caption

Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Comparison of platforms for General-purpose computing on graphics processing units

Examensarbete utfört i Mindroad AB
vid Tekniska högskolan vid Linköpings universitet
av

Torbjörn Sörman

LiTH-ISY-EX--YY/NNNN--SE

Linköping 2015

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Comparison of platforms for General-purpose computing on graphics processing units

Examensarbete utfört i Mindroad AB
vid Tekniska högskolan vid Linköpings universitet
av

Torbjörn Sörman

LiTH-ISY-EX--YY/NNNN--SE

Handledare: **Robert Forchheimer**
ISY, Linköpings universitet
Åsa Detterfelt
Mindroad AB

Examinator: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Linköping, 24 december 2015

Avdelning, Institution
Division, Department

Organisatorisk avdelning
Department of Electrical Engineering
SE-581 83 Linköping

Datum
Date

2015-12-24

Språk
Language

□ Svenska/Swedish
☒ Engelska/English
□ _____

Rapporttyp
Report category

□ Licentiatavhandling
☒ Examensarbete
□ C-uppsats
□ D-uppsats
□ Övrig rapport
□ _____

ISBN
__

ISRN
LiTH-ISY-EX--YY/NNNN--SE

Serietitel och serienummer
Title of series, numbering

ISSN
__

URL för elektronisk version
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-XXXXX

Titel
Title

Svensk titel
Comparison of platforms for General-purpose computing on graphics processing units

Författare
Author

Torbjörn Sörman

Sammanfattning
Abstract

If your thesis is written in English, the primary abstract would go here while the Swedish abstract would be optional.

Nyckelord
Keywords

problem, lösning

Abstract

If your thesis is written in English, the primary abstract would go here while the Swedish abstract would be optional.

Acknowledgments

I would like to thank Åsa for the opportunity to make this Thesis work at MindRoad AB. I would also like to thank Ingemar and Robert at ISY.

Linköping, December 2015
Torbjörn Sörman

Contents

Notation

Några mängder

Notation	Betydelse
$\mathbb{N}$	Mängden av naturliga tal
$\mathbb{R}$	Mängden av reella tal
$\mathbb{C}$	Mängden av komplexa tal

Förkortningar

Förkortning	Betydelse
ARMA	Auto-regressive moving average
PID	Proportional, integral, differential (regulator)

1

Introduction

This chapter gives an introduction to the thesis. It describes the background, purpose and goal of the thesis, and also a list of abbreviations and the structure of this report.

1.1 Background

The computationally demanding problems have during a long period of time been solved faster by technical improvements in hardware. However, some limitations have been reached the last decades. Operating frequency of the CPU is no longer significantly improved. Problems relying on single thread performance are limited by three primary technical factors:

1. The Instruction-Level Parallelism (ILP) wall
2. The memory wall
3. The power wall

The first wall states that its hard to further exploit simultanious CPU instructions, techniques like instruction pipelining, superscalar execution and VLIW exists but complexity and latency of hardware reduces the benefits. Related to the first is second wall, the gap between CPU speed and memory access time, that may cost several hundreds of CPU cykles if accessing primary memory. The third wall is power and heating problem. The power consumed is increased exponentially with each factorial increase of operating frequency.

Improvements can be found in exploiting parallelism. Either reconstruct the problem or the problem itself is already inherently parallelizable. This trend

manifests in development towards use and construction of multi-core microprocessors. The graphical processing unit (GPU) is one such device, originally exploited the inherent parallelism within visual rendering but now is available as a tool for massively parallelizable problems.

1.1.1 Problem statement

Programmers might experience a threshold and slow learning curve to move from sequential programming to thread-parallel programming that is GPU programming. Obstacles involve learning about the hardware architecture and restructure the algorithm or solution. Knowing the limitations and benefits might even provide evidence of not utilizing the GPU and instead choose to work with a multi-core CPU.

Depending on one's preferences, needs and future goals; selecting one platform over the other might be derived from portability needs, hardware requirements, programmability, how well it integrates with other platforms or how well it's supported by the provider or the developer community. Within the range of this thesis, the covered platforms or frameworks are CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language), DirectCompute and OpenGL Compute Shaders.

1.1.2 Purpose and goal of the thesis work

One goal is to evaluate, select and implement an algorithm suitable for GPGPU (General-purpose computing on graphics processing units). Implement the same algorithm in important frameworks for GPGPU:

- CUDA
- OpenCL
- DirectCompute (DirectX Compute Shaders)
- OpenGL Compute Shaders

The purpose is to compare the different APIs/frameworks by means of benchmarking performance and make qualitative assessments.

1.2 Algorithm

This part cover some choices of algorithm for a GPGPU study. The basic theory and motivation why they are suitable for benchmarking GPGPU platforms is presented. For this thesis, the Fast Fourier Transform is selected and will be more detailed in another part of the report.

1.2.1 Discrete Fourier Transform

The Fourier transform is of use when analyzing the spectrum of a continuous analog signal. When applying transformation to a signal it is decomposed into the frequencies that makes it up. In digital signal analysis the Discrete Fourier transform (DFT) is the counterpart of the Fourier transform for analog signals. The DFT converts a sequence of finite length into a list of coefficients of a finite combination of complex sinusoids. Given that the sequence is a sampled function from the time or spatial domain it's a conversion to the frequency domain. It is defined as

$$X_k = \sum_{n=0}^{N-1} x(n) W_N^{kn}, k \in [0, N-1] \tag{1.1}$$

where $W_N^{kn} = e^{-2iknN}$, commonly named the twiddle factor[4].

The DFT is used in many practical applications to perform Fourier analysis. In digital signal processing, such as discrete samples of sound waves, radio signal or any continuous signal over a finite time interval. In image processing the sampled sequence can be pixels along a row or column. The DFT takes input in complex numbers and outputs in complex coefficients. In practical applications the input is usually real numbers.

1.2.2 Fast Fourier Transform

The problem with the DFT is that it has by definition a time complexity of $\mathcal{O}(n^n)$ that makes it too slow for some applications. The Fast Fourier Transform (FFT) is one of the most common algorithm used to compute the DFT of a sequence. An FFT computes the transformation by factorizing the transformation matrix of the DFT into a product of mostly zero factors. This reduces the time complexity to $\mathcal{O}(n \log n)$. The FFT was made popular in 1965 by J.W Cooley and J.W. Tukey and it found its way into practical use at the same time and meant a serious breakthrough in digital signal processing [3, 1]. However the complete algorithm was not invented at the time, the history of the Cooley-Tukey FFT algorithm can be traced back to around 1805 by work of the famous mathematician Carl Friedrich Gauss[5].

The algorithm is a divide and conquer algorithm that relies on recursively dividing the input into sub-blocks and eventually the problem is small enough to be solved and the sub-blocks are combined into the final result.

1.2.3 Image processing

Image processing consists of a wide range of domains. Earlier academic work with performance evaluation on the GPU[7] tested four major domains (3D shape reconstruction, feature extraction, image compression and computational photography) and compared with the CPU. Generally image processing is by nature parallel and one can expect good results on a GPU.

Most of image processing algorithms apply the same computation on a number of pixels and that is a typically data parallel operation. Some algorithms can

then be expected to have huge speed up compared to an efficient CPU implementation. A representative task is applying a simple image filter that gathers neighboring pixel-values and compute a new value for a pixel. If done with respect to the underlying structure of the system one can expect a speedup near linear to the number of computational cores used. That is a CPU with four cores can theoretically expect a near four time speedup compared to a single core. This extends to a GPU so a GPU with n cores can in ideal cases expect a speedup in the order of n. An example of this is a Gaussian blur (or smoothing) filter.

1.2.4 Image compression

The image compression standard JPEG2000 offers algorithms with parallelism but is very computationally and memory intensive. The standard aims to improve performance over JPEG but also adding new features. The following sections are part of the JPEG2000 algorithm[2].

1. Color Component transformation
2. Tiling
3. Wavelet transform
4. Quantization
5. Coding

The computation heavy parts can be identified as the Discrete Wavelet Transform (DWT) and the encoding engine using Embedded Block Coding with Optimized Truncation (EBCOT) Tier-1.

The important difference between the older format JPEG compared to JPEG2000 is the use of DWT instead of Discrete Cosine Transform (DCT). In comparison to the DFT, the DCT operates solely on real values but at the same time complexity. DWT's on the other hand uses another representation that allows for a time complexity of $\mathcal{O}(N)$.

1.2.5 Linear algebra

Linear algebra is central to both pure and applied mathematics. In scientific computing it's a highly relevant problem to solve dense linear systems efficiently. From the initial uses of GPUs in scientific computing the graphics pipeline was successfully used for linear algebra through programmable vertex and pixel shaders[6]. Later on methods and systems for utilizing GPUs have been shown efficient also in hybrid system (multi-core CPUs + GPUs)[8]. Linear algebra is highly suitable for GPUs and with careful calibration it is possible to reach 80%-90% of the theoretical peak speed of large matrices[9].

Common operations are vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. Matrix multiplications are of much interest since the high time complexity $\mathcal{O}(N^3)$ makes it a bottleneck in many algorithms. Matrix decomposition like LU, QR and Cholesky decomposition are used very often and are subject for benchmarking GPUs to linear algebra[9].

1.2.6 Sorting

The sort operation is an important part in computer science and have been a classic problem to work on. There exists several techniques and mostly it comes down to what problem you have and choose the best suited algorithm.

Sorting algorithms can be organized into two categories, data-driven and data-independent. The classic quicksort algorithm is probably the best known example of a data-driven sorting algorithm. It performs with time complexity On log(n) on average but have a time complexity of ON2 in the worst case. Another data-driven algorithm that does not have this problem is heap sort but instead it suffers from difficult data access patterns. Data-driven algorithms are not the easiest to parallelize since the behaviour is unknown and may cause bad load balancing.

The other category are the algorithms that always perform the same process no matter what the data. This behaviour makes suitable for implementation on multiple processors, fixed sequences of instructions where the moment in which data is synchronized and communication must occur are known in advance.

Efficient sorting algorithms

Bitonic sort have been used early on in the utilization of GPUs for sorting, even though it has the time complexity of On log(n2) it's been an easy way of doing reasonably efficient sort on GPUs. Other high performance sorting on GPUs are often combinations of algorithms. Examples of fast sorting algorithms on GPUs have used bucket sort or quicksort that first splits the list into sublist and then sort in parallel with merge sort or by using bitonic sort followed by merge sort.

A popular algorithm for GPUs have been variants of radix sort which is a non-comparative integer sorting algorithm. Radix sorts can be described as being easy to implement and still as efficient as more sophisticated algorithms. Radix sort works by grouping the integer keys by the individual digits value in the same significant position and value.

1.2.7 Criteria for Algorithm Selection

For this thesis, a benchmarking application is sought after that have the necessary complexity and relevance to both practical uses and the scientific community. The algorithm with enough complexity and challenges is the FFT, compared to the other presented algorithms the FFT are more complex than the matrix operations and the regular sorting algorithms. The FFT does not demand as much domain knowledge as the image compression algorithms but its still a very potent algorithm for many specific applications.

The major difficulties working with multi core systems are applied to GPUs. What GPUs are missing compared to multi-core CPUs are the power of working in sequential, instead GPUs are excellent at fast context switching and hiding memory latencies. Most effort of working with GPUs must be to put into supply with enough parallelism, avoiding branching and refine memory access patterns.

One important issue is also the host to device memory transfer-time. If the algorithm is much faster on the GPU, the CPU could still be the winner if the host to device and back transfer is a large part of the total time. By selecting an algorithm that have much scientific interest and history; a lot of comparisons can be made and it is sufficient to say that one can demand a reasonable performance by utilizing information sources concerning other implementations on GPUs.

2

Theory

Det här är kapitlet där teorin presenteras.

3

Implementation

The FFT algorithm has been implemented in C/C++, CUDA, OpenCL, DirectCompute and OpenGL on a GeForce GTX 670 and Radeon R7 260X graphics card and a Core i7 3770K 3.5GHz CPU.

3.1 Algorithm

The implementation can be broken down into a few steps, see figure 3.1 for an simplified overview. The algorithm setup is platform-dependant but some steps are common; get platform and device information, allocate device buffers and upload data to device.

The next step is to calculate the specific FFT arguments for each kernel. The most important differences between devices and platforms are local memory capacity and thread and block configuration. Threads per block was tweaked for the best performance. See table 3.1 for details.

The implementation of a N-point radix-2 FFT algorithm have $\log_2 N$ stages with $N/2$ butterfly operations per stage. A butterfly operation is an addition, a subtraction, followed by a multiplication by a twiddle factor, showed in figure 3.2.

The algorithm thread and block scheme was one butterfly per thread, so that a sequence of sixteen elements require eight threads. Each platform was config-

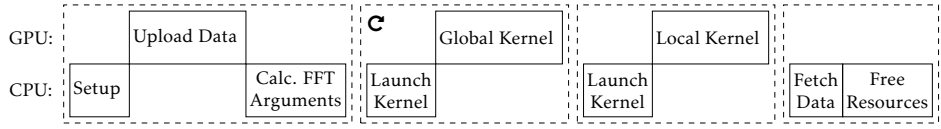


Figure 3.1: *Overview of the events in the algorithm.*

Device	Platform	Threads / Block	Max Threads	Shared memory
GeForce GTX 670	CUDA	1024	1024	49152
	OpenCL	512		49152
	OpenGL	1024		32768
	DirectX	1024		32768
Radeon R7 260X	OpenCL	256	256	32768
	OpenGL	256		
	DirectX	256		

Table 3.1: *Shared memory size, threads and block configuration per device.*

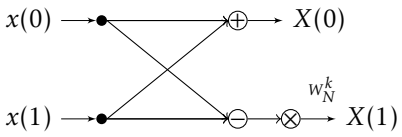


Figure 3.2: *Butterfly operations*

ured to a number of threads per block (see table 3.1) and any sequences twice as long as the threads per block configuration needed the algorithm to be split over several blocks. Example: if the threads per block limit is two, then four blocks would be needed for a sixteen element sequence.

Thread synchronization is only available within a block. When the sequence or partial sequence fitted within a block all data was transfered to local memory before completing the last stages. If the sequence was larger and required more then one block the synchronization was handled by launching several kernels executed in sequence. The kernel launched for block wide synchronization is called the global kernel and the kernel for thread synchronization within a block is called the local kernel. The global kernel had an implementation of the Cooley-Tukey FFT algorithm and the local kernel had constant geometry (same indexing for every stage). The last stage outputs from shared memory to the bit reversed index of the complete sequence. See figure 3.3 where the sequence length is sixteen and the thread per block is set to two.

The FFT algorithm for two dimensional sequences, such as images, is first a transform of each row (each row as a seperate sequence) and then a transform of each column. This GPU implementation does the first rowwise transformation and then transposes the whole image and repeat these to operations. An example is shown in figure 3.4.

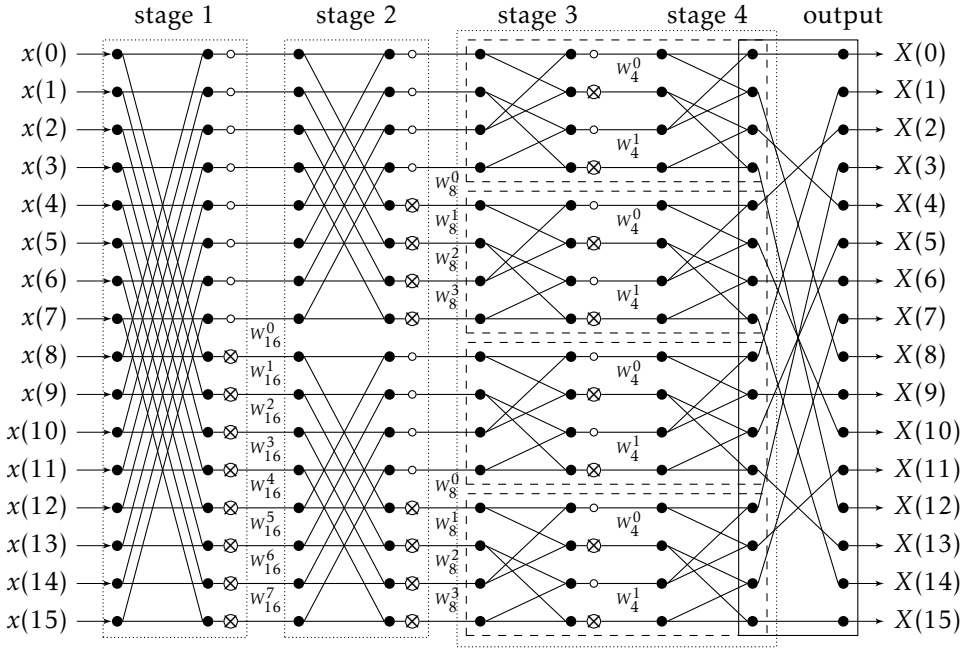
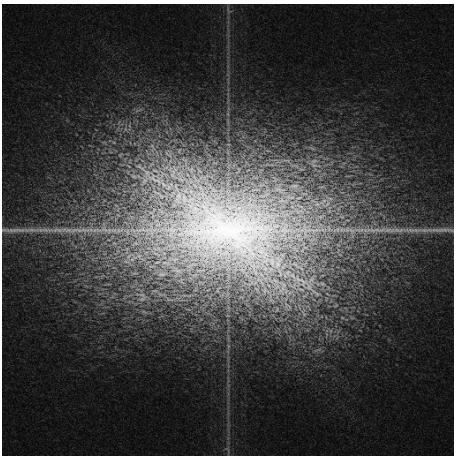


Figure 3.3: *Example flow graph of a 16-point FFT using (stage 1 and 2) Cooley-Tukey algorithm and (stage 3 and 4) constant geometry algorithm. The solid box is the bit-reverse order output. Dotted boxes are seperate kernel launches, dashed boxes are data transfered to local memory before computing the remaining stages.*

(a) Original image



(b) Magnitude representation

Figure 3.4: *Original image to the left. The image to the right is a quadrant shifted magnitude visualization of the original image.*

4

Resultat

Det här är kapitlet där resultaten presenteras.

4.1 Ditten

Liksom [**?**] har vi kommit fram till att glass smakar bäst på sommaren.

När vi nu går in på hur glass smakar vid olika tidpunkter under dagen hänvisar vi till figur 4.1, och speciellt till figur 4.1b. Jämför sedan med figur 4.2 för att se hur det kan bli när man äter glass vid okontrollerade tidpunkter.

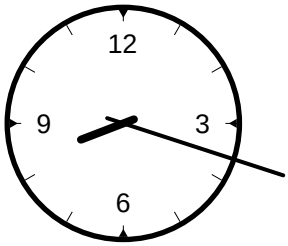
Veselić, Krešimir (Veselić, Krešimir) skrev en gång en artikel med titeln *Bounds for exponentially stable semigroups*.

Kommer att tänka på en liten anekdot...

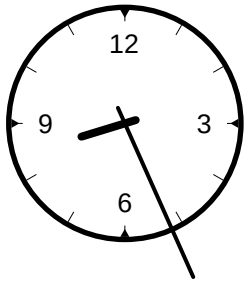
TODO:
Ta bort den löjliga anekdoten!

4.2 Framtiden

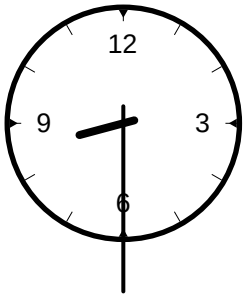
Sen när glassen är uppäten är det bara till att sätta igång och skriva på exjobbet igen!



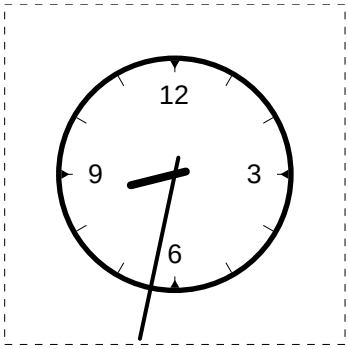
(a) *Det här är väl tidigt — din glass hinner smälta innan ditt sällskap dyker upp.*



(b) *Kiosken stänger snart, men inte nu — perfekt!*



(c) *Precis i tid — du får in ett finger i luckan just när kiosken ska stänga. Han som jobbar blir sur, och det blir smolk i bägaren.*



(d) *Du är sen — kiosken är stängd.*

Figure 4.1: *Illustration av* subfloats. *Den så kallade* bounding box*en visas i (d). Lägg märke till att bounding boxen har satts så att alla bilder har samma storlek, med enhetlig placering av själva innehållet i förhållande till bounding boxen. Antag att du ska träffa en kompis för att äta glass just när kiosken stänger för dagen vid 08:30. När dyker du upp?*

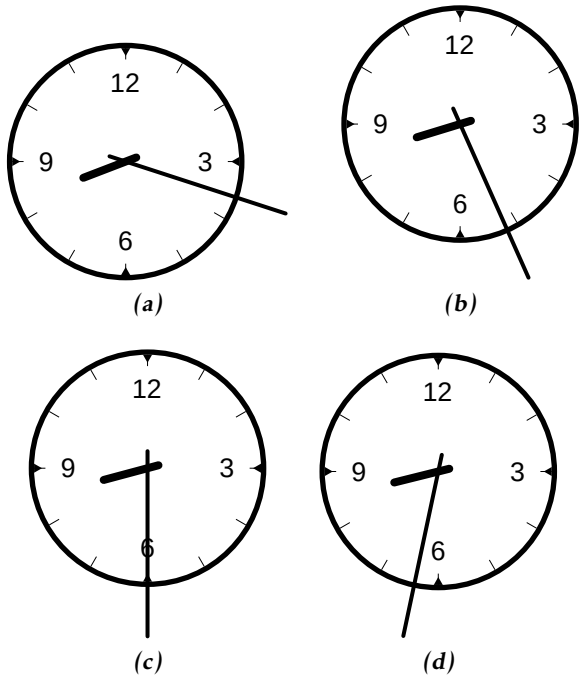


Figure 4.2: *En andra illustration av* subfloats. *Den här gången har bounding boxen gjorts så liten som möjligt runt själva innehållet. Resultatet är stökiga placeringar på sidan. Samma sak kan hända med vanliga fyrkantiga figurer när man har text som spretar ut åt lite olika håll från själva rutan med kurvor i.*

Appendix

4.A Ett par långa bevis

Det här är en appendix-del av det aktuella kapitlet.

5

Avslutande kommentarer

Sätt av ett kort kapitel sist i rapporten till att avrunda och föreslå rikningar för framtida utveckling av arbetet.

Appendix

Bibliography

[1] E. O. Brigham and R. E. Morrow. The fast Fourier transform. *Spectrum, IEEE*, 4(12):63 –70, 1967. ISSN 0018-9235. doi: 10.1109/MSPEC.1967.5217220. URL http://ieeexplore.ieee.org/ielx5/6/5217195/05217220.pdf?tp={&}arnumber=5217220{&}isnumber=5217195$\delimiter"026E30F$nhttp://ieeexplore.ieee.org/stamp/stamp.jsp?tp={&}arnumber=5217220. Cited on page 3.

[2] Charilaos Christopoulos, Athanassios Skodras, Touradj Ebrahimi, and Corporate Unit. The {JPEG2000} Still Image Coding Systems: An Overview. *IEEE Trans. Consumer Electronics*, 46(4):1103–1127, 2000. Cited on page 4.

[3] James W. Cooley, Peter a. W. Lewis, and Peter D. Welch. The Fast Fourier Transform and Its Applications. *IEEE Transactions on Education*, 12(1), 1969. ISSN 0018-9359. doi: 10.1109/TE.1969.4320436. Cited on page 3.

[4] W M Gentleman and G Sande. Fast Fourier Transforms: for fun and profit. *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966. URL papers2://publication/uuid/7065C1C0-089B-4DA8-8524-D5B62CB2B37A. Cited on page 3.

[5] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast Fourier transform. *Archive for History of Exact Sciences*, 34(3):265–277, 1985. ISSN 00039519. doi: 10.1007/BF00348431. Cited on page 3.

[6] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3): 908, 2003. ISSN 07300301. doi: 10.1145/882262.882363. Cited on page 4.

[7] Ik Park, Nitin Singhal, Mh Lee, Sangdae Cho, and Cw Kim. Design and performance evaluation of image processing algorithms on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):91–104, 2011. Cited on page 3.

[8] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. *Proceedings of the*

2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW 2010, 2010. ISSN 1878-3449. doi: 10.1109/IPDPSW.2010.5470941. Cited on page 4.

[9] Vasily Volkov, James Demmel, and U C Berkeley. Benchmarking g GPUs to Tune Dense Linear Algebra. (November), 2008. Cited on page 4.