Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Martin Brehovský

GPU Accelerated Adversarial Search

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Branislav Bošanský

Study programme: Informatics

Specialization: Software Systems

Prague, 2011

This thesis would not have been possible without the continuous support of my parents who always support me and give me the will to succeed.
I would like to thank my supervisor Branislav Bošanský for his support and advice.
Finally, I thank all my friends who so patiently put up with me while I worked on my thesis.

Název práce: Akcelerace adversariálních algoritmů s využití grafického procesoru
Autor: Martin Brehovský
Katedra / Ústav: Kabinet software a výuky informatiky
Vedoucí diplomové práce: Mgr. Branislav Bošanský, Katedra kybernetiky, Fakulta elektrotechnická, České vysoké učení technické v Praze

Abstrakt: Moderní programovatelné grafické čipy umožňují významným způsobem urychlit běh výpočetně náročných algoritmů. Tato technologie schopná masivní paralelizace výpočtů významně zvyšuje výkon velké skupiny algoritmů. Tato práce se zaměřuje na využití grafických procesorů (GPU) v akceleraci algoritmů na takzvané prohledávání herních stromů. Zkoumáme, zda jsou tyto algoritmy vhodné pro paralelizace typu SIMD(single instruction multiple data), jež GPU poskytuje. Proto byly paralelní verze vybraných algoritmů pro GPU srovnány s algoritmy běžícími na CPU. Získané výsledky ukazují výrazné zlepšení rychlosti a dokazují použitelnost GPU technologií v oblasti prohledávání herních stromů.

Klíčová slova: Grafický procesor, Adversariální algotitmy, SIMD paralelizmus, prohledávání herního stromu

Title: GPU Accelerated Adversarial Search
Author: Martin Brehovský
Department / Institute: Department of Software and Computer Science Education
Supervisor of the master thesis: Mgr. Branislav Bošanský, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague

Abstract: General purpose graphical processing units were proven to be useful for accelerating computationally intensive algorithms. Their capability to perform massive parallel computing significantly improve performance of many algorithms. This thesis focuses on using graphical processors (GPUs) to accelerate algorithms based on adversarial search. We investigate whether or not the adversarial algorithms are suitable for single instruction multiple data (SIMD) type of parallelism, which

GPU provides. Therefore, parallel versions of selected algorithms accelerated by GPU were implemented and compared with the algorithms running on CPU. Obtained results show significant speed improvement and proof the applicability of GPU technology in the domain of adversarial search algorithms.

Keywords: GPU computing, adversarial search, SIMD parallelism, game tree search

Table of Contents

1. Introduction

An increasing demand for ever-greater processing power has been one of the greatest challenges facing modern-day science and industry in recent years. Computers are used in computational science, allowing scientists to perform a variety of experiments. Currently, these scientists are bound firstly by the processing performance of the hardware available to them; and consequently by budgetary constraints that may inhibit the desired evolution of the hardware model. The computational challenges cannot be faced so easily with traditional laboratory approaches. The overall result of an experiment may be beyond the scope of a conventional setup due to the complexity of the evaluated elements and executed computations. Detailed precision of measurements are difficult to obtain in common laboratory conditions and hence, computers themselves are effectively becoming the new laboratories.

Simulations optimized and broken down into a huge amount of simpler computations put an enormous load on computer systems. They can fully utilize what traditional central processing unit (CPU) based computers have to offer, but CPUs organized in clusters are the only alternative for significant performance gains. Investments to construct new supercomputers are enormous and therefore access to high performance is limited for common research. As described in [1] a notable example is represented by microscopic models of the structure of surfaces at the nano-scale, which cannot yet be characterized experimentally using available imaging techniques. Conditions that cannot be created in laboratories are also simulated over different time scales. A pretense setup can monitor, for example, each nanosecond of an experiment or the evolution over thousands of years in soil water to climate change.

In many fields, computer simulations are essential and cannot be replaced by experiments. In multiple physical, medical and chemistry disciplines highly complex programs are at the cutting edge of new discoveries. The most difficult boundary they struggle against is the performance limitation of the available computer systems. The evolution of performance tuning has reached a new era where graphical processing units (GPU), previously used only to process graphics, are now widely used to support such problems in a parallel manner. Developments in the GPU world now allow easier access to computational resources for a broader range of users by creating a more user-friendly interface.

1.1 Graphical processing units in high performance computing

The performance of the central processing unit, which increases by Moore's law, estimates the upper boundary for high performance computing (HPC). Computations in science use more sophisticated algorithms and models, so the demand for better performance from these HPC systems easily outruns Moore's Law. The previous generation of scalable supercomputers relied on vendor-specific systems using high-performance, proprietary microprocessors, proprietary interconnects and vendor system software. This customized hardware is usually very expensive. One of the promising ways to increase processing power is to increase the granularity of parallelism in applications. Utilising simultaneous multi-threading can exploit thread-level parallelism. As is discussed in Chapter 2, the CPU itself is not designed to fully utilize the problems that HPC struggles with and so new technologies must be evaluated.

As mentioned, up until a few years ago, graphical processing units were employed for graphics only. Programming on them was very difficult and data had to be mapped into textures. Only highly experienced programmers were able to utilize their potential power. In recent years, attempts to unify access to GPU led to proprietary interfaces to access GPUs from the manufacturer NVIDIA. Cuda is the first example that provides high level access for the common programmer. This interface revealed its performance for massive, general purpose, parallel computing. OpenCL is an open standard for writing programs that execute across heterogeneous platforms including CPU and GPU. Suitable cases for GPU are computationally intensive tasks, that require high floating point performance on multiple independent data sets. Data parallelism supported by GPU has its limitations and further investigation is needed to fully utilize GPU's resources across specific domains.

1.2 Researched domain

One of the candidates to examine the suitability of using GPU to accelerate algorithms is in Artificial Intelligence (AI). Not a lot of research was initially done in applying GPU hardware to this field, while AI also faces a number of computationally expensive problems. One example is adversarial search or planning algorithms. This domain is interesting to research since the problems it faces have a simple definition although solutions are computationally very intensive. Some successful attempts to improve parallel path-finding algorithms on CPU based clusters and grids were researched in [2, 3]. Similar research in the field of parallel pathfinding [4] took place. The promising results showed the feasibility of applying parallel processing to this field.

Adversarial search algorithms, often known as games algorithms, have significant importance in game theory. Every multi-agent environment can be viewed as economy where agents/players interact in a cooperative or competitive manner. The problem in adversarial search is to reach the goal of a player, with respect to the behavior of other agents also trying to reach their goals. In adversarial search, agents are usually competitive. A basic approach to solve an adversarial search problem is the simulation of all the possible states that can be reachable and by choosing a proper strategy to reach that goal. The number of actions in such economy allowed to perform by each agent defines the branching factor and by that, the complexity of the searched space. The size of this searched space size then grows exponentially with each simulated step.

With sequential algorithms, it can be very time consuming to examine the game space into levels of depth sufficient to determine the optimal strategies. Many current non-parallel CPU based approaches use heuristics to improve performance algorithms. The purpose of this thesis is to investigate further possibilities for the use of GPU based technology in adversarial search and to examine the limitations of the technology in this field. Further in this thesis we investigate currently available sequential and parallel algorithms and their implementations. Analysis of researched fields provides a broad overview with respect to the specifics of GPU architecture and its programming interface. Identification of suitable approaches serves as a starting point for case studies. We prove that chosen algorithms fit for GPU processing by implementing analyzed algorithms first on theoretical game scenarios. Consequently, the results are backed up by practical implementation on representative games. We discuss benchmarks that present speedups with respect to the specifics of GPU architecture. An analysis of the results presents speedups and recognized drawbacks and limitations.

1.3 Outline

The purpose of this thesis is to investigate further possibilities for the use of GPU based technology in adversarial search and to examine the limitations of the technology in this field.

In Chapter 2 we briefly describe the evolution of CPU and GPU based systems. This gives the reader a better insight in architectural differences and technological details. Recognizing them help to realize the potential of this technology, its strengths and its limitations. The basic differences enlighten the suitability for different tasks and by that, to help recognize the best usage on algorithms more easily.

OpenCL as an open standard for programming on heterogeneous platforms is presented in Chapter 3. This framework provides an interface to access and use GPU capabilities. The technical specification of this framework is essential for the design of performable, massively parallel applications using GPU. The basic architecture, programming and memory model is described and we concentrate on well known limitations and recommendations. These are used as the basic knowledge in the analysis of approaches from the researched domain.

In Chapter 4 we provide a detailed presentation of key attributes of an examined Artificial intelligence domain. First, we introduce basic problems and concepts then we describe the standard approaches that are used to tackle them, with currently available sequential and parallel algorithms presented. We introduce theoretical approaches used to solve adversarial games and inspired by successful real life applications where speedup was proven, we identify the ones that are mots suitable for implementation with GPU acceleration.

In Chapter 5 the parallel nature of adversarial search and applicable use of GPU power is examined to cut down processing time. We concentrate on suitable tree decomposition, identification and assignment of sub-tasks that will be performed in parallel and has to fit the GPU architecture. Communication and synchronization issues influencing the performance of algorithms are mainly determined by memory architecture. We present and discuss benchmarks for several scenarios on multiple setups that showed both gains and limitations of GPU technology on this type of problems. The researched algorithms were tested on different configurations of synthetic game trees and the speedup are presented on an implementation of the game Fox and Hounds. A comparison of benchmarks obtained from both sequential and parallel algorithms is presented. The suitability of GPU to accelerate adversarial search algorithms is discussed and illustrated on benchmarks.

In Chapter 6 we conclude the thesis and discuss possible future work.

2. CPU architecture evolution, problems, limitations and comparison to GPU

Problems inhibiting the performance of CPU are motivating developers to look for other ways to accelerate their applications. GPU emerged as a possible addition and access to its computational power is easier now than ever before. In this chapter we present the major differences between CPU and GPU architectures. Each technology is designed to serve a predefined purpose - CPU to perform control logic and GPU to perform huge number of parallel computations, of limited complexity very fast. A good understanding of both helps to choose the best possible technology with respect to their specifics and to fully utilize the potential of the underlying hardware.

2.1 Evolution of the CPU over the last 20 years

Over the last twenty years, the performance of central processing units (CPU) based on microprocessors has increased while the price rapidly decreased. CPUs from Intel and AMD were able to perform over one billion floating point operations per second (GFLOPS) at a reasonable price for the average user and hundreds of GFLOPS solutions in cluster. Software development also evolved over those years to use the power of available resources. With the vision of ever-growing performance of available hardware, programmers were not motivated to investigate new approaches in the development of software or alternative uses of the hardware. Moore's law describes a long-term trend in the density of transistors on an integrated circuit. It states that the density of transistors on the circuit will approximately double every two years. It held over the last 50 years [5]. In recent years it has proven to be harder and harder to keep up the exponential growth due to manufacturing issues and higher current leakage on smaller scales. The cost of developing processes to manufacture these circuits rises perhaps even faster than the transistor count. Another important factor is the manufacturing cost. A necessary change in architecture introduced processors with multiple cores that are used per single chip, but an increase in the number of transistors does not guarantee linear growth in practical CPU performance. In fact, a multi-core CPU's speed doesn't greatly increase in many applications that are not specifically designed and implemented with respect to the hardware capabilities. Case studies showed that an increase of 45% in the number of transistors on a processor translated to only about a 10–20% increase in processing power [6]. The majority of applications are implemented as sequential programs, thus delivering limited performance by not taking advantage of hardware resources. Performance improvement in applications running on an ever-growing number of transistors is achieved by changing the nature of these applications to become parallel programs. Concurrency revolution [7] is a dramatically escalated incentive of parallel programs where multiple threads of execution, running at the same time, co-operate to deliver work much faster. Such applications run on large scale computers or computer clusters that are very expensive. The necessary performance growth in many fields of computational science requires a dramatic increase. There are many predictions about when the continuous trend described in Moore's law will end. There are still a few other approaches on how to enhance the performance of computing, and one of the most promising areas is to take advantage of a larger number of small processors dedicated for specialized operations organized into one high-speed, massively parallel processing system. This is where graphic processing units(GPU) emerged as a promising candidate.

2.2 Changes in CPU architecture

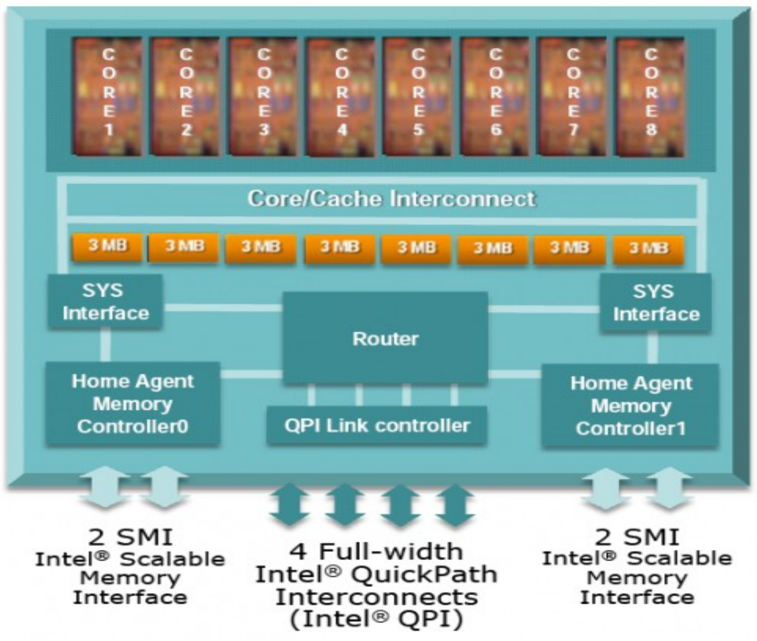
Since the introduction of one of the first modern Pentium microprocessors on March 22 1993, the CPU evolved in many ways to try to scale performance and match Moore's law. In the first concept of this architecture, a major part of the chip's surface area was covered with transistors. Products with code 80501 contained 3.1

million transistors that covered 293.92 mm^2 [8]. Further improvements came with the next generation Pentium II processors presented in 1997, doubling the number of transistors to 7.5 million as well as being half the size of it's predecessor. This model was designed to cut costs while introducing a larger L1 cache and cheaper and slower L2. Pentium III in 1999 brought with it a total of 9.5 million transistors over 128 mm^2 and introduced an enriched SSE instruction set to better support and accelerate floating point operations. The following generation increased the cache further while decreasing the size of the transistor, making the instruction set richer while introducing hyper-threading technology. Noticeably, the main idea behind increasing performance was to make the transistors smaller and the cache larger while supporting a more complex instruction set. Limitations on size reduction, cache size and energy consumption pushed hardware architects into exploring new approaches. Hyper-threading as the way to create virtual processors and by that improve the parallelism of computations showed possibilities to improve the overall performance.

Codename:	Penryn	Bloomfield	Gulftown	Beckton
Architecture	Penryn	Nahalem	Westmere	Nehalem EX
Socket	775	1366	1366	1367
Cores/Threads	2/2	4/8	6/12	8/16
Hyper-threading	No	Yes	Yes	Yes
L3 Cache	No	8MB	12MB	24MB

Table 1: Overview of current multicore CPUs by Intel

Further research into performance enhancements by CPU vendors such as Intel resulted in a multicore solution. As seen from Table 1, the number of cores almost doubles with each generation. Such chips are out-of-order, multiple-instruction issue processors implementing a full x86 instruction set suited to optimize the performance of sequential programs. Compared to single core CPUs, cache sizes increased and a new L3 cache with a 200GB/s bandwidth and 24MB size was introduced in the latest version of the Nahalem EX architecture. With 2.3 billion transistors, it is the most complex processor produced by Intel.



CORE 1
CORE 2
CORE 3
CORE 4
CORE 5
CORE 6
CORE 7
CORE 8
Core/Cache Interconnect
3 MB
SYS Interface
Router
SYS Interface
Home Agent Memory Controller0
QPI Link controller
Home Agent Memory Controller1
2 SMI Intel® Scalable Memory Interface
4 Full-width Intel® QuickPath Interconnects (Intel® QPI)
2 SMI Intel® Scalable Memory Interface

Figure 1: Architecture of Nahalem EX [8]

From Figure 1 we can see that the 24 MB L3 cache is divided into eight separate blocks of 3 MB. All of these blocks can be used simultaneously for multiple cores. In this way the bandwidth between cores and cache is maximized. From the description above it is clear that the development of new CPU architectures leads into more extensive parallelism of execution with an increased number of cores. We also get a faster, dedicated cache adjusted to provide a higher throughput suited for further generations of CPUs. Of course there are several limitations, which is described further.

On the other hand, the GPU world is based on a multi-core architecture designed to achieve the highest possible throughput of parallel programs. They are well suited for single instruction multiple data (SIMD). Thanks to their in-order, single instruction issue processor, the performance of the most recent GPUs surpasses CPUs' performance by an order of magnitude. While the CPU's current peak performance is in the tens of GFLOPS, GPU's performance is about 1000 GFLOPS. As mentioned in [9] the peak performance growth rate of CPU microprocessors over the last decade has been relatively slow, while GPU performance has grown exponentially. The performance gap started widening around 2004. CPU annual growth stabilized at around 20% annually, while the GPU keeps scaling up at a rate of 50% of performance per year.

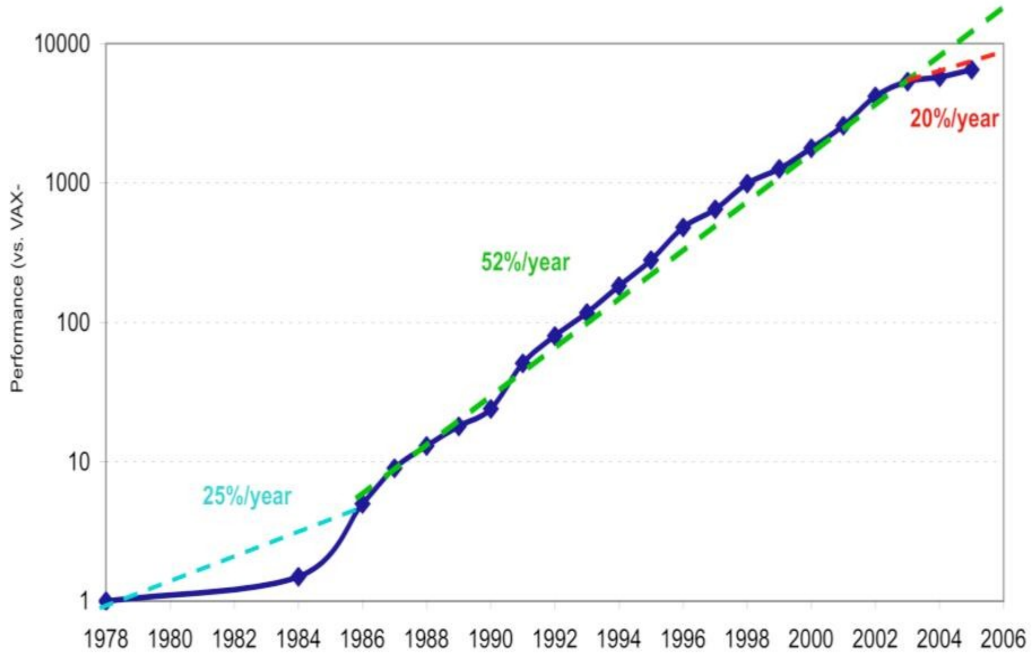


Figure 2. Performance growth over past years [9]

At this rate, the CPU will lose performance 1000 times compared to Moore's law until the year 2016 and observation of declining performance growth can be confirmed nowadays. The peak performance of the fastest CPU processor is 107.55 GFLOPS, while the NVIDIA Tesla C2050 performs around 515GFLOPS [10] in double precision calculations, and in single precision performance around 1.03 TFLOPS. As we can see from Figure 3, the availability of increasing performance changed the approach of building supercomputers and current top supercomputers nowadays are based on GPUs. For example, Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C, currently the fastest supercomputer in the world [11] is based on GPU technology and is able to perform 2566 TFLOPS. Third in the list is Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU with performance 1271TFLOPS. These recently constructed supercomputers show further investments in gaining performance into combined systems consisting of both GPU and CPU technology. Again from Figure 3, the inclusion of GPU power helps a lot these days to preserve the growth of performance of supercomputers. A combination of these technologies enables us to bridge the insufficiency of two different approaches of handling both instructions and data.

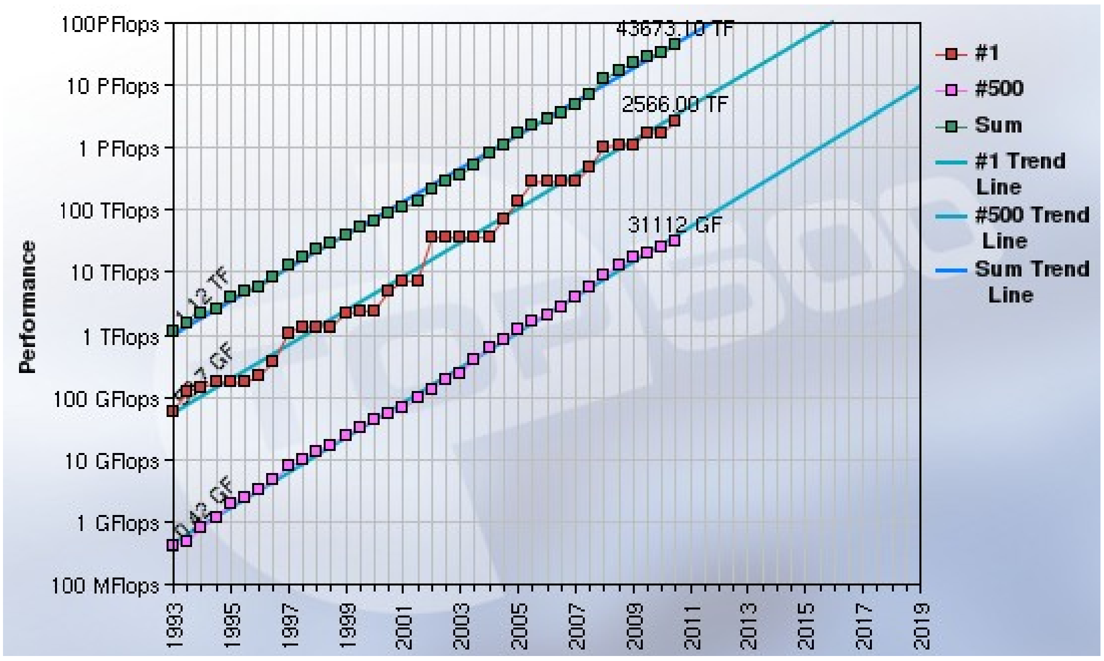


Figure 3. Supercomputers' evolution prediction [11]

There are fundamental differences in the designs of the two architectures, the main one being that the CPU is designed to increase performance of sequential applications. Control logic of CPU schedules instructions to be executed in an optimal manner on all the resources it has. It tries to organize instructions running in a single thread to be executed in parallel or even reschedule them out of their sequential order while preserving the overall picture of sequential execution. Instructions and data access latencies are reduced by using big caches. The complexity of instructions and a lack of parallelism in applications make the presence of those parts necessary. Arithmetical logical units (ALU) or processor cores are the parts of the CPU that perform all of the instructions. Usually, in current processors, there are 4-16 cores. In comparison, the GPU uses minimalistic control logic and cache - the number of cores is 100 times higher, but their instruction set is simpler.

Another very critical aspect while comparing performance is memory bandwidth. In the GPU world it is very important to handle large textures, filtering and anti-aliasing. Memory bandwidth is important in nearly every aspect of graphics processing. Designers pushed by the demands of the gaming world are challenged by the requirements of an economically strong industry. The number of simple floating point operations per video frame is significantly increasing so that the execution of a huge number of threads needs to be optimized. The hardware performs minimal control logic of execution for each thread and thus it reduces long latency memory access. Current graphic cards, for example the NVIDIA Tesla, has a throughput of about 150 gigabytes per second (GB/s) [10], while the peak in CPU models is about 35GB/s.[8]

2.3 Problems influencing performance of CPU based systems

There are many problems that limit the performance growth of a system. The speed of a program is always influenced by the performance of the whole system, not just the part where computation is done. For example, important aspects are memory and disk access latency. As described in [9] the value of resources changed over the years and with them, the view on computer architecture design. Several barriers were defined from this perspective as a challenge to hardware engineers and software developers. These are power, memory, frequency, cost and parallelism.

In the past, these barriers in computer architecture were defined differently than they are nowadays. Power itself was not a limitation at all, while the density of transistors per chip was an issue. Proportionally, executing operations such as multiplication were considered to be slow, while accessing memory was fast. A higher level of of parallelism was achieved by investing in out-of-order execution, speculation and branch prediction in sequential programs. The situation changed and new boundaries have now been reached, so the aforementioned problems had to be redefined.

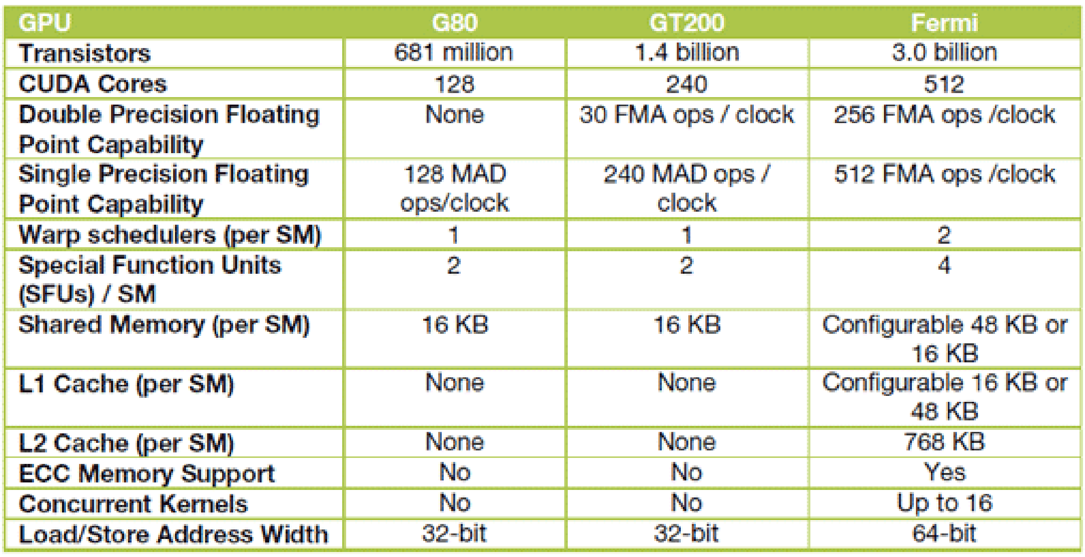
A memory barrier limits the bandwidth of the channel between the CPU and a computer's memory. Increasing the number of cores increases the demanded memory bandwidth. The cache itself occupies a fair part o the chip and requires a lot of power to manage. Increasing the cache size and bandwidth requires an increase in power consumption. One of the basic laws of physics states that all electrical power consumed by a system is eventually radiated as heat. The chip's overall temperature and power consumption is limited and depends highly on the availability and cost of cooling technologies. The power per transistor rises with frequency but decreases with area. Smaller transistors require less power which can lead to an increased frequency. However, the transistor density also increases, which leads to a problem with heat dissipation. By splitting them into multiple units (multiple cores) they can be run in parallel at lower frequencies to maintain a similar throughput while saving on power consumption. Increasing the frequencies would lead to the power wall, so performance may be increased by increasing the level of parallelism. Recently, parallelism has been performed mainly at the instruction level. The instruction level parallelism (ILP) wall emerged with the availability of enough discrete parallel instructions for a multi-core chip. The processing of single instruction multiple data (SIMD) instructions or vector parallelism combined with out-of-order execution reached the point where substantial effort (and an increase in transistor count) brings only marginal gain. Limiting issues are clock rate, instruction fetch and decode per clock rate, memory bandwidth and its locality. Scaling the ILP wall requires a significant change in the level of parallelism of applications - using SIMD or vector instructions. One possible way to exploit the effect of ILP is to increase the granularity of parallelism in applications by simultaneous multi-threading or data intensive computing.

All these barriers influence each other and a change in one causes another to reach its limits. A significant additional factor is cost. For example, increasing the level of abstraction of developed applications would enhance parallelism but increase the number of man-days spent in design and development of programs. From the perspective of development of new hardware, costs can be categorized as design and manufacturing costs. Power limitations can also be addressed by increasing the number of much slower cores, rather than their frequency. One of the possible ways is in the use of multiple heterogeneous cores, specialized on a smaller set of instructions to improve power and performance.

The limitations presented create a significant shift towards a heavily multicore architecture. An increase in hardware support for parallel computing and its specialization is emerging as a possible gain for performance. An example may be a cooperation between a core that is designed primarily to handle control flow and branching logic, and different hardware accelerators suitable to accomplish massive SIMD operations. Promising candidates to accelerate data intensive computing are graphic cards. The newest architecture from NVIDIA is called Fermi, and an introduction to its specification and capabilities follows.

2.4 Overview of GPU architecture

In the past, NVIDIA presented tree generations of graphic card designs. The breakthrough was the G80-based GeForce 8800 introduced in November 2006. It substituted the separate vertex and pixel processors with a unified one programmable in the C language. This allowed a broader range of applications. The peak performance of 681 million transistors was about 500 GFLOPS compared with the peak of 20 GFLOPS for CPU at that time. It supports a single instruction multiple thread (SIMT) execution model and inter-thread communication using shared memory and synchronization barriers. The second generation of this architecture was GT200 first introduced in 2008 as GeForce GTX 280. Compared with the previous generation it consists of 1.5 billion transistors and delivers over 900 GFLOPS. The biggest improvement besides speed is the addition of IEEE 754R double precision floating point arithmetic and hardware memory access coalescing. The amount of memory doubled and the bus width of the memory to the GPU interface was also increased. The number of simultaneously processed threads increased three-fold (to 30,000).



GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Warp schedulers (per SM)	1	1	2
Special Function Units (SFUs) / SM	2	2	4
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache (per SM)	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Figure 4. NVIDIA GPU architecture overview [10]

As shown in Figure 4. the architecture of GT200 doubled most of the parameters over the previous generation. The architecture introduces cache hierarchy, error correction code (ECC) protection, faster double precision floating point arithmetic, context switching, PTX 2.0 instruction set and faster atomic operations.

In Fermi architecture, a chip consists of 3 billion transistors, 512 streaming processor (SP) or Cuda cores, which are grouped into blocks of 32 per streaming multiprocessor (SM). Each block of 32 cores has 64KB memory, that can be configured as shared memory or L1 cache in ratio 1:2 or 2:1. The L2 cache is a 768KB memory shared between the blocks. The GPU supports up to 6GB GDDR5 memory. In each SM there are two warp schedulers that simultaneously dispatch instructions from two independent warps. GPU is connected to CPU using a PCI-Express interface. Switching between applications is supported by GigaThread hardware scheduler. It is 20 times faster than any previous GPU and it can manage all 1536 simultaneously active threads.

Each of the 16 streaming multiprocessors that execute programs and manipulate data contains 32 SP as well as 16 load/store units, four special function units (SFUs), a 64KB block of high speed on-chip memory that can be used either to cache data for individual threads and/or to share data among several threads; and an interface to the L2 cache shared among all sixteen SM's. Each core can perform one single precision fused multiply-add(FMA) operation per clock cycle and one double precision operation per 2 cycles. The IEEE 754-2008 floating-point standard that Fermi supports includes all four rounding modes and subnormal numbers. FMA support increased the accuracy of several numeric operations. The SFU can handle four special operations such as sin, cos, exp and reciprocal per clock cycle.

The new parallel thread execution (PTX) 2.0 instruction set supports greater accuracy, performance and programmability. It implements a unified address space for all three memory spaces, thread local, block shared and global space for load/store instructions. Addressing is done in a 64-bit manner.

Compared to the previous generation of GPUs, Fermi comes out on top. For example, in the implementation of radixsort, Fermi was 4.3 times faster than its predecessor. In double precision applications like matrix multiplication and tri-diagonal solver, the performance increased 4.2 times [11]. Physical algorithms including fluid simulations delivered a 2.7 times speedup.

2.5 Summary

All the described differences show specific fields for the use for each technology. The resources that are available have to be used for what they are best suited. That is, the GPU for parallel processing of computationally expensive parts of applications, with the CPU running their control logic. A combination of the two different approaches delivers higher performance and makes it possible to execute numerically expensive applications. An application requiring huge processing power that can be achieved only by running it on large processor clusters limits the customer base that can afford it. The multi-core architecture of GPU makes resources for parallel processing feasible at lower costs and higher availability for the market. The size and price of a cluster itself is both a demotivating and limiting factor for the development and use of parallel applications. An overview of the framework OpenCL that allows us to access and use the computational power of GPU is presented in the next chapter. A general introduction into the architecture, execution and memory model allow a closer look into the characteristics of this technique. An important part of Chapter 3 exhibits well known recommendations and limitations.

3. OpenCL

OpenCL (OPEN Computing Language) is an open standard based on the C programming language to unify general-purpose computations on heterogeneous systems such as multi-core CPUs and the latest GPUs. The language specification is C-based across the platform programming interface, as a subset of ISO C99 with language extensions and a numerical rounding accuracy for all floating point operations with a defined maximum error based on IEEE 754. A brief overview of the platform, execution and programming model follows. It determines the way how resources of underlying hardware are used. Knowledge of them is crucial to be able identify their fit on researched algorithms and to create performable application.

3.1 Platform model

OpenCL provides a hardware abstraction layer over diverse computational resources. The host connects to one or more OpenCL devices . A device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PE). The devices needs to be queried, selected and initialized to use for further usage. Computations on a device occur within the processing elements. An application submits commands from the host to execute computations on the processing elements within a device. Execution can be performed in a single stream of instructions as SIMD units (execute in lock step with a single stream of instructions) or as SPMD units (each PE maintains its own program counter). All the devices that should be used in such an execution must be recognized, registered and assigned into one compute context. Each device must be associated with the command queue that allows the host application to assign commands to it. OpenCL takes care of compiling commands into the devices' specific instruction set. Recognition of heterogeneous devices and their characteristics is performed during execution time so, for example, the most suitable device can be picked at runtime based on input data or other parameters. The execution model about how to identify, assign and manage the devices follows. Illustration of platform model is displayed in Figure 5.

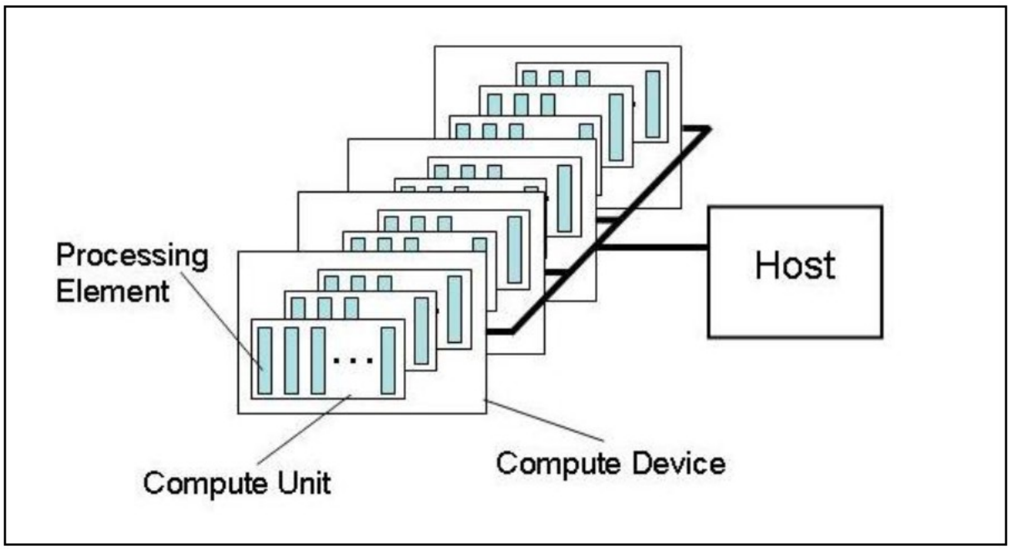


Figure 5. Platform model [14]

3.2 Execution model

A programmer has flexibility in the type of compute kernel that is used. Compute kernels can be thought of either as data-parallel, which is well-matched to the architecture of GPUs, or task-parallel, which is better suited to the architecture of CPUs. A compute kernel is the basic unit of executable code and can be thought of as a C function. Communication between the host application and the devices is managed via command queues. Each device must have an associated command queue within this context. Kernels are distributed to the processing elements of each device by commands. A command queue associated with a device schedules commands on its processing elements from the host application. Commands are distinguished as memory commands and kernel execution. Execution of kernels can proceed either in-order or out-of-order depending on the parameters passed to the system when queuing the kernel for its execution. Events are provided with each execution so that the developer can check on the status of outstanding kernel execution requests and other runtime requests. Details on exact API commands on how to create context, query devices and assign queues to them can be found in [16].

3.2.1 NDRange

A unit of work is called a work-item. Each work-item can be identified by it's specific global id. Work items are grouped into a work-group and each have their specific work-group IDs. Each work item can be identified within a work-group based on local ID. A combination of the local ID and a work-group ID uniquely represents each created kernel instance. Work-group items are executed concurrently on the compute unit associated with this group. On GPU, each work item runs in its own hardware managed lightweight threads. The entire index space of work items is called NDRange. It is N dimensional index space defining computation space, where N is one, two or three. An array of length N defines the number of items in each dimension.

According to [15] the recommended sizes of work-group size are powers of two when the possible and maximum local work-group size is less then or equals to 512.

3.2.2 Synchronization

Synchronization can be performed on multiple levels. In the case of one device and multiple kernels that are scheduled in its queue, synchronization depends on the chosen type of execution. When in-order execution is selected, commands are launched and finished in the same order to how they were added into the queue. In this case, no explicit synchronization is needed. Out-of-order execution, on the other hand, starts execution in the order they were added but it is non-blocking, so the second command is initiated immediately without waiting for the first command to complete. There is no guarantee that the result of the first command is available at the time the following command starts and in fact. The synchronization has to be done explicitly by using synchronization commands. These commands and the execution of the kernel and memory commands generate events. Such events can be used for this explicit synchronization between the commands, the host and the device. This includes the case when more devices are used, since each of them has its own command queue within the context.

A second type of synchronization is on a work-item level. As id described later, synchronization of memory between work-items of a given work-group can be done via shared memory. Since OpenCL uses relaxed memory consistency, memory visible to different work-items or commands may be different. Except at a barrier or other synchronization point. OpenCL provides a work-group barrier function that ensures all the work-items in work-group must reach a barrier before all of them can continue in an execution. With this barrier it is important to realize that the execution flow can be different per work-item. Since all work-items must reach a barrier, there must be careful usage of it in combination with *if, while, do, for* and *switch* instructions. In the case that a barrier is not reachable by even a single work-item, the execution never finishes.

3.2.3 OpenCL execution flow

Here is a brief summary of the steps of the execution. A host application identifies devices, queries their properties and picks the ones that fit the specific needs of application. It creates computational contexts for each chosen device. Each context must be associated with a command queue via that kernel so that the executions can be scheduled. Based on the C implementation provided, a program object must be created. A program is created from a source code string or binary. A program object encapsulates this list of devices, the latest successfully built executable for each device and a list of kernel objects. The kernel contains specific kernel functions in a program and argument values. The core of the whole execution starts by defining the order of commands to be passed to queues created beforehand. Data has to be written into a device using the appropriate queue before executing the kernel. Definition of dimensions have to be passed into the kernel as it has to be initialized on NDRange index space. Reading of results is the last stage of this simple execution model. All operations passed into the command queue can be performed in an asynchronous/non-blocking or synchronous/blocking manner. This is defined as a parameter passed to each function. For more details about the above mentioned functions and their parameters refer to [16].

3.3 Memory model

As displayed in Figure 6, a memory model in OpenCL consists of a multi-level hierarchy that differentiates in size, speed, location and type of access for the host and processing elements on a device. There are four memory spaces defined: private, local, constant and global. From the host, the only accessible memory are global and constant for both read and write operations. On a device, private memory is assigned and visible to every individual processing element. This memory space is dedicated to it and no other PE can access it. On the other hand, local memory is shared for all the processing elements within one processing unit and may be used for synchronization and exchange of information between them. The size of private and local memory is relatively small but speed of access is high. A processing element from one compute unit can not access the local memory of other unit. Global memory is accessible from all processing elements on a device. The size of it is much bigger than local and private memory, but speed of access is significantly slower. Constant and global memory are filled from the host.

3.3.1 Host to device memory transfer

This can be done by explicitly copying data or by mapping and unmapping regions of a memory object. As described above, commands to read or write memory objects from or to a device can be queued in a command queue, in both blocking or non-blocking ways. Mapping methods allows to map memory from hosts into devices' address spaces. One of the most important measures of performance for a system is memory bandwidth. Data transfer between a host and a device is considered to be slow compared to transfers on a device alone [17]. Such transfers should be minimized and the use of intermediate structures created and destroyed on a device is preferable.

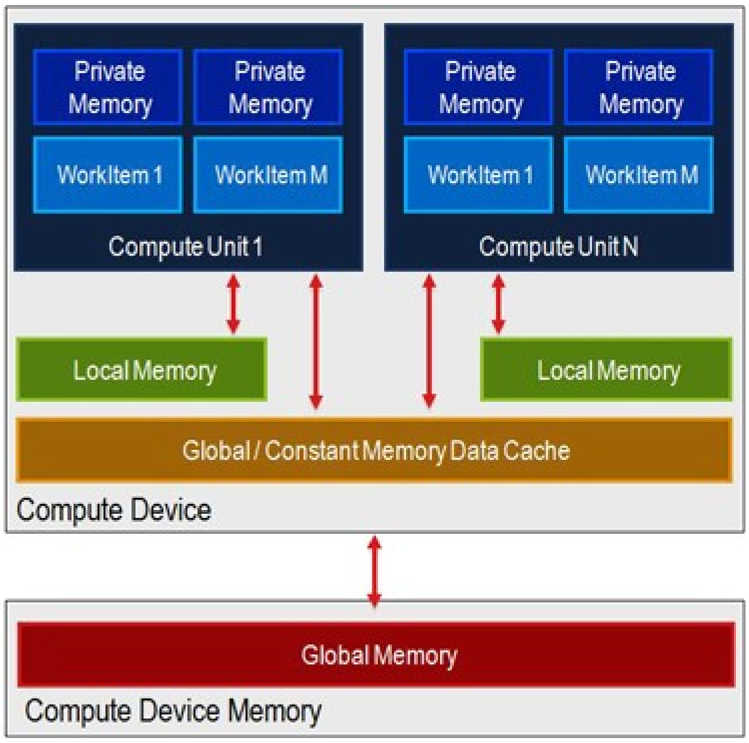


Figure 6. OpenCL memory model [18]

3.4 OpenCL C99 structure

There are a few differences between regular C99 and OpenCL C99, which was designed to fully support GPU architecture and conveniently control parallel execution. Some of the differences define direct limitations of the framework. Based on [16, 18], an overview of major differences are presented to give the reader a better insight into the capabilities and limitations of this framework.

Vital OpenCL specific extensions include the *kernel* or *__kernel* identifier that specifies what functions can be called by the host application via the OpenCL API. A prefix of "__" is not compulsory and may be ignored in all the identifiers mentioned here. In the declaration of variables, this is a keyword to distinguish address space, where this variable should be created and must be declared. The region of memory that is used to allocate the object can be identified by the *global,* *local, constant* and *private* keywords. If this identification is missing, generic address space is used.

As described above, each work-item can be identified uniquely in two ways; by its global ID or combination of group ID and local ID. Specific kernel instances must identify the part of the data array that it handles based on those attributes. Unique identification within all work-items globally per each dimension (1,2 or 3) can be queried using *get_global_id(dimension)*. The function *get_global_size(dimension)* returns the number of work-items specified to execute the kernel. Similar identification can be queried by *get_local_size* and *get_local_id* for work-items within a work-group. The number of workg-roups and work-group ID for a specific work-item is available via *get_num_groups* and *get_group_id.* Using these functions, it is possible within each kernel instance uniquely to recognize its input and output data structures in parallel execution.

Significant importance have built-in vector types, especially if using the GPU as a device, vector types are basic structures supported by the hardware. Build-in mathematical and common functions with hardware support may increase performance of whole processing. For example, multiply-add (mad) or *native_* geometric functions, cross and dot products for vector manipulation. Such functions may map to one or more native instructions and have typically better performance. An important note here is that the accuracy of *native_* functions is implementation defined.

Limitations of current OpenCL 1.1 C99 implementations compared to standard C99 include the absence of recursion support, dynamic memory allocation, pointers to functions, variable length arrays, bit-fields, and a lack of support for functions included in standard C99 headers. A list of them can be found in [16]. As discussed in [18], random number generators that are not included cause difficulties to implement learning and evolutionary algorithms that are by nature parallel and can benefit from parallel processing.

Referencing a programming model, a program to execute on a device is compiled at runtime. A program object can be created using *clCreateProgramWithSource()* or *clCreateProgramWithBinary()* so the definition of a program can be changed programatically in the host. This allow wider possibilities on how to manage the application logic.

3.5 Recommendations and optimizations

A best practices guide by NVIDIA in [19] describes some of the recommendations and optimizations for development using OpenCL on NVIDIA's devices. A subset of the most important are presented here to give the reader a clearer picture about use of OpenCL on GPU devices. The key issue is to gain the best performance from a GPU device via its support of a lightweight thread model. The application must be suitably adapted to fit into this architecture and the user should concentrate mainly on exploiting the level of parallelism in code with respect to underlying resources. A theoretical maximum speedup factor of program is defined by Amdahl's law. Since the number of processors that are available is high, a proportion of the program running in parallel must be increased as much as possible to gain significant speedup.

In the case of GPU, a PCI interface that usually connects host and device has a limiting bandwidth. To achieve maximum application performance, it is essential to minimize data transfer latency and therefore data transfers should be limited. The price paid for the amount of data transferred has to be justified by the complexity of computations on the device. Data should be kept on a device as long as possible and sometimes it is worth executing a command that is slower on the device than on the host without data movement to achieve an overall better performance.

Optimization of memory access by the kernel is crucial in GPU devices, as described in [19, 21] to fully utilize texture efficiency and coalesced data transfers. GPU memory access patterns enable the hardware to accelerate data access and perform read and write operations on multiple data structures in one operation. One of the most important characteristics of GPU architecture as a main architecture of

OpenCL-enabled devices from NVIDIA, is coalesced global memory access. The architecture bundles several threads from one workgroup for execution and each workgroup is partitioned into warps, each of which usually contains 32 threads (G80/GT200). The hardware executes an instruction for all threads in the same warp before moving to another. As described in [21], the advantage is when all threads in a warp execute the same instruction at any given point in time. When all threads in a warp execute a load or store instruction, the hardware detects whether the global memory locations are consecutive. If this is the case, the hardware combines all the consecutive accesses into a single request to DRAMs. Once a control flow is different for items within a warp, multiple passes are required to satisfy divergent memory requirements. Those passes are sequential so it increases the overall time of execution. The effects of misaligned access or stride access are displayed in [19]. The access to global memory is much slower than local memory, so for synchronization purposes it is recommended to use local memory instead of global.

Usage of instructions with high throughput is preferable. The precision of computations should be adjusted to the needs of the application. Double precision floating point operations are slower and so single precision should be used instead the former is not necessary. Built-in native mathematical functions are mapped directly to the hardware level and are faster at the cost of lower precision.

All the above mentioned characteristics significantly influence the performance of applications, with respect to specifics of NVIDIA's GPU architecture. Since the main purpose of this thesis is to evaluate possible acceleration of adversarial search algorithms using GPU, those specifics play an important role in the use cases presented in Chapter 5. All the specifics of GPU architecture are fully accessible using OpenCL. Besides the standard, other frameworks, some of which are described briefly next, were created to access computational performance of GPU devices.

3.6 Alternatives

Some alternatives to OpenCL are available. One of them is a vendor specific solution from NVIDIA. It is called Cuda SDK and it supports GPU-specific characteristics and optimizations from this vendor very well. Compared to OpenCL, the architecture and memory model are very similar. For most of the fundamental characteristics there exists a one-to-one mapping. The work item is represented in Cuda as a thread, a work-group is called a thread block and NDRange is called a grid. The main difference is that OpenCL as a standard is designed to support code portability across devices produced by different vendors, thus allowing for greater hardware diversity. Device management, kernel compilation and execution are more complex in OpenCL. In Cuda there is a special kernel calling syntax and a variety of hardware specific variables. But as examined in [22] a comparison of Cuda versus OpenCL performance showed very similar results on benchmarks with high data sizes. Cuda is tailored for Nvidia products, but suffers from a lack of support by other vendors. Portability on heterogeneous devices is very limiting compared to OpenCL. However, by allowing access to specifics of the hardware it makes it possible to use its full potential.

4. Adversarial search

A basic introduction into the adversarial search domain presents in this chapter it's complexity, main problems and algorithms used to solve them. Improvements in precision, speed and the size of space of search depends complexity of algorithm and it's ability to use underlying hardware resource. Recently the most significant results were achieved using parallel algorithms. In this chapter we introduce theoretical approaches used to solve adversarial games and inspired by successful real life applications where speedup was proven, we identify the ones that are mots suitable for implementation with GPU acceleration.

4.1. Multi-agent environments

Artificial intelligence provides a way to formalize environments, where large number of agents interacts and changes it's state. The behavior of agents can be cooperative or competitive, depending on their individual goals. Each agent or player has a predefined set of possible moves. Because players interact, some of the moves may not always be possible to perform. Agents evaluate the situation of the world independently, based on their specific rules and knowledge. One can not predict another player's next move, so to make a decision, one has to consider all possible moves of other players. A combination of allowed move sequences for every player generates all possible states of the world. States where agent has reached its goal are called terminal states. Each performed move has a cost that influences all agents welfare. Sequences of moves leading to a terminal state may have different costs. The goal of an agent is to reach terminal state with the maximal possible value. In competitive environments the goals are usually adverse, so minimizing one agents costs maximizes the costs of the others. In general, adversarial search is defined as search in multi-agent environment where the goals of players are contrary. Important representatives of adversarial search problems are games.

4.2. Games

Games have a great importance. Their rules are usually simple to formalize, while complexity of search space for optimal strategy is high. Games have structured tasks and clear definition of agents goals. In general, games that we discus can be defined as multi-agent environments, where players' moves are unpredictable and change as their goals are usually conflicting. As described in [23] a game can be formally defined as a search problem with few characteristics. The initial state represents the positions of each player on the playing board. Because of the turn-taking nature of discussed games, only one player has to perform its move each turn. All the available legal moves are known to the player, and the set is determined by the position of other players on the board. It defines a transition model from this state. Each move changes the world state and tests to see if the world state is terminal. Terminal states define end of the game. A function to evaluate a terminal state is called a utility function. It numerically represents the outcome of game. In some cases it just identifies the winner for example as -1 for agent number 1 ,0 for a draw and 1 as a win for agent 2, but often it also indicates the final score.

Each game satisfying the definition can be represented using a game tree. The initial state defines a root node, a transition model represents all the child nodes of root and for each of them recursively applies a transition model. Recursion ends by definition in terminal nodes. The game tree represents all the possible states of the game, thus the complexity of a searched problem for the optimal strategy can be calculated. It leads to the lowest cost, or in other representation to the highest score. In chess the average branching factor is 35, so investigating 5 moves ahead would require an evaluation of about 50 million states [23]. An even bigger branching factor of 361 is in GO.

A move performed by one of the players is called a ply. Usually there is a time constraint for each move or for the whole game and evaluating the entire game tree in such cases is not possible and so decision of the player is not optimal in most cases. The result is based on an approximation of world state evaluation.

```
function MINIMAX-DECISION(state) returns an action
    $v \leftarrow$ MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value $v$

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow -\infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s$))
    return $v$

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    $v \leftarrow \infty$
    for $a, s$ in SUCCESSORS(state) do
        $v \leftarrow$ MIN($v$, MAX-VALUE($s$))
    return $v$
```

Figure 7. Minimax algorithm [23]

4.3 MiniMax algorithm

Zero-sum games are defined as games where the total amount of points owned by all players in the game is 0. Usually in two player zero-sum games, players are called MIN and MAX. Because of the conflicting goals of the players, each of them tries to reach different terminal nodes within the game tree. With this assumption, each player chooses an action that would assure maximum benefit, and minimize the chance of losing. Assuming this, one has to always take into consideration optimal play from an opponent. For each state in the game tree, a minimax value can be defined. Expected optimal play dictates that the first player choose the move with maximum minimax value from the child nodes of its current state to maximize the gain (MAX). The second player wants to minimize MAX payoff (MIN), to maximize his benefit, therefore he chooses a minimum from minimax values of his child nodes. The minimax algorithm as defined in [23] recursively computes for each node in the search tree a minimax value as illustrated on Figure 7.

The recursive definition of algorithm traverse whole tree down to terminal nodes. An utility function is used to evaluate leaf nodes and to deliver their minimax value. This value is then propagated up the tree. Altering MAX and MIN moves changes the minimax values of nodes from leaves towards the initial state. Choosing the child node where the highest minimax value came from determines the optimal strategy for MAX. The value is a guaranteed minimum for MAX if MIN plays optimally. In the case of a suboptimal play by MIN, MAX can reach even higher value. Branching factor is determined by a number of legal moves for player from a position. Based on the branching factor, defining the width of game tree, the complexity of game is determined. In some cases there are time limitations to provide approximate results where terminal nodes are not reachable in reasonable time. For those purposes, the look-ahead value restricts maximum number of plies examined by algorithm. In the case terminal nodes are deeper in the tree than the look-ahead value, approximation of minimax value, provided by evaluation function is taken into account. This algorithm is optimal against an opponent employing optimal strategy.

The minimax algorithm is based on the depth first search (DFS). Given the maximum depth or the look-ahead value M and the number of moves for each player that defines a branching factor of the tree B, the time complexity of the minimax algorithm is $O(B^M)$. Algorithm can be implemented in different ways with various space complexity. The version that generates and stores all of the child nodes for position at once has space complexity O(BM) while the version that generates one successor at the time has O(M).

4.3.1 Optimizations and heuristics

Exponential time complexity of evaluating the whole game tree is not suitable for games where time per move is limited. There are optimizations and heuristic techniques that may decrease the number of examined states without modifying the overall result. One of the examples is alpha-beta pruning. The main idea of alpha-beta pruning is to prune unevaluated branches of the tree that cannot change the final value in the root. Due to the DFS nature of minimax, if there is an unexamined node x during evaluation and the algorithm has already found a better choice somewhere in the parent or up in the hierarchy, node x may be excluded from the evaluation since it never influences the parents' minimax value. This comes at a cost, however, as an improvement in execution time is needed to maintain two parameters - alpha and beta - that define the boundaries of examined nodes. A detailed description of this algorithm can be found in [23].

The order of examined child nodes determines the improvement of the approach. To define useful ordering rules, substantial knowledge about the searched game is required. The heuristics such as the previously best move first, may be also applied. As discussed in [23], time complexity of a minimax algorithm using alpha-beta pruning is reduces to $O(B^{(M/2)})$. In an average case, without applying special ordering, complexity is reduced to $O(B^{(3M/4)})$.

Identical states in the game tree usually appear more than once. Information about already evaluated states may be stored in transposition tables. This increases the space, but may reduce the time complexity. In this case, space complexity grows with each distinct examined state and can lead to exponential growth of search space.

As briefly discussed, different domain specific evaluation functions are used to estimate the expected utility value. Evaluation functions, in general, are replacing branches of the tree with estimated values once they pass a cutoff test. States for those branches are replaced by a terminal state with an estimated value. Tests may be defined according to the look-ahead value if there are no more time/resources to evaluate the next level or based on game specific features of the state. Even minimax with alpha-beta pruning can evaluate only a limited depth of around 5 to 10 due to the exponential growth in number of states [24]. Proper evaluation functions based on detailed domain knowledge decrease the size of the search space significantly. They provide values that doesn't influence the decision in negative way. More details about different evaluation functions and learning techniques are described in [23, 24].

4.4. Approaches to solve well known games

We discuss well known games with attention to the way how they have been solved, or the solutions that gave good approximation of the result. Strategies applied, are usually game specific, but some common approaches can be found. Their properties are discussed further. The analysis serves as a starting point for examined use case study.

4.4.1 Ultra-weakly, weekly and strongly solved games

To solve a game means, by the definition provided in [25], to find a game-theoretical value for each game position. Based on the definitions provided in [26,27] a game can be solved on three different levels;

1. Ultra-weak – For a given starting position, the result of perfect play is known, but the proof is not conclusive. A strategy for a perfect play does not have to be provided. An example of such a game is Hex. It can be shown by the "strategy-stealing" argument described in [28], that the first player has a winning strategy, but no winning strategy is currently known.

2. Weakly Solved - a strategy is known to achieve the maximum game-theoretic value of the game from the initial position for all players under reasonable resources. Most of the well known games have been solved at this level. Examples are Go-Moku [29], Nine-Men's Morris [30] and Checkers [31]. Resource limitation is substantial and limits impractical solutions that would need too much time to provide a solution.

3. Strongly solved – For all possible legal positions on the board, the game-theoretical value and strategy is known for all players, using reasonable resources. Strongly solved games are for example Awari [32], Kalah [33] and Connect Four [34].

This thesis evaluates mainly strongly and weakly solved games and the techniques that were used to solve them. Exact approaches used to solve games follow, with a description of techniques used. Specific implementations of representative examples of games are discussed in Chapter 5, with special attention to parallel solutions and the possibility to use GPU to accelerate the computation.

4.4.2 Tree search techniques

Out of all techniques used to solve well known games, game tree search algorithms play major role. As defined in [31] we can distinguish two types of tree search: Forward search and Backward search. The first processes the game tree from the starting position towards terminal nodes and gradually expands the tree depth until subtrees are solved. This is a straight-forward top to bottom approach. Exhaustive search methods like alpha-beta, and its variants, help to reduce the searched space. However, they are very time consuming as evaluation exponentially increases the state space size. A backward search, on the other hand, starts in the terminal nodes and evaluates their terminal values. From this point it searches for all positions that lead to the evaluated terminal nodes. Minimax values for the nodes that directly precede the terminal nodes are calculated. Minimax values calculated in this manner are propagated until the initial node is reached. All of them are stored and this creates the end game database. Retrograde analysis is the most commonly used representative of a backward search. This approach is has practical use for games where the search space converges like in Chess, Othello, Awari and Checkers.

In a forward tree search, situations emerge when a different combination of moves leads to the same state. As described above, transposition tables may be used to optimize the performance and cut-off sub-trees where the game's theoretical value is already known. It is a trade-off between memory usage and computational complexity. A time management strategy called iterative deepening was introduced.

4.4.3 Solved games

One of the best known games, tic-tac-toe, is a representative of a connection game, where the goal is to connect predefined number of pieces on the playing board with a straight line. As discussed in [35] these kinds of games have too many terminal states so backward search is not feasible. The game Connect Four was proven to be winning for the first player [27, 36]. Forward search techniques were used in a combination of alpha-beta search with a transposition table and move ordering heuristics (killer move heuristic [37], conspiracy-number search [38]). The result is that the first player has to start in the middle column to win. An extension of tic-tac-toe is called go-moke and it is also known as 5-in-a-row. There are many modifications that define the number of pieces in a row, board size, or defining restrictions for moves. Techniques used to evaluate such games in [26] are forward search with move ordering heuristics such as thread -based search, best first search and dependency based search. Free-style go-moku starting in an initial state was proven [26] to be winning for the first player in 18 moves against an ideal opponent. In Renju without opening rules, it was proven [39] that the first player wins by using iterative deepening, transposition tables and a dependency based search [26].

Different kind of games like Kalah and Awari belong to the Mancala family of games. The playing board is represented as set of holes and seeds in them. Each player in a move takes all seeds from an allowed hole and places them into subsequent holes, one seed per hole. A player can capture seeds at the end of his turn based on specific rules. For example, the last placed seed was third in the hole. Approaches described in [33] solve most of the variations of Kalah while retrograde analysis was used to pre-compute end game databases. Program created can solve several starting configurations up to six holes and 5 counters per hole, by usage of iterative deepening, and other search enhancement techniques like move ordering, transposition tables and Futility pruning. Awari was solved in [32] using retrograde analysis and a computing score of 889,063,398,406 positions. In combination with forward search it was shown that the result is a draw. Since an enormous database was created to contain all the positions that can occur in the game, Awari was solved strongly. Nine Men's Morris was solved in a similar way. The solution presented in [30] proved that the result is a draw. Forward search and retrograde analysis were used where the end game database holds 7,673,759,269 reachable states.

As described above, many games were strongly solved. Approaches presented in most cases consist of a combination of forward search and a precomputed end-game database using retrograde analysis. Usage of alpha-beta search in combination with different move ordering techniques and transposition tables helped to decrease the search space.

4.4. Parallel approaches on CPU based systems

There are many attempts to provide parallel applications that solve adversarial search problems. Many of them have successfully improved the performance of sequential approaches. Parallelism was used in different ways. Tree decomposition, identification and assignment of sub-tasks performed in parallel differ per approach. These are defined by capabilities of the hardware and the application that was running. Basic problems tackled are tree decomposition, computation complexity, synchronization and communication.

4.4.1 Static parallel evaluation

The evaluation of the static independent sub-parts of an adversarial search can be performed in parallel. Generation of moves or computation of the evaluation function were implemented as parallel processes. An example is Cray Blitz, a chess program that uses vector processing. Another chess machine Deep Thought uses special hardware for parallel generation of all legal moves for all positions. Details on the implementation and hardware can be found in [40, 41]. The evaluation function can be very computationally intensive, so an application can benefit from its parallelization. The speedup is limited by the degree of parallelism in move generation and evaluation. However, domain specific implementations show promising results.

4.4.2 Tree decomposition

The transition between static parallel evaluation and real game tree decomposition is an approach called Parallel search window. This method performs a game tree search on the whole tree using every available processor. Each iteration is performed with a different cost bound (window) [42].

Hierarchical node assignment to processors

An algorithm defined in [43, 44] decomposes the tree structure statically into disjointed parts and searches them in parallel. All of the processors are arranged in a tree structure and are then mapped to the searched tree. The inner nodes of the processor tree generates successors for positions assigned to them and propagates them to be evaluated by successive processors. Leaf processors perform a sequential minimax and return calculated values to their parents in the processor tree. After that, a new sub-problem from the queue of its parent is assigned to them if it exists.

Generate and compute in parallel

The algorithm presented in [40] distinguishes two types of processors. The first type searches the tree up to a predefined depth d. All positions found are then divided into two groups. The first group contains only nodes of the minimal game tree, while the second contains the rest. The first group is then processed on parallel processors by a sequential algorithm to a depth d. The result is returned to the master processor and then the second group is enqueued for processing.

Central control approach

An example of centralized control approach is the Principal Variation (PV) splitting algorithm. It can be described as a global synchronization model, since the application logic is centralized in one processor that takes care of load balancing. The algorithm takes advantage of move ordering heuristics and performs best on well ordered game trees. The ordering of moves in combination with cutoffs is used to decrease the size of the search space. To perform cutoff, the leftmost part of the tree has to be evaluated first. The idea is to use all processors to search in parallel the left part and only after retrieving boundaries for cutoffs is the right part searched in parallel. A process is performing sequential version of minimax algorithm (e.g. alpha-beta). One of the improvements of this approach is enhanced dynamic load balancing, since the trees searched in parallel are not of a uniform size. Idle processors may help busy ones by sharing their tasks.

Client-server approach

Young brother wait concept (YBWC) is a representative of the client-server or master-slave model of parallelism in a game tree search. The main idea is that the boundary has to be obtained first for the leftmost branch and then all the other branches can be examined in parallel. This concept defines the highest split node as the node for that value of which the first branch is known and its height is minimal. The algorithm assigns the first processor to the root of the tree and starts searching the tree usually using an alpha-beta algorithm. The list of split points is maintained and if some processor is idle, the branch from the highest split point is assigned to it. A processor starts searching the subtree it owns and maintains its split point information for the purpose of load balancing. Once the branch is examined, the processor returns the value to the server that assigned it. YBWC is described in further detail in [45].

Peer-to-peer approach

An approach called Dynamic tree splitting described in [46] is one of the more complex algorithms. None of the processors own any node and the processor that finishes last on the split point reports a calculated value to the parent of the split point. Classification of nodes and criteria for choosing split points are detailed in [46]. During the algorithm, an idle processor chooses the split point with the highest priority from the global list. Once the work is finished, the processor updates bounds. The algorithm was specifically designed for a shared-memory, multiprocessor architecture where the goal was to optimize load balancing and it is used in several chess programs such as Cray Blitz.

4.5 Summary

The parallel algorithms described above were proved to gain a substantial performance increase over sequential versions of adversarial search algorithms. Each of them approaches the domain in its specific way and is trying to utilize all resources available in the system it is implemented in. They addresses architectural issues such as the effective use of all processors and load balancing by proper decomposition of the searched tree that are best suited to the processing power of the available hardware. Communication and synchronization issues influencing the performance of algorithms are mainly determined by memory architecture. Accessing shared memory or maintaining a distributed model is distinguishes the algorithms and their usability of use on certain machine architectures.

5. GPU accelerated adversarial search algorithms

In the following chapter we present implementations of algorithms used to solve adversarial search problems. Special attention is given to parallel algorithms described in chapter 4. Their fit on GPU is discussed and the most promising ones are implemented. The researched algorithms were tested on different configurations of synthetic game trees and the speedup are presented on an implementation of the game Fox and Hounds. A comparison of benchmarks obtained from both sequential and parallel algorithms is presented. The suitability of GPU to accelerate adversarial search algorithms is discussed and illustrated on benchmarks at the end of the chapter.

5.1 Testbed description

All of the measurements were performed on two different configurations of hardware. The first was a Macbook Pro notebook. The CPU configuration for this model is 2.0GHz quad-core Intel Core 2 Duo T9400 i7 processor with 6MB shared L3 cache and 4GB (two 2GB SO-DIMMs) of 1333MHz DDR3 memory. The graphical card was a NVIDIA GeForce 9600M GT that featured 32 stream processors and 256 MB GDDR3 memory of its own that contained 314 million transistors. We refer to this configuration as "standard". More advanced hardware was used in second configuration. The graphical card used was a NVIDIA GeForce GTX 480 with 480 CUDA cores with 1536 MB GDDR5 containing 3 billion transistors. This GPU is based on Fermi architecture that was described briefly in Chapter 2 while this configuration is referred to henceforth as "advanced".

5.2 Sequential algorithms

In general, most of the sequential approaches described in Chapter 4 are based on the minimax algorithm. A standard tree search, when the terminal nodes are not reachable within a reasonable amount of time and evaluation functions are used, consists of several phases [41].

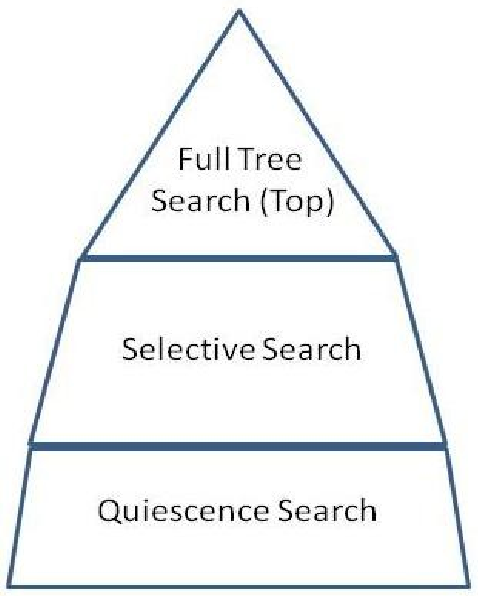


Figure 8. Tree search phases

In the first phase, as displayed in Figure 8, at the top of the searched tree, all the nodes are visited. Usually there are no cutoffs in this stage and tree is searched completely. Once the required depth is reached, a selective search phase is initiated and only the interesting nodes are visited and investigated further. The definition of interesting is usually domain specific and is based on certain compositions of board figures or boundaries of the evaluation function. A selective search usually also has a predefined maximal depth. The third phase is called the quiescence search. The principle of it is very similar to a selective search in that it evaluates into more depth only positions, such as the ones with direct threat, that are marked as noisy.. A search is then performed until a noisy position changes to quiet one, and this improves a position's minimax value by emulating moves that significantly influence its evaluation.

As described in the previous chapter, a reduction in the number of searched positions can be achieved by using endgame databases. They often contain evaluated positions of a certain type. For example, all positions for a number of pieces on the board are evaluated. Retrograde analysis is used to emulate the searched tree traversal backwards from the terminal nodes. Parallel implementation of this preprocessed database proved suitability of it's practical use while solving the games Awari [32].

5.1.1 Sequential algorithm setup

All implemented algorithms were compared based on the total time elapsed. At first, algorithms were tested on two player board game. We investigate adversarial search algorithm on synthetically generated game tree. The game that takes place is a standard chess board where only black squares are used. Each player has a configurable number of pieces. Players alternate their moves, each with one piece per move. Pieces can move one square in any diagonal direction in case square is free and chess board limits are not violated. The result of the evaluation function defines leaf value distribution that influences the strength of the ordering heuristics. Uniformly distributed trees are used here, so that every successor has the same probability to be the best one. Trees constructed in this way provide a good approximation for real life situations and are usually referred to as synthetic trees, frequently used in the literature [51]. Multiple measurements were taken with a different combination of parameters like number of pieces and search depth.

Based on the division of the tree search into the three phases described earlier, we can recognize three possible scenarios. The first is of particular importance, later referred as the "top", where the search is performed over all of the nodes. For a given position, using the minimax algorithm a whole tree is generated. The search stops at a predefined depth where the evaluation function can be used to determine the value of a position.

We are considering selective and quiescence search phases as separate scenarios. We assume that those phases of the search are performed separately, without top phase. We have *n* nodes at a generated depth *d* and for those, a selective or quiescence search is performed. This means that we have to traverse *n* trees with branching factor *b*. In a selective search we limit the number of branches examined by picking the most promising ones. For the purpose of generality, a restrictive definition to perform a search on the best few moves from each node is provided and search ends in predefined depth. A quiescence search is performed on all branches until the branch is no longer stable. This means that the function to evaluate stability of a position has to be applied to every new successor of a position in the searched tree. Restriction is defined for a maximal depth of quiescence search.

Benchmarks for a sequential approach were measured using a standard, depth-first, search-based, minimax algorithm in non recursive implementations with respect to minimize memory requirements and processing speed. Data structures used and description of algorithm are described further in the text.

5.1.2 Evaluation function complexity

The evaluation function takes into consideration many aspects of the game. It is usually created by game experts, evaluates the current position and determines its minimax value statically. In many board games, including chess, the static function is constructed as a weighted sum of various factors. An example of game with complex definition of evaluated factors is chess. According to [47] we can recognize seven main factors that influence the static value of position: Material, King Safety, Piece Mobility, Pawn Structure, Space, The Center, Threats. Each of them has a different complexity to calculate. Material is usually defined as the sum of the values of pieces for each player on the board. Valuation of material takes into account the number of pieces on the board. Mobility is defined as the number of legal moves each player can make with all the pieces on the board. The complexity to calculate this factor is influenced by the number of pieces on the board and the number of their legal moves. Board control factor is defined in chess as the number of squares controlled by each player. A square is controlled by player once it has more pieces attacking it than the opponent's. Controlled squares in chess carry different weights - those in the middle of the board are fundamental for opening up the game. Of course, examples of the factors described here have different relevance per game. Mobility is, for example less important in chess than it is for Othello and board control is a crucial aspect for Go. From the examples described above it is clear that the complexity in calculating factors that influence the evaluation function differs per game.

5.1.3 Rules of tested game Fox and Hounds

The second tested implementation is the game Fox and Hounds. The rules of the game, as described in [48], define that the game is to be played on a standard 8x8 chess board where only the black squares are used. Four hounds are initially placed on the dark squares at once edge of the board while the fox is placed on any dark square on the opposite side of the board. Initial positions of pieces are shown on Figure 9.

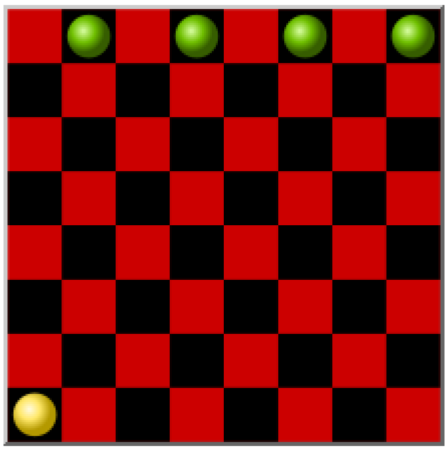


Figure 9. Initial position of Fox and Hounds board game [48]

The objective of the fox is to cross the board from its initial position to the opposite end of board, to the initial position of any hound. The hounds' objective is to prevent this by moving only diagonally forward while the fox moves are also allowed backwards. Hounds' moves are like a man and fox like a king in draughts. No jumping, promotion, or removal of pieces is permitted. Each player has to move with exactly one piece per turn. There are two types of terminal positions in the game: the first is a victory for hounds when the fox is trapped and it does not have a legal move. The fox wins by evading the hounds and making its way to one of the hounds' initial positions.

The game has been proven in [49] to be winning for hounds from an initial position allowing a choice of any of the four starting positions for the fox. We have obtained benchmarks for the implementation of Fox and Hounds. Game positions are represented as positions of all five pieces on the board. Positions on the board are numbered from the top-left corner starting from first black square as 0 until 31 at the bottom-right. As shown in Figure 10, a position in this scenario would be represented as [17,8,5,10,7]. The first number always represents the position of Fox with the other four numbers marking the positions of the hounds. The second number is the hound that started at position 0, ..., and the fifth number is hound that started at position 3. Obviously two pieces cannot occupy the same position at a given time.

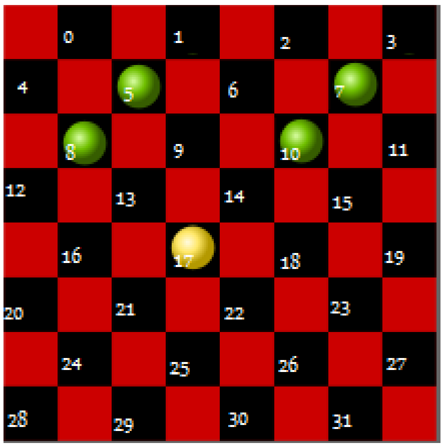


Figure 10. Fox and Hounds game representation

The program calculates the minimax value for a given position by performing a depth-first search based the minimax algorithm in non-recursive form. Algorithm with same structure and objects is used in case of synthetic game trees. High level description of algorithm can be represented as:

```
while (!done){
    if( level is max or no new move was generated ){
        decrease level
        if (terminal position or level is max) => evaluate_function(position)
        else if (level == -1) return results[0];
        else {
            store minimax value for this node as results[level+1]
        }
        calculate min or max based on level and stores to results[level]
        return_back_to_parent_position
        }
    }
    try_to_generate_new_move(position, movement_history)
    if(new move generated){
        increase level
    }
}
```

The state of algorithm is maintained in stack called *movement_history* in representation of algorithm above. To reduce memory requirements, and by that the space complexity, the stack contains a numerical representation of piece and chosen directions on each tree level. Each direction for each piece is represented by a number. Figure's id is stored on the positions of decimals and direction of it's move on position of units. With a combination of actual positions (five numbers), it gives a representation of the actual search state and information how this state can be achieved from initial position. The memory requirements are to store five numerical representations of the actual state and one number for each level up to predefined look-ahead. The look-ahead represents the maximum number of plies *max*. Movement back from a position in the tree is represented as negative value of number used to get into the position. To store the temporary minimax values for each level, an array *results* of length of *max* is used. Once the algorithm reaches a predefined *max* level and no terminal position is reached, the evaluation function *evaluate_function* is used to determine a minimax value for that node. The function calculates the number of reachable positions for the fox. It is implemented as a breadth-first search algorithm. The algorithm requires to store already visited squares. For this purpose a specific array is used.

Similarly to algorithms on artificial trees, several scenarios were tested. The First one (FnH1) is a basic algorithm that evaluates one initial position. The Second scenario (FnH2), to simulate quiescence, selective search phase and retrograde analysis, where as an input *n* different positions are provided. A third benchmark is created on a setup where the evaluation function is calculated for *n* positions. All the benchmarks are later evaluated and compared with the parallel implementation on GPU.

5.2 Applying GPU architecture

In this section we evaluate the suitability of the aforementioned algorithms to be implemented on GPU architecture. We try to identify and evaluate the attributes of algorithms that would indicate their suitability for acceleration using GPU. Afterwards, we propose an implementation of a test scenario that is used in both standard and advanced hardware configurations to retrieve benchmarks and further be compared with sequential approach.

5.2.1 Investigated attributes

An introduction to GPU programming and a description of the architecture are presented in Chapter 3. Specifics of the execution model that determine the type of parallel processing must be taken into account while determining the overall algorithm suitability. The number of threads that run in parallel on GPU is huge. To maximise this potential, we have to be able to identify parts of the evaluated algorithms that can be performed in parallel with respect to specifics of GPU architecture. Most of the algorithms described above are typically recursive. Their non-recursive implementations are requiring an extra memory maintenance. Missing recursion inside kernels also leads to substantial algorithmic overhead.

Synchronization defined only within a given work-group is limiting and so we have to try to identify the parts of the algorithm with the most scope for parallelization. The sequential parts of algorithms are also limited by system requirements, which must be be taken into account.

Awareness of the GPU's memory model is very important. Once identifying a suitable way for parallelism, there are certain restrictions on each kernel that may be crucial. Restricted local memory space and expensive global memory access define the boundaries for the data types used in algorithms running in kernels. Since copying from host is considered expensive, this means that the complexity of computation in a kernel has to be worth the copying overhead as restrictions on granularity of kernel programs indicate. Non-coalesced memory access causes substantial delays and decreases the overall performance. This can also be due to different execution paths within kernels. Restrictions and recommendations for GPU architecture must be taken into account in the analysis phase to determine the best-fit for the algorithm or its part for GPU suitable parallel processing while still benefiting from its SIMD nature.

5.2.2 Analysis and design of benchmark scenarios

Generation of all possible moves or the next move is the base of game tree generation. In practice static parallel evaluation was used to generate moves in chess using special hardware. The idea of generating moves for one position is not suitable for processing on GPU for several reasons. The number of threads that are initiated is very low and since the branching factor of chess is 35, only 35 threads would be needed for that game. A second scenario is that we would generate more moves (more plies) for more initial positions. This would theoretically occupy enough threads to fully utilize GPU, but the ratio of data transfer operations to the complexity of calculations per kernel may not be beneficial. As an input data, information about all pieces on the board has to be transferred from host to device, and each kernel generates a subtree for a predefined number of plies and for each position check if it is legal.

Static parallel evaluation

A static evaluation can be performed in parallel on evaluation functions for different leaf nodes. As described in the overview of sequential algorithms, the evaluation functions may have different complexity depending on the type of game. Complexity of evaluation functions usually depends on the number of pieces or the number of squares on the board. More complex game specific evaluations that detect patterns of position of pieces on the board are out of scope. For example in [50], the evaluation function of the game Awari consists of 12 features with different weights that influence the final function value. All of the features take into account the number of pits with a special number of seeds. We consider a scenario where the evaluation function is applied to multiple positions in parallel. Since the input is a board status and the result is in fact one number per position, data transfer delay should not harm the overall performance.

Parallel window search

In a parallel window search as described in [42], the total range of values in a game tree is divided into p (the number of processors) ranges that do not overlap. All ranges are examined in parallel for the whole tree. If we want to utilize the maximum number of processors available in GPU it would cause ranges to be very narrow. A lot of cutoffs would be performed in the early stage of processing, then they would be idle and have to wait until the others finish. Limited local memory for work-items also causes problems since it defines a boundary for the number of plies each kernel is able to examine. Execution paths per kernel differ and non-coalesced access to memory negatively influences the performance.

Hierarchical node assignment, combined CPU, GPU approach

A method that maps processors into tree structures is not suitable for GPU. Although we have a lot of processors available, synchronization between them is restricted to the work-groups level. The type of synchronization required by this approach would be very difficult to maintain within GPU. Synchronization of work-items on GPU device is restricted due to SIMD architecture. Taking into consideration an approach that generates nodes to a depth d and then runs all nodes in parallel suits our architecture very well. So instead of assigning work to kernels by other kernels, a straight-forward approach is where the host defines independent SIMD or actually STMD to be processed by the device. A tested scenario generates nodes in the tree until predefined depth. It must be deep enough to generate an appropriate number of nodes to run in parallel and utilize the whole potential of the available GPU. Once the nodes are generated, the parallel part of the evaluation starts where the minimax algorithm is initiated on GPU per each kernel. A basic version of minimax is implemented without cutoffs to keep execution paths the same. Due to memory restrictions, a non-recursive version of minimax as described in the sequential algorithms section is implemented. Several versions of each algorithm are tested. and tests are performed for variable values of parameters: the number of inputs for parallel processing, number of pieces, depth of game tree generated in each kernel. We refer to this scenario as *generate_and_parallel_minimax*.

Quiescence and selective search in parallel

To get a comparable scenario with a sequential version of a selective search, a branching factor in tests are restricted. We compare results from this benchmark, *parallel_selective_search*, with results from a sequential selective search. A similar approach is used in the case of the quiescence search whereby the algorithm evaluates stability for every generated position. Since the stability function has similar properties to the evaluation function, its complexity is also one of the parameters. This scenario is called *parallel_quiescence_search* and it has all of the previously mentioned parameters including depth and number of inputs.

Principal variation, young brother wait concept, dynamic tree splitting

The essence of the principal variation splitting algorithm as a representative method with centralized control indicates a few complications. Splitting the tree into sequential parts usually produces not too many trees, with different height and branching factor. This method may not be able to utilize the relatively high number of processors available on GPU. The structure of branches searched in parallel is irregular and PV in CPU implementation lacks any load balancing between processors. This algorithm is effective only when the leftmost branch provides good bounds so cutting off some of the right branches is possible. The synchronization overhead is large and the speed overhead is usually huge if the tee is not well-ordered. Due to an insufficient level of parallelism and the irregular structure of branches examined in parallel, we are not investigating this method further. The irregularity of branching factor and subtree height would cause a high overhead because of non-coalesced access to memory. Many of the processors would become idle too soon while a few may hit the boundary of local memory while examining large trees. Division of work is not suitable for the SIMD nature of parallel processing on GPU. Although a similar, more uniform version of this scenario without pruning has already been tested in *generate_and_parallel_minimax*.

Peer-to-peer approach, dynamic tree splitting, as the enhanced version of Young Brother Wait Concept, requires overly complex coordination in synchronizing work to be carried out at runtime. Data parallelism of GPU is not suitable for this level of self-load balancing between processors. In other words, processing elements are scheduled by the device to perform a job and only after all of them finish can they be reused. This is why it is essential to assign the same job to all the kernels and to follow the SIMD paradigm of GPU.

5.3 Analysis of empirical results

In this section we present measured benchmarks. Empirical results directly prove the suitability of a GPU processor to accelerate the described representative scenarios. Each tested scenario is discussed with respect to its performance and gained speedup while comparing sequential and parallel implementation. First, we examine algorithms on synthetic game trees simulating a two player game on a chess board. Results of measurements are then backed up with benchmarks obtained from an implementation of the game Fox and Hounds. On this application we practically prove the suitability of adversarial search algorithms for parallel processing with GPU. The benchmarks disclosed several limitations of GPU usage.

5.3.1 Parallel adversarial search on synthetic trees

A synthetic game tree is created for all scenarios based on the configuration of the number of pieces per player and look-ahead. Speedup is defined as the ratio between the time of sequential algorithm running on CPU to the time of parallel algorithm on GPU running against the same problem.

Minimax search		Total look-ahead		
Total number of pieces		4		8
	Look-ahead parallel	14		8
CPU (ms)		4067		33167
GPU Standard (ms)	6	2533	2	5473
	8	5409	4	8506
Speedup factor	6	1.61	2	6.06
	8	0.75	4	3.90

Table 2. Minimax search on artificial board game on standard hardware setup with 4 and 8 pieces on the board

Hierarchical node assignment, combined CPU, GPU approach

The first examined scenario is *generate_and_parallel_minimax.* As described above, based on one initial position, the algorithm performs a minimax search on the tree until a predefined depth, after which a minimax value is returned. Two basic setups are examined for this scenario. In the first we examined a game with 4 pieces (2 per player). Since each piece can move in 4 diagonal directions, the maximal branching factor per player move is 8. In the second scenario there are 8 pieces on the board. The first player has 2 and second 6, so this scenario generates many more possible positions. The algorithm that uses GPU to accelerate its processing consists of three parts. In the first part, an in-host program algorithm generates the game tree up to a predefined depth that is sufficient as an input for the GPU accelerated parallel part. In the second, the GPU takes over and performs a minimax algorithm on each of the nodes generated by the first part. Minimax values for all nodes are then returned and in the third phase the host program finishes the search and returns a minimax value for each input position.

Number of pieces: 4		Look-ahead			
	Look-ahead parallel	14	16	18	20
CPU (ms)		4067	53309	647546	8036000
GPU Advanced (ms)	6	2332	4260	11061	N/A
	8	3231	5260	17566	N/A
	10	9175	21293	45021	227204
Speedup factor	6	1.74	12.51	58.54	N/A
	8	1.26	10.13	36.86	N/A
	10	0.44	2.50	14.38	35.37

Table 3. Minimax search on artificial board game on advanced hardware setup with 4 pieces on the board

Benchmarks measured with a standard hardware setup are shown in Table 2. Although in a standard hardware setup there are only 256MB memory on GPU, we were able to gain maximal speedups of 6 in the case of the 8 piece scenario and a look-ahead of 8. In the first scenario the speedup was only 1.61 but the look-ahead is 14 in proportion to time elapsed. The difference in look-ahead is caused by a different branching factor and memory requirements to store examined positions. An advanced setup shows much more interesting results from the perspective of speedup. We were able to evaluate the game tree up to depth of 20 with a speedup factor of 35.37 in the case of the first scenario with 4 pieces on the board. The first part of the algorithm has to produce enough input to fully utilize all processors of the GPU.

In second setup of the first scenario with 8 pieces on the board, a speedup factor of 82.91 was achieved with look-ahead of 10. N/A values in Table 4 with benchmarks of advanced hardware setup on scenario with 8 pieces are caused by memory constraints on the host side or by not having enough data to start parallel processing on GPU. In the case of the advanced setting it is 1024 inputs and in the case of standard it is 512. The range of data sizes are chosen in the way that they fit into GPU's memory at once. In this case there is no need to batch the same execution and encounter a longer delay caused by the host to device memory transfer. In the other case, we would have to schedule commands for data transfer and algorithm execution into the command queue multiple times. N/A values for example in Table 3 are there because of this restriction.

Number of pieces: 8		Total look-ahead	
	Look-ahead parallel	8	10
CPU (ms)		33167	2551000
GPU Advanced (ms)	2	2126	N/A
	4	3017	30769
	6	N/A	70305
Speedup factor	2	15.60	N/A
	4	10.99	82.91
	6	N/A	36.28

Table 4. Minimax search on artificial board game on advanced hardware setup with 8 pieces on the board

In tables 2 and 3 there are reported speedups of 0.75 and 0.44. As seen in both tables with an increasing number of look-ahead, the parallel speedup decreases, although with the overall look-ahead it increases. This observation proves that GPU as a SIMD processor can work in a better way with more data and simpler operations. A parallel look-ahead of 10, leaves only 4 to be executed by host. The size of the input to be performed in parallel is low and the computation on it is very complex. The paradigm of SIMD processing defines suitable input in the opposite way. There is an observable trend in all the results for this scenario towards setup with a lot of data produced by the host program and a lower number of parallel look-ahead, with less complex work done on GPU.

Another observation is that with a higher look-ahead for the second part, that should be executed in parallel, memory requirements for GPU increase. With more branches to examine, the structure of created trees differs further. Some of the branches, created by legal moves, have more nodes due to board limitations and the presence other pieces on the board. With more execution paths the application loses the advantage of fast coalesced memory access as described in Chapter 3 - OpenCL.

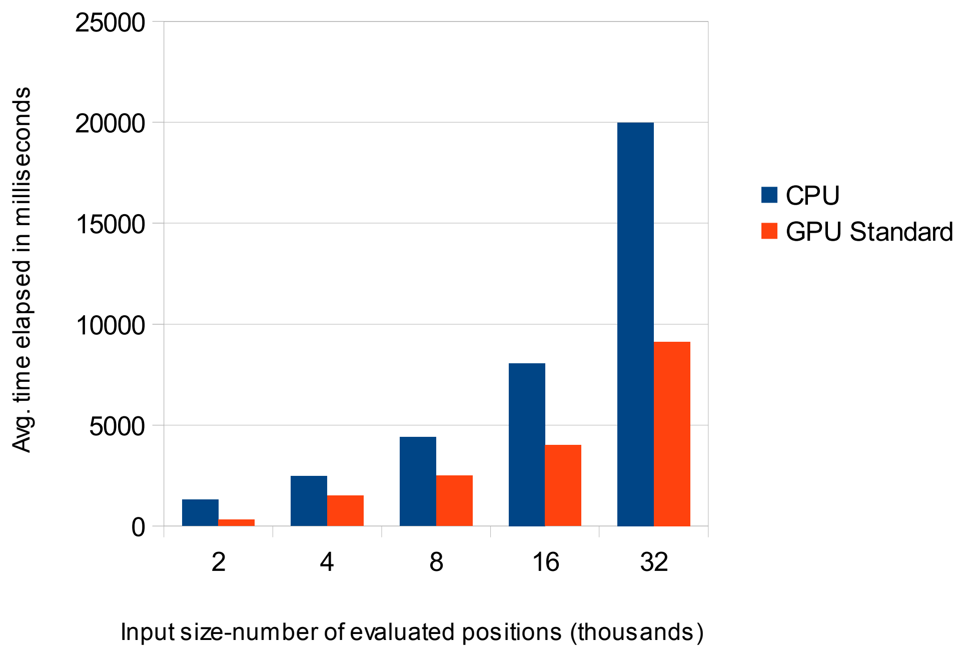


Figure 11. Selective search on artificial board game – standard hardware setup (lower is better)

Selective search in parallel

A second scenario examined is *parallel_selective_search.* In this scenario we have a measured speedup factor of the applied selective search on a generated game tree while selecting only moves that look interesting from a board position perspective. An input for this scenario marked as *Data size* defines the number of positions that should be examined further in this selective manner and the look-ahead value. In our test case, parameters for the chess board-based game are four pieces - two per each player. For each position on the chess board, only diagonal moves are examined. Gathered results on a standard hardware setup for a look-ahead of 6 are displayed in Figure 11. An average speedup factor of 2.03 was achieved.

The speedup factor of the selective search with an advanced hardware setting and with the same configuration is displayed in Table 5. The difference in hardware setups is clearly recognizable and the speed of processing is lower for much bigger input data sizes. Speedups of 122 while investigating millions of positions shows suitability of this kind of application for GPU.

Selective search	Data size				
	512*128	512*256	512*512	512*1024	512*2048
CPU (ms)	49556	122881	304704	755561	1873532
GPU Advanced (ms)	1424	2308	4166	7079	15389
Speedup factor	35	53	73	107	122

Table 5. Selective search on artificial board game on advanced hardware setup

Quiescence search in parallel

Similar results are achieved in an implementation of *parallel_quiescence_search.* With a standard hardware configuration we have gained a speedup factor of 3.59 as illustrated in Figure 12. The function that checks the stability of a position performs a measurement on the average distance of pieces with complexity O(board_size*number_of_pieces). In our case those were four pieces on a chess board. An evaluation function is applied on every position that is generated to check its stability. Evaluation of each position causes computational complexity of parallel search to increase, in case a lot of positions would look promising. Since a quiescence search has a maximal look-ahead, we limit our scenario to 6. This boundary was proven to be efficient in the *generate_and_parallel_minimax* scenario to balance the complexity of the algorithm executed in parallel and data complexity. As mentioned earlier, computation of the evaluation function is itself a suitable candidate for parallel processing. This suitability is examined further on an implementation of Fox and Hounds.

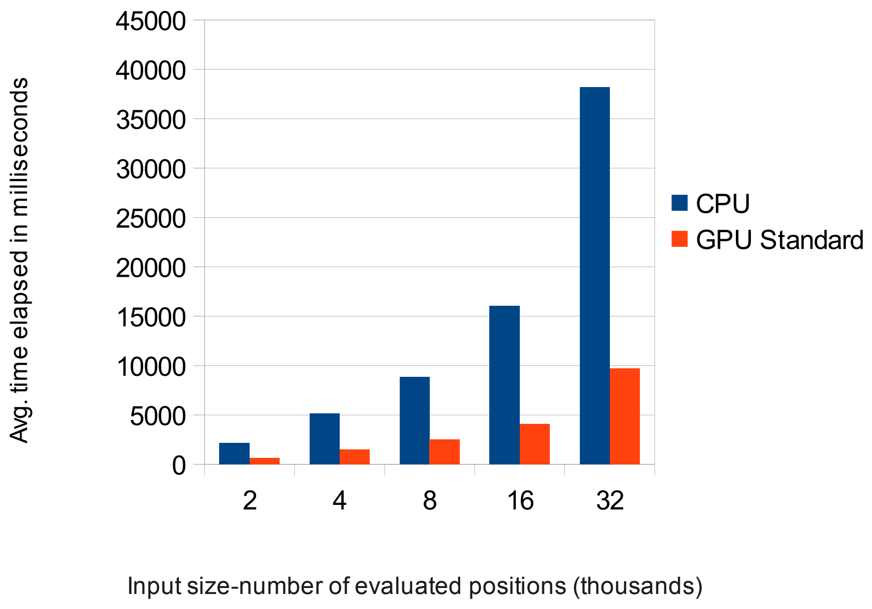


Figure 12. Quiescence search on artificial board game

– standard hardware setup (lower is better)

Benchmarks from a quiescence search performed on an advanced hardware setup show an almost linear time increase both on CPU and GPU. This linearity is caused by fewer cutoffs. Evaluation of position is done for every new position in this scenario to illustrate the stability of parallel processing speedup with respect to the amount of computation performed per position. As can be seen from Table 6, almost linear speedups per number of examined positions proves that performing more complex algorithms on the same set of data is suitable for GPU architecture and can deliver total speedups in ranges of around 115. As previously mentioned, while taking advantage of GPU parallelism we have to take into account the number of inputs that we have and the complexity of an algorithm that should be performed on top of this data. The overhead of slow copying of data via PCI Express from host to device has to be balanced against the speed achieved from data parallel execution of algorithms.

Quiescence search	Data size					
	512*128	512*256	512*512	512*1024	512*2048	512*4096
CPU (ms)	79963	167520	350950	735230	1540286	3226856
GPU Advanced (ms)	1335	2149	3728	7450	14284	28108
Speedup factor	60	78	94	99	108	115

Table 6. Speedup factor of quiescence search on artificial board game on advanced hardware setup

5.3.2 Benchmarks for tested game Fox and Hounds

Until now, we have presented very positive speedups gained on implementations of parallel algorithms on artificial game trees. The theoretical value of results shown above is described next, backed up by examining the same types of algorithms on a practical example, such as Fox and Hounds. Tested scenarios for this game prove that good speedups can be achieved also in practice. Benchmarks for the minimax game tree search for one position along with selective and quiescence searches are presented. In addition, measurements for static parallel evaluation are discussed in the case of the evaluation function for the game. In the next part, we present gathered empirical results.

Hierarchical node assignment, combined CPU, GPU approach

Interesting speedups were measured on scenario, with one input position, for the algorithm before described as FnH1. As displayed in Table 7, reported results ranging from 0.02 to 64.47 were seen. There are huge differences in speedups with each new ply. A speedup 0.2 of setup with a look-ahead of 8 where two are performed in parallel jump to a speedup factor of 8.2 with look-ahead of 10. This increase is caused by the CPU speed. In the first case 6 out of 8 are computed on the CPU. The CPU speed computes, on average, 14815 in 81.44 milliseconds based on the algorithm's log. The speedup drops with the increasing parallel look-ahead for a look-ahead of 8. With increasing parallel look-ahead, the amount of inputs decreases and that has negative effect on the speedup. This indicates that the number of inputs generated by the CPU is insufficient for the GPU. The overhead of copying the data is not compensated by the speedup factor of calculating minimax value for trees with a look-ahead of 2 to evaluate that position. The CPU performs better in such a case. In the setup with a speedup factor of 8.2, 299117 positions are created by the CPU. Such a range of data is suitable for standard hardware configurations to deliver the best results. With an increasing number of look-ahead in parallel and the same total look-ahead, the performance speedup decreases since less data is provided to the GPU while more complex computation is expected. From the same table, we can also see that those ranges of data are not suitable for an advanced hardware setup. Since the number of board positions generated by the CPU with a look-ahead of 6 is low, no speedup is gained for a look-ahead of 10. Interesting values are in the range of look-ahead 14 and 16. Speedups of 49.72 and 64.47 are significant.

Fox and Hounds (FnH1) speedup factors		Total look-ahead				
	Look-ahead parallel	8	10	12	14	16
GPU Standard (ms)	2	0.20	8.20	N/A	N/A	N/A
	4	0.17	3.05	3.64	N/A	N/A
	6	0.05	0.95	2.97	N/A	N/A
GPU Advanced (ms)	4	0.03	0.56	13.63	46.19	N/A
	6	0.03	0.61	14.68	49.72	64.47
	8	N/A	0.18	2.95	42.13	62.54
	10	N/A	N/A	0.28	3.41	51.25

Table 7. Speedup factor of game Fox and Hounds scenario FnH1

Static parallel evaluation

As discussed earlier, static parallel evaluation was used to speed up the evaluation function of chess positions. We examine the possible acceleration of speedup in the case of the evaluation function for Fox and Hounds. The evaluation function in this case searches for the given state of the board all legally accessible fields for fox. The implementation uses a BFS-based algorithm that visits all neighboring squares. Memory requirements are determined by an array to store previously visited squares and a storage for the evaluated position so the id of field for each piece on the board. Evaluated positions for pieces on the board were generated randomly.

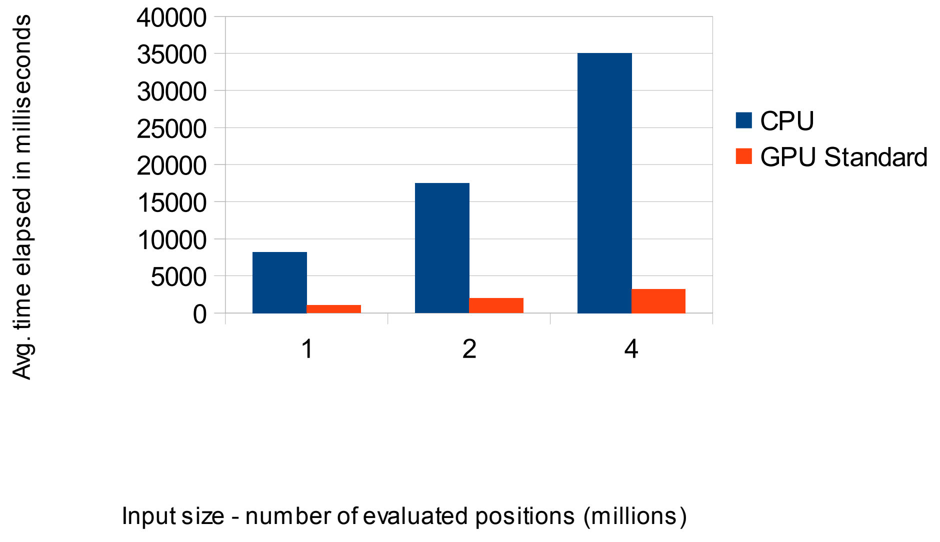


Figure 13. Evaluation function for Fox and Hounds

Comparison of CPU and standard GPU (lower is better)

Results for a standard hardware setup are displayed in Figure 13. Due to the high number of input positions and suitable complexity of algorithm, significant speedups were gained. Maximal acceleration is 10.95 in case of the largest data set. On advanced hardware we tested the evaluation static on data sets that were eight times larger. Results show speedups of 69.01. The data parallel nature of these kinds of sub-algorithms for solving adversarial search problems are very well suited for GPU architecture. Speedups of GPU implementation show an increasing trend with the amount of data evaluated. That is caused by higher utilization of parallel processors on GPU.

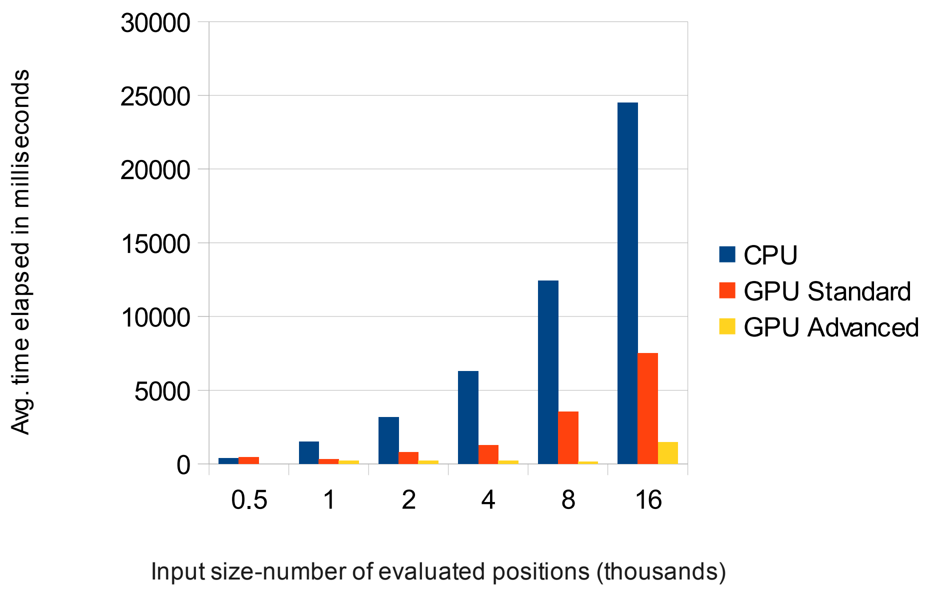


Figure 14. Selective search for Fox and Hounds

Comparison of CPU and GPU (lower is better)

Selective and quiescence search in parallel

Benchmarks for the implementation of selective and quiescence search for Fox and Hounds shows similar results. The memory requirements for both in both cases are restrictive on the size of look-ahead that we can achieve. To display a representative selection of benchmarks, scenarios were tested with a look-ahead of 4. The average speedup for a selective search is displayed in Figure 14 - for a standard GPU configuration this is 4.13. In the case of synthetic game trees, the average speedup obtained was only 2.3. One of the reasons for this increase is the complexity of the evaluation function. In the case of synthetic trees, the function takes into account the position of each piece and calculates distances between them. Fox and hounds evaluates the whole board with respect to positions of all the pieces on it. Additional memory usage and access for this purpose causes the CPU to be slower in computation of the value. Memory that has to be used to calculate the evaluation function is allocated in private memory for each work item. Fast access to local memory that is executed in parallel increases the speedup. Similar results are presented for a quiescence search in Figure 15.

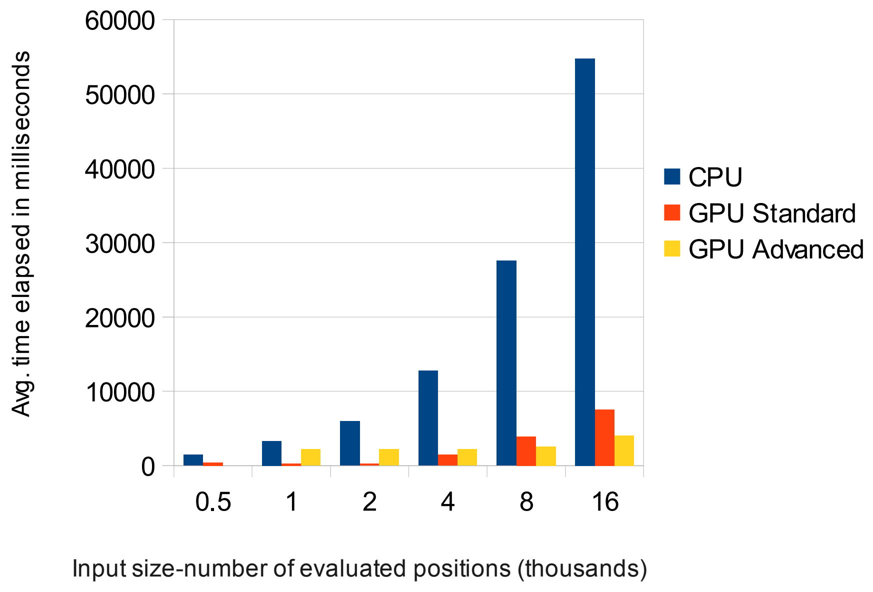


Figure 15. Quiescence search for Fox and Hounds
Comparison of CPU and GPU (lower is better)

Results of implemented scenarios for Fox and Hounds proved the suitability of GPU accelerated algorithms in the domain of adversarial search algorithms. High speedups were gained on synthetic game trees scenarios with a higher number of pieces on the board as well as in cases with a high look-ahead of around 20. Acceleration using GPU can be used in multiple ways. On the top levels of a tree, in combination with CPU as described in a *generate_and_parallel_minimax* scenario or in further stages to perform quiescence and selective search. Parallel static evaluation of the evaluation function was proven to deliver very promising results and can help to accelerate overall performance of more complex algorithms in this field.

5.4 Summary

Empirical results that we have presented in previous section prove suitability of GPU based acceleration in the field of adversarial search. Results showed significant speedups in all tested scenarios. Major differences in hardware specification of used GPUs are reflected in overall results. With standard hardware setup we achieved speedups factor of 6.06 in first tested scenario for 8 pieces on the board. Advanced setup reached speedup factor of 58.54 with look-ahead of 18. Very positive results achieved on artificial trees are supported by benchmarks of representative game Fox and Hounds where speedups of 10.01 and 64.47 are reached. First scenario can be described also as combined computational effort by CPU and GPU. Benchmarks show for lower look-ahead values significantly better CPU performance. Searches to depth of 8 in the trees with low branching factor does not provide enough inputs for GPU to take advantage of their parallel execution. The reason is already mentioned limiting factor in speed of data transfer between host and GPU device. This delay must be balanced by executing sufficiently large number of parallel processes. Once this limit is reached we were able to achieve significant speedups for this scenario.

While moving towards larger look-ahead values, we have observed increasing speedup for cases where look-ahead parallel value was lower. Better results for less workload in parallel on larger input data set can be rationalized by greater divergence in execution paths. Additional experiment was created to support this theory where selective search scenario was executed first with randomly generated positions of pieces and then with all the positions being the same. Since the order of move generation is predefined. In second setup accesses to memory were coalesced. In first, number of moves concurrently examined per position differs due to difference number of legal moves(board's boundary, other pieces). Average speedup was 4.38 for coalesced access on standard hardware setup. This shows one of the possible improvements that can be investigated further since average speedup factor of selective search was only 2.03. In case of advanced hardware setting, where huge amounts of data were examined in parallel, top speedup is 122. Amount and complexity of computations that CPU has to do in case of quiescence search can be enormous for this big data set. Speedup increased linearly with number of inputs reaching to acceleration factor of 115 on tested benchmarks.

Presented speedups on representative adversarial search algorithms proved fit for GPU acceleration in this domain. However, we have to keep in mind setups where CPU showed better results and adjust usage of resources to the specific requirements and properties of implemented problem. Merits and restrictions of both CPU and GPU has to be taken into account and combination of their use balanced to push performance limitations of specific application further.

Conclusion

Increasing need for processing power in many fields motivates to examine new technologies and different approaches to achieve it. GPU as a resource, that is no longer used only for displaying graphics shows promising possibilities to use. It represents widely accessible and cheap resource with enormous computational power. The power comes from its specific architecture supporting highly parallel computations. However, exploiting its full potential can be difficult.
It requires being acquainted with the differences in architecture and programming for CPU and GPU architectures. Only then we are able to optimally utilize both and more importantly, to choose the best architecture for the problem at hand.
We presented problems, limitations and comparison of each technology to provide a better understanding of the behaviour of AI algorithms on massively parallel architectures. OpenCL, an open standard for computing on heterogeneous platforms, served as an interface to perform the programming of GPU. Ability to uniformly access both platforms and combine their advantages allows us to improve performance of applications in many fields. GPUs with their extensive parallelism allow allow programmers to achieve impressive speedup factors.

The main focus in the described domain of adversarial search is on games. Games, in general, are of special interest since their definition is usually very simple, but solving them completely presents significant challenges and needs serious computational power. Complexity of the problems grows exponentially with the level of detail we are examining. Different algorithms were created over time, using sequential and parallel approaches to examine game trees. We have researched available algorithms nowadays and analyzed their fit for acceleration on GPUs. We applied theoretical knowledge about GPU architecture and OpenCL framework to provide implementation of selected adversarial search algorithms chosen in analysis phase. Tested scenarios covered different states of game tree evaluation.
All nodes at the top of the searched tree are visited on the CPU, however, selective search phase, in which only the interesting nodes are visited and investigated further, initiated at a certain depth and is run on the GPU. The evaluation function in

complex sub-problems can be easily parallelized applying the GPU acceleration.
We presented and discussed benchmarks for all the scenarios on multiple setups that showed both gains and limitations of GPU technology on this type of problems. We identified scenarios where GPU was more suitable. Discussion and reasoning for it concluded practical prove. Significant results were achieved on implementation of game Fox and Hounds. The algorithm used CPU in the initial phase while number of nodes was low. We used it in combination with GPU accelerated parallel execution applied once number of nodes is high enough. With look-ahead depth of 16, the algorithm showed speedup factors of 60 compared to CPU based sequential algorithm. This particular game showed also downsides and the need to be aware of technology limitations, where sequential solver running on CPU performed better than just GPU. Insufficiently large number of inputs for processing on GPU caused slowdown of the application. Overall, GPU proved to be a very suitable coprocessor to CPU and using them in cooperation we were able to both reduce time and increase depth of search significantly.

Implementation of more complex games like chess and go, using combined performance of both GPU and CPU is still a challenge. We have proved that algorithms used to solve this kind of problems have suitable granularity to exploit the level of parallelism offered by the GPU. This thesis can serve as starting point for further research into applications of this technology in the domain of adversarial search.

Bibliography

[1] Experiment and theory have a partner: Simulation, Arnie Heller, Ann Parker, U.S. Department of Energy, http://www.eurekalert.org/features/doe/2005-02/dl-eat021105.php

[2] The Method of the Chess Search Algorithms - Parallelization using Two-Processor distributed System, Vladan Vučković, FACTA UNIVERSITATIS (NIS), Ser. Math. Inform. Vol. 22, No. 2 (2007), pp. 175–188

[3] Scalability and Parallelization of Monte-Carlo Tree Search, Bourki A., Chaslot G., Coulm M., Danjean V., Doghmen H., Herault T., Hoock J.-B., Rimmel A., Teytaud F., Teytaud O. et al. The International Conference on Computers and Games 2010, Japon (2010)

[4] GPU accelerated pathfinding, Proceedings of the 23rd ACM SIGGRAPHEUROGRAPHICS symposium on Graphics hardware (2008) Volume: 65, Issue: 17, Publisher: Eurographics Association, Pages: 65-74, ISSN: 17273471

[5] Classic.Ars: Understanding Moore's Law, Jon Stokes, http://arstechnica.com/hardware/news/2008/09/moore.ars

[6] Silicon Insider: Welcome to Moore's War ABC News, Michael S. Malone, 27 March 2003, http://abcnews.go.com/Business/story?id=86673&page=1

[7] Software and the Concurrency Revolution, H. Sutter, J. Larus, Queue - Multiprocessors, Volume 3 Issue 7, September 2005

[8] www.intel.com

[9] The Landscape of Parallel Computing Research: A View from Berkeley, Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick, EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183 December 18, 2006

[10] www.nvidia.com

[11] www.top500.org

[12] Artificial Intelligence: A Modern Approach (2nd Edition), Stuart J.Russell , Peter Norvig, Prentice Hall, ISBN: 0137903952, 2003

[13] www.nvidia.com
[14] An Introduction to OpenCL, http://www.amd.com
[15] http://www.macresearch.org/opencl
[16] The OpenCL Specification, Editor: Aaftab Munshi, Version: 1.1, Revision: 36, Khronos OpenCL Working Group
[17] NVIDIA CUDA C, Programming Guide, NVIDIA, Version 3.2 10/22/2010
[18] OpenCL Tutorial with OpenCLTemplate and Cloo, http://www.cmsoft.com.br/
[19] NVIDIA OpenCL Best Practices Guide, NVIDIA, Version 1.0, www.nvidia.com
[20] Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow; Wilson W. L. Fung, Electrical and Computer Engineering University of British Columbia
[21] David B. Kirk and Wen-mei Hwu, Programming Massively Parallel Processors: A Hands-on Approach, February 5, 2010, ISBN 0123814723
[22] NVIDIA Fermi with CUDA and OpenCL, http://blog.accelereyes.com/blog/2010/05/10/nvidia-fermi-cuda-and-opencl/
[23] Artificial Intelligence: A Modern Approach (2nd Edition), Stuart J.Russell , Peter Norvig, Prentice Hall, ISBN: 0137903952, 2003
[24] Minimax Search and Reinforcement Learning for Adversarial Tetris, Maria Rovatsou, Thesis for Department of Electronic and Computer Engineering Technical University of Crete
[25] Solving difficult game positions, Jahn-Takeshi Saito. Maastricht, August 2010
[26] Games solved: Now and in the future, Artificial Intelligence, H. Jaap van den Herik, Jos W.H.M. Uiterwijk, Jack van Rijswijck, Artificial Intelligence 134 (2002) 277–311
[27] V. Allis, Searching for Solutions in Games and Artificial Intelligence. PhD thesis, Department of Computer Science, University of Limburg, 1994
[28] Hex, Thomas Maarup · University of Southern Denmark, Department of Mathematics and Computer Science, 2005
[29] Go-Moku Solved by New Search Techniques, Allis, L.V., Huntjes, M.P.H., and Herik, H.J. van den, 1996, Computational Intelligence, Vol. 12, No. 1
[30] Solving Nine Men's Morris, R. Gasser, 1996, Computational Intelligence, Vol. 12
[31] Checkers is solved, Schaefer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., and Sutphen, S., 2007, Science, Vol. 317, No. 5844

[32] Solving the Game of Awari using Parallel Retrograde Analysis, Romein J. W., Bal, H. E. In: IEEE Computer 2003; 36 (10): 26-33.

[33] Solving Kalah, Irving, G., Donkers, H.H.L.M., and Uiterwijk, J.W.H.M. (2000). ICGA Journal, Vol. 23, No. 3

[34] Solving Connect-4 on Medium Board Sizes, Tromp, J. (2008), ICGA Journal, Vol. 31, No. 2

[35] Lambda-search in game trees – With application to Go, Thomsen, T., 2000, International Compuer Games Association Journal

[36] Heuristic Programming in Artificial Intelligence 1: the First Computer Olympaid, Allen, J. D. (1989), pages 134–135. Ellis Horwood, Chichester, England.

[37] The Principal Continuation and the Killer Heuristic, Selim Akl and Monroe Newborn, 1977 ACM Annual Conference Proceedings, pp. 466-473. ACM, Seattle, WA.

[38] Conspiracy Numbers, Jonathan Schaeffer, 1990, Artificial Intelligence, Vol. 43, No. 1, pp. 67-84. ISSN 0004-3702

[39] Solving Renju, Wágner, J. and Virág, I., (2001), International Computer Games Association Journal, 24

[40] Large scale parallelization of Alpha-Beta Search: An algorithmic architectural study with computer chess, F.H.Hsu, Phd. thesis. Carnegie Mellon University, Pittsburg. USA. (1990)

[41] Advances in computer chess IV., R.M. Hyatt, B.E. Gower, H.L. Nelson, Cray Blitz, D.F. Beal (Editor), Pergamon press. pp. 8-18

[42] Single-Agent Parallel Window Search, C. Powley, Richard E. Korf, IEEE Transactions on Pattern Analysis and Machine Intelligence archive, Volume 13 Issue 5, May 1991

[43] Parallel Alpha-beta search on Arachne, R.A. Finkel, J.P. Fishburn, IEEE International conference on parallel programming; pp. 235-243 (1890)

[44] Parallelism in Alpha-beta search, R.A. Finkel, J.P. Fishburn, Artificial intelligence Vol. 19 pp. 89-106 (1982)

[45] Game tree search on massively parallel systems, Rainer Feldmann, Thesis, University of Paderborn, 1993

[46] The Dynamic Tree-Splitting Parallel Search Algorithm, Robert Hyatt (1997).

ICCA Journal Vol. 20, No. 1, pp. 3-19.
[47] Chess Programming Part VI: Evaluation Functions, François Dominic Laramée, http://www.gamedev.net/page/resources/_/reference/programming/artificial-intelligence/gaming/chess-programming-part-vi-evaluation-functions-r1208
[48] http://en.wikipedia.org/wiki/Fox_games#Fox_and_Hounds
[49] Winning ways for your mathematical play, E.R. Berlekamp, J.H. Conway, and R.K. Guy. Academic Press, 1982
[50] Ayo, The Awari Player,or How better representation trumps deeper search, Daoud M., Kharma N., Haidar A, Popoola, J., Dept. of Electr. & Comput. Eng., Concordia Univ., Montreal, Que., Canada
[51] Parallel Tree Search on SIMD Machines, Holger Hopp, Peter Sanders, Proceedings of the Second International Workshop on Parallel Algorithms for Irregulary Structured Problems, Vol. 980, p. 349-361, 1995.

List of Tables

Attachments

Content of the CD:

1) An electronic copy of the master thesis in folder: <CD>/gpu_accelerated_adversarial_search.pdf

2) Source code for all tested scenarios is placed in two separate folders, cpu and gpu. In each of them, appropriate versions of tested scenarios can be found. Scenarios are divided into folders, each containing make file configured to build executable from source code. Applications were created and tested in Ubuntu 9.01. OpenCL needs to be installed on the system and supported graphic card is required. Details about installation can be found in <CD>/readme.txt

Configuration, important parameters (if applicable):
LOOK_AHEAD – look-ahead value
PIECES_TOTAL – total number of pieces on the board
ITERATIONS, DATA_SIZE – input size
pieces_player – starting index of first piece per player
initialPosition – predefined initial position

List of main source code files:

- cpu
  contains sequential algorithms tested on CPU
  - fnh1/FoxNHounds-FnH1.c
    game Fox and Hounds scenario FnH1
  - fnh2_quiescence/FoxNHounds-FnH2-quiescence.c
    Quiescence search for Fox and Hounds
  - fnh2_selective/FoxNHounds-FnH2-selective.c
    Selective search for Fox and Hounds
  - fnh3_evaluation_function/FoxNHound-evaluation.c
    Evaluation function for Fox and Hounds
  - synthetic_board_game_no_recursion/synthetic_board_game.c
    Minimax search on artificial board game

  - synthetic_board_game_quiescence/synthetic_board_game_quie.c
    selective search on artificial board game
  - synthetic_board_game_selective/synthetic_board_game_selective.c
    quiescence search on artificial board game

- gpu
  contains parallel algorithms tested on GPUs
  - fnh1/combined.c
    Parallel version of game Fox and Hounds scenario FnH1
  - fnh2_quiescence/OpenCl-FoxNHounds-quie.c
    Parallel version of Quiescence search for Fox and Hounds
  - fnh2_selective/OpenCl-FoxNHounds-selective.c
    Parallel version of Selective search for Fox and Hounds
  - fnh3_evaluation_function/OpenCl-Evaluate_FoxNHounds.c
    Parallel version of Evaluation function for Fox and Hounds
  - synthetic_board_game_no_recursion/board_combined.c
    Parallel version of selective search on artificial board game
  - synthetic_board_game_quiescence/board_game_quie.c
    Parallel version of quiescence search on artificial board game
  - synthetic_board_game_selective/board_game_sel.c
    Parallel version of quiescence search on artificial board game