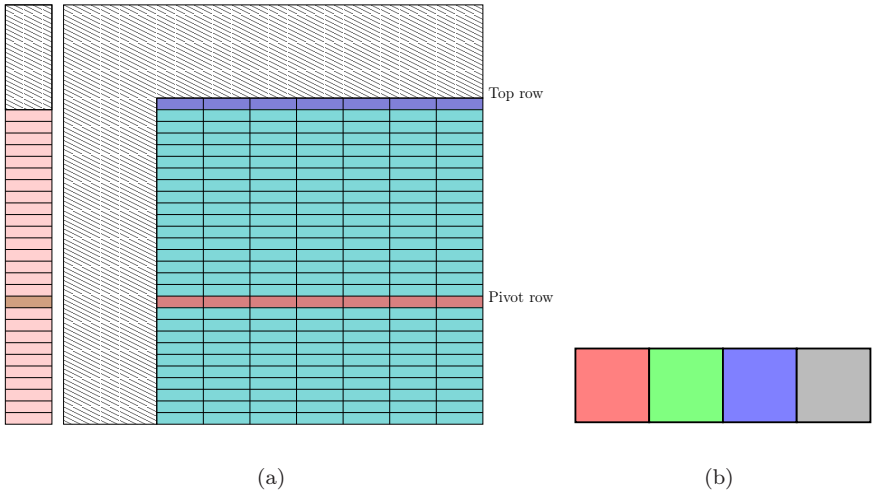


Figure 3: Data representation on the GPU: (a) Row interchange of the multipliers (leftmost column) and the rest of the matrix (cyan part). (b) The leftmost column of the texture, with both the multiplier, and the reduced next column in the PLU factorization. The multiplier is stored in the red color channel, and the reduced next column is stored in the blue color channel.

Listing 2: Setting up row- and column-communicators

```
/* Set up row communicators */
MPI_Cart_sub(origcom, {0, 1}, &rowcom);

/* Set up column communicators */
MPI_Cart_sub(origcom, {1, 0}, &colcom);
```

3.2.4 Reduce

The reduction step is trivial on the local node. Using a for-loop, we sequentially reduce the whole remaining sub-matrix by looking up one row from U and one column from L, and calculating the reduced A as $A_{i,j} := A_{i,j} - L_{i,k} \cdot U_{k,j}$.

3.2.5 Sending of data

This section describes how data is sent between different nodes. The use of MPI-2 for this inter-node communication will also be explained in detail.

Based on the algorithm discussed in Section 3.1 we have the following communication scenarios:

1. Sending data to all processes in the same row as active process (to *normalize* and *reduce*).
2. Sending data to all processes in the same column as active process (to *eliminate* and *reduce*).

For broadcasting data to all processes in the same row as the active process, the broadcast function in MPI, `MPI_Bcast`, is used. This function takes a communicator, a pointer to the data, and a count of data elements as arguments. When called, it broadcasts the data to all processes within that communicator. Broadcasting data to the same row as yourself is done by calling `MPI_Bcast` with the row communicator.

To broadcast to columns we use the column communicator instead of the row communicator.

Since the `MPI_Bcast` function is collective, it needs to be called in every process within the current communicator. This implies that each process needs to know a priori from which node it will receive the next broadcast. In our application we have a function dedicated to calculate this. This function bases the calculation on which global pass the process is currently in, and which type it currently is (*pivot, normalize, eliminate* or *reduce*). This method is fairly complicated, but can be briefly explained as follows: The *normalize* nodes will always receive a broadcast from the *pivot* node, which is the diagonal

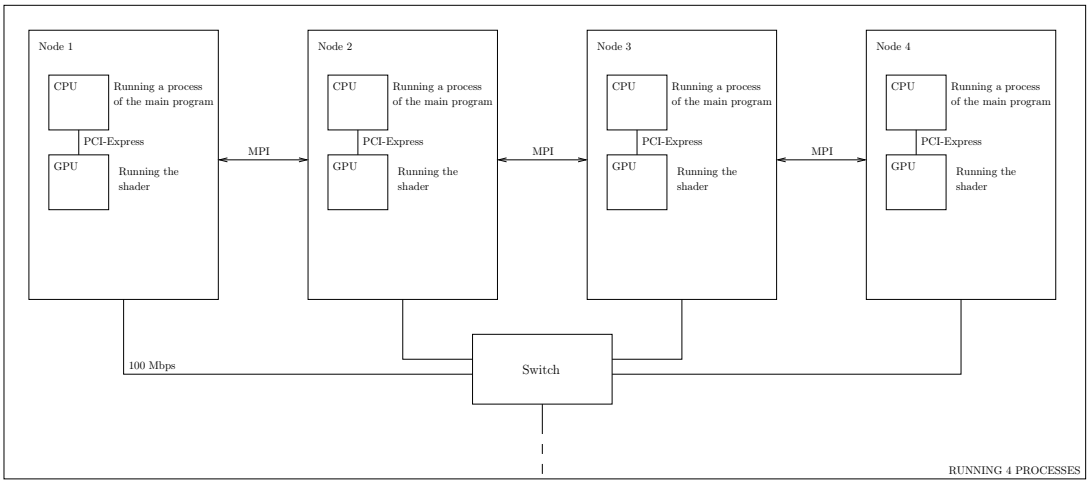


Figure 4: Overview of physical setup of nodes.

element in its row communicator. *Eliminate* is similar, but will receive from the diagonal element in its column communicator. Finally, *reduce* will receive data from *normalize*, which is the node with the same column index as the current node, and the same row index as the current *pivot* node. *Reduce* also receives data from *eliminate*, which is computed in a similar fashion.

To facilitate the communication needed by our algorithm, row- and column-wise communicators are set up. Listing 2 shows the code used to create these communicators. In this listing, the array sent as the second parameter sets which dimension we wish to keep in the new communicators. When we create the row communicators we keep the y-dimension intact, and when creating the column communicators we do the opposite and keep the x-dimension. When the code is executed, each process will set up a row communicator called `rowcom` and a column communicator called `colcom` relative to the process' location in the grid.

4 Results

The cluster which we benchmarked our application on consists of four one-CPU, one-GPU nodes as shown in Figure 4. The nodes were all equipped were Intel Pentium 4 processors with Hyper-Threading Technology (HTT) and 2 GB of RAM. All nodes had an NVIDIA GeForce 7800 GT graphics adapter on a PCI-Express 16× slot.

4.1 Benchmark

Benchmarking of our algorithms showed that it gives sufficiently accurate results considering that all computation is executed on single precision hardware.

When benchmarking the algorithm, we have varied several variables to identify possible bottlenecks. The variables we have varied are:

1. Number of nodes.
2. Number of processes.
3. The size of the block to factorize in each global pass (subsize).
4. The total size of the problem matrix (n).

In addition, we have benchmarked the pivot operation on a single node executed on the full matrix, as well as only network communication. This gives us performance results for our network setup, the local algorithm, as well as the global algorithm, enabling analysis of the limiting factor.

Table 1: Variation of the subsize parameter, as well as the impact of several nodes. The number of processes is 4, and the times are in seconds.

-		Nodes			
n	Subsize	1	2	3	4
128	8	0,20607	0,14006	0,13756	0,28482
	16	0,23247	0,11918	0,14593	0,28739
	32	0,19208	0,10213	0,11030	0,27609
	64	0,13572	0,09238	0,08232	0,24506
512	32	0,54457	0,25811	0,28454	1,11726
	64	0,49388	0,24161	0,26360	0,78500
	128	0,32307	0,23648	0,24194	0,64518
	256	0,24012	0,20242	0,21688	0,36138
2048	128	3,17257	2,43952	2,95311	3,19620
	256	3,07729	2,43028	2,95513	3,15248
	512	2,88925	2,41467	2,87859	3,05907
	1024	2,59612	2,39955	2,68849	2,93310
4096	256	13,76410	13,03520	14,77890	15,26550
	512	13,70820	13,18710	14,74090	15,78910
	1024	13,59550	13,59640	14,73430	16,26440
	2048	14,62520	14,45370	14,50600	16,66760

Table 1 shows the time used to compute the PLU factorization while varying the number of nodes, size of the matrix, and the block size. The maximum achieved performance is 3.5 GFLOPS (for $n = 4096$ on two nodes), and the general trend seems to suggest that using only two nodes is faster than using four. This can somewhat be explained by interprocess communication being faster with two processes per node, than one process per node, as this eliminates a lot of network communication.

Table 2 shows the time used to compute the PLU factorization while varying the number of processes on four nodes. As the table shows, the speed of the algorithm can be greatly influenced by tuning this parameter. However, the optimal number of processes seems to vary with the size of the matrix. The maximum achieved performance achieved was now increased to 4.2 GFLOPS (16 processes on four nodes). We also timed the network-communication, and measured the percentage of the total time used for network communication. The percentages show that there is a substantial time used to send and receive data alone.

To analyze the impact of the network, we ran the network communication while varying the number of nodes. Table 3 shows the time of the network communication, and the impact of the subsize parameter,

Table 2: Variation of the number of processes. The number of nodes is four, and the times are in seconds.

Procs	Subsize	Time	Network time %
4	256	97,78950	42
	512	98,19350	36
	1024	99,65560	31
	2048	102,98800	30
16	256	86,30310	37
	512	88,69330	35
	1024	89,53110	35
	2048	95,1656	33
64	128	122,72900	24
	256	124,48000	23
	512	120,79000	23
	1024	124,32000	21

Table 3: The time spent transmitting data. The number of processes is four and the problem size is 2048, while the number of nodes is varied. This shows the impact of the network communication.

-	Nodes			
Subsize	1	2	3	4
128	0,57228	3,18597	5,70082	6,30781
256	0,59500	3,14385	5,72201	5,73072
512	0,62175	3,13140	5,64543	5,65009
1024	0,69741	3,02938	5,37086	5,38025

Table 4: The time spent computing using only a single node where subsize = n. The times are in seconds.

n	Time
64	0,0284489
256	0,0491337
1024	0,280545
2048	1,44955
4096	10,051

as well as the use of multiple nodes. The subsize parameter seems to have little effect on the time, whilst the number of nodes has a massive impact. Using two nodes with four processes is approximately half as expensive as using four nodes.

Finally, we have benchmarked the pivot operation on one node. This is the most computationally heavy operation, and a limiting factor. Table 4 shows the time spent to compute a full matrix using the pivot operation. The peak performance was measured for the largest matrix, 4096×4096, where the algorithm performed 4.6 GFLOPS. As a comparison, we timed the ATLAS implementation used in MATLAB, which achieved 3.5 GFLOPS on the same problem size.

4.2 Analysis

Our global algorithm had a maximum measured performance of 4.2 GFLOPS using four nodes, while our local algorithm showed a promising 4.6 GFLOPS. The network communication could account for at least 20% of the total runtime. However, because of the way the presented algorithm is executed, most of the processes simply idle, waiting for data. This is the largest bottleneck, but there are some solutions.

Using a look-ahead strategy, as used in the HPL [1] algorithm, will increase the workload per node, and decrease the idling. In addition, restructuring the computation into smaller parts, so that pivot, eliminate, normalize and reduce are split into smaller subproblems, will also decrease the time spent idling per node.

We have not been able to show the full potential of this algorithm, because we have only have had four nodes at disposal. Having only four nodes makes almost all the computation execute serially, because we only have one node per operation at each global time-step. This parallelizes the computation of normalize and eliminate only. Using more nodes, will parallelize the reduction step of the algorithm as well, and probably speed up the total computational speed.

5 Conclusions and further research

We have presented a new way of computing the PLU factorization of a matrix, by using the GPU on a cluster of nodes. We have shown that the algorithms computed locally are efficient, even outperforming ATLAS. Our global algorithm, however, is less efficient. We have pointed to a slow network link, a lot of idling of nodes, and the use of only four nodes as the main reasons.

A faster network link will decrease the impact of the network communication in our algorithm. It is also possible to lessen the issue with idling of nodes by using techniques such as look-ahead, or splitting up the computation further.

It is possible to extend our algorithm to include forward and backward substitution, as the HPL algorithm does. The computation of the forward substitution will be virtually free, while the backward substitution will require more global passes. Including the forward and backward substitution in the algorithm will fulfill the complexity demands for the Top500 benchmark [2].

6 Acknowledgements

We would like to thank J. Hjelmervik for first proposing this project to us, and our supervisors K.-A. Lie and T. R. Hagen for their helpful guiding and notes on our white paper. We would also like to thank G. W. Ma for all his assistance, and our fellow master students at SINTEF ICT for insightful discussions.

References

[1] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl. http://www.netlib.org/benchmark/hpl/. [accessed 2007-04-18].

[2] University of Mannheim, University of Tennessee, and NERSC/LBNL. Top 500 supercomputer sites. http://www.top500.org/about. [accessed 2007-04-18].

[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[4] Neoptica. Programmable graphics – the future of interactive rendering. Online; http://www.neoptica.com/NeopticaWhitepaper.pdf, 2007. [accessed 2007-04-23].

[5] Khronos Group. OpenGL – the industry's foundation for high performance graphics. Online; http://www.opengl.org, 2007. [accessed 2007-04-25].

[6] Microsoft Corporation. Microsoft DirectX. Online; http://www.microsoft.com/directx, 2007. [accessed 2007-04-25].

[7] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[8] MPI Forum. Mpi documents. Online; http://www.mpi-forum.org/docs/docs.html. [accessed 2007-04-18].

[9] Argonne National Laboratory. Mpich2. Online; http://www-unix.mcs.anl.gov/mpi/mpich2/. [accessed 2007-04-18].

[10] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[11] A. Moravánszky. Dense matrix algebra on the GPU. Online; http://www.shaderx2.com/shaderx.pdf, 2003. [accessed 2006-05-11].