

GPU computing in discrete optimization. Part I: Introduction to the GPU

André R. Brodtkorb · Trond R. Hagen ·
Christian Schulz · Geir Hasle

Received: 17 April 2012 / Accepted: 28 March 2013 / Published online: 30 April 2013
© Springer-Verlag Berlin Heidelberg and EURO - The Association of European Operational Research Societies 2013

Abstract In many cases there is still a large gap between the performance of current optimization technology and the requirements of real world applications. As in the past, performance will improve through a combination of more powerful solution methods and a general performance increase of computers. These factors are not independent. Due to physical limits, hardware development no longer results in higher speed for sequential algorithms, but rather in increased parallelism. Modern commodity PCs include a multi-core CPU and at least one GPU, providing a low cost, easily accessible heterogeneous environment for high performance computing. New solution methods that combine task parallelization and stream processing are needed to fully exploit modern computer architectures and profit from future hardware developments. This paper is the first part of a series of two, where the goal of this first part is to give a tutorial style introduction to modern PC architectures and GPU programming. We start with a short historical account of modern mainstream computer architectures, and a brief description of parallel computing. This is followed by the evolution of modern GPUs, before a GPU programming example is given. Strategies and guidelines for program development are also discussed. Part II gives a broad survey of the existing literature on parallel computing targeted at modern PCs in discrete optimization, with special focus on papers on routing problems. We conclude with lessons learnt, directions for future research, and prospects.

Keywords Discrete optimization · Parallel computing · Heterogeneous computing · GPU · Survey · Introduction · Tutorial · Transportation · Travelling salesman problem · Vehicle routing problem

A. R. Brodtkorb · T. R. Hagen · C. Schulz · G. Hasle (✉)
Department of Applied Mathematics, SINTEF ICT, P.O. Box 124, Blindern, 0314 Oslo, Norway
e-mail: geir.hasle@sintef.no

Introduction

Applications of optimization problems abound in society. Today, there are many examples of optimization-based decision support tools that improve important processes both in industry and the public sector. Such tools are becoming more powerful, more widespread, and more critical to the performance of their users. A successful tool provides substantial improvement of key factors to the user organization. Examples are savings of economical and environmental costs, enhanced customer service, higher revenues, less use of critical resources, and improvement of human factors. Vehicle routing software (Partyka and Hall 2012) is but one example.

The impact of such tools is to a large degree dependent on their optimization performance, i.e., the quality of solutions produced within a given response time requirement. Optimization performance is largely determined by the selected optimization method, the implementation of this method on the targeted hardware platform, and the computational performance of the hardware. These three factors are closely intertwined.

More often than not, the optimization problem to be solved is computationally hard. This is particularly true for discrete optimization problems (DOPs). Over the past few decades, there has been a tremendous increase in the ability to solve ever more complex optimization problems. Bixby (2002) reminds us that the performance of commercial Linear Programming solvers increased by a factor of one million in the period 1987–2000. Roughly a factor of 1,000 is due to better methods, and a similar factor stems from the general performance increase of computers.

For many applications, there is still a large gap between the requirements and the performance of today's optimization-based decision support systems. The ability to provide better solutions in shorter time will give substantial savings through better optimization performance of existing tools. Moreover, applications that are too complex to be effectively solved by the technology of today may become within reach of the optimization technology of tomorrow. More integrated, larger, and richer optimization problems may be solved. Again, further performance increase will result from a combination of better optimization algorithms that are implemented in more efficient ways on more powerful computers.

For many decades, Moore's law materialized in the form of a doubling of clock speed for commodity processors every 18 months or so. This was the realm of the tongue-in-cheek "Beach law"¹. Around year 2000, the architecture of processors for commodity computers started to change. Multi-core processors with an increasing number of cores and higher total theoretical performance than their single core predecessors emerged, but each core had lower clock speed. Hence, developers of sequential software could no longer enjoy the pleasant, serendipitous effects of the Beach law. From then on, algorithms for computationally hard tasks such as solution of optimization problems need an efficient, task parallel implementation to fully utilize multiple CPU cores².

¹ One way of doubling the performance of your computer program is to go to the beach for two years and then buy a new computer.

² For a brief introduction to main concepts in parallel computing, see "[Parallel computing](#)" below.

In addition, there has over the past decade been a drastic improvement of performance and general programmability of massively parallel stream processing (data parallel) accelerators. Data parallelism, also called stream processing, means that each processor performs the same task on different pieces of distributed data. The origin was the graphics processing unit (GPU) that was a normal component in common PCs. Primarily driven by requirements from the gaming industry, the computational performance of GPUs developed rapidly. Thus, it became more and more interesting to utilize GPUs as accelerators for compute bound tasks in general purpose computing. This trend became a natural driver for better programmability of GPUs through industry-standard languages and high quality development tools.

GPUs of today have a large number of relatively simple processors that have general purpose computing capabilities and their architecture supports data parallelism. The theoretical GPU performance has lately increased far more rapidly than the theoretical CPU performance, as illustrated in Fig. 1. The GPU is now regarded as an accelerator to be used in tandem with a multi-core CPU. Leading processor manufacturers have recently developed an integrated multi-core CPU and GPU on a single die.

To fully profit from the general recent and future hardware development on modern PC architectures, optimization methods that combine task and data parallelism must be developed. Ideally, such methods should be flexible and self-adaptable to the hardware at hand. The parallel, heterogeneous architecture of modern processors also motivates a fundamental re-thinking of solution methods. Algorithms that are obviously inefficient in a sequential computing model may be optimal on a massively parallel architecture.

This paper has two main goals. First, we provide a tutorial style introduction to the modern PC architecture and how to exploit it through parallel computing. Second, we give a critical survey of the literature on discrete optimization for such architectures with a focus on routing problems. For selected papers, we discuss implementation details and insights. Our intended main audience consists of researchers and practitioners in discrete optimization, routing problems in particular, which are not proficient in modern PC hardware and heterogeneous computing. We hope the paper will serve as a useful basis for increased, high quality research and development efforts in this combined research area of high importance.

The area of GPU-based methods for discrete optimization is still in its infancy. The bulk of the limited literature consists of reports from rather basic implementations of existing optimization methods on GPU, with measurement of speedup relative to a CPU implementation of unknown quality. This is not necessarily uninteresting. A speedup of existing solution methods has great pragmatic value. It enables resolution of large, complex, and time critical applications of discrete optimization that are beyond the reach of current technology. Also, it enables more comprehensive and thorough empirical scientific investigations in discrete optimization and, hence, a deeper understanding. However, it is our opinion that research in this area should be performed in a more scientific fashion: with thorough and fair measurement of speedup, and also with focus on efficiency of the implementation. An important research avenue is the design of novel methods that exploit the full heterogeneity of modern PCs in an efficient, flexible, and possibly self-adaptable

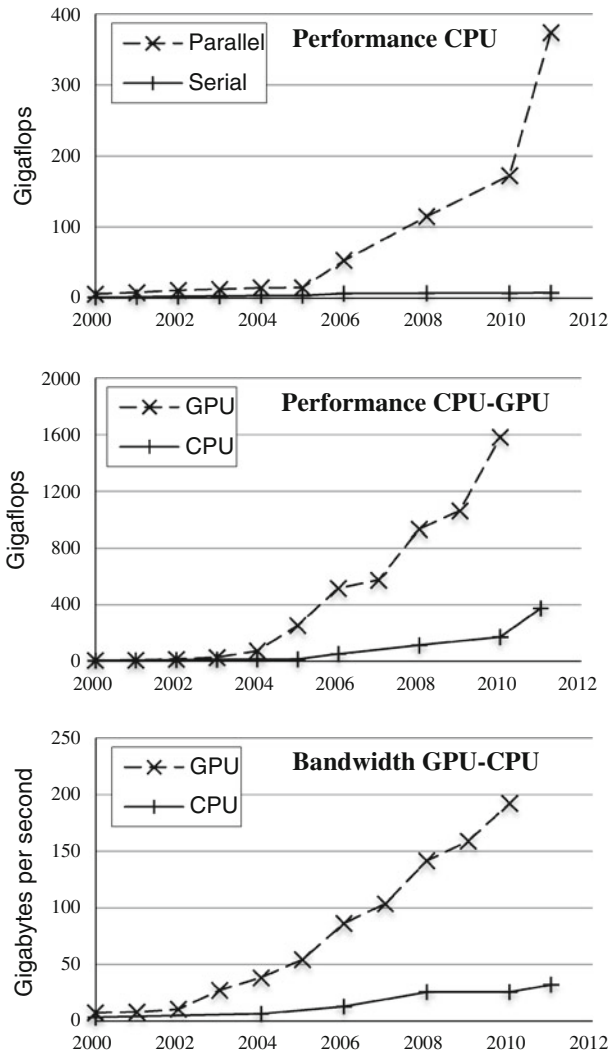


Fig. 1 Performance of CPUs versus GPUs in terms of theoretical peak performance in single precision, and memory bandwidth. Data constructed from processor specifications as reported from Intel (2012) and NVIDIA (2012)

way. As far as we can see, there are no such publications in the literature. If this paper will inspire research in this direction, a main objective has been fulfilled. We strongly believe that the potential is huge.

The remainder of Part I of this paper is organized as follows. The “[Parallel computing](#)” section gives a brief introduction to parallel computing in general. In “[Modern computer architecture](#)”, we describe modern computer architectures with multi-core processors for task parallelism and accelerators for data parallelism (stream processing). Alternative programming environments for such hardware are

discussed in “[Development of modern GPU technology](#)”. In “[Programming example in CUDA](#)”, a simple prototype of a GPU-based local search procedure is presented to illustrate the execution model of GPUs. We proceed in “[Development strategies](#)” with guidelines and strategies for optimizing GPU code. For illustrative purposes, we then take a closer look at the performance of GPU codes in “[Profiling the local search example](#)” and sum up Part I in “[Summary and conclusion](#)”. In Part II (Schulz et al. 2013), we give a survey of the literature on GPU-based methods in discrete optimization, with focus on routing problems.

Parallel computing

The idea of parallel computing dates back to the Italian mathematician Menabrea and his “Sketch of the Analytical Engine Invented by Charles Babbage” in (Menabrea 1842). Menabrea’s paper has extensive notes by the now famous translator, Lady Lovelace. In the notes she wrote what has been recognized as the world’s first computer program. It was not until the late 1960s that computers with multiple processors emerged and parallel computing was realized, however.

There are several main types of parallel computing. Apart from the low level *instruction level parallelism* that is offered by modern processors, there are two main categories: *task parallelism* and *data parallelism*. In task parallelism, different procedures are performed on possibly different sets of data, typically using different processes or threads. Normally, but not necessarily, the parallel threads or processes execute on multiple processors, and there is communication between them. In the basic form of data parallelism, the same procedure, often referred to as *the kernel*, is executed on multiple data in parallel, on multiple processors. There is also a distinction between *fine-grained parallelism*, where processes or threads synchronize or communicate many times per second, *coarse-grained parallelism* if they communicate less frequently, and *embarrassingly parallel* if they only rarely need to communicate or synchronize.

Parallel computer systems can be categorized by the nature of their processors, their processor interconnection, their memory and the communication between the processors. The set of processors may be homogeneous or heterogeneous. They may be integrated on the same chip and communicate via a high-bandwidth bus such as, modern multi-core PC processors, or physically distributed around the globe and communicate over the Internet as in grid computing. Main memory may be either *shared* between the processors or *distributed*. Computer clusters are groups of loosely connected, fully-fledged, typically general purpose, not necessarily similar computers that are tightly connected and communicate through a network.

In this paper, we concentrate on modern commodity processors with multiple cores that share memory, and one or more data parallel accelerators with separate memory such as the GPU, as the platform for parallel, heterogeneous computing. There is a substantial literature on scientific computing that exploits such hardware (Brodtkorb et al. 2010; Owens et al. 2008).

Modern computer architectures

From the first microprocessor emerged in the 1970s, up until 2004, virtually all mainstream computers have used a serial execution model, in which one instruction is executed after another. The exponentially growing performance of such CPUs has traditionally come from two main factors: an increasing number of transistors, and an increasing frequency. Around 2004, however, we saw an abrupt halt to the serial performance. Increasing the number of transistors yielded only marginal performance increases, and the frequency had reached the physical limit that the chip can withstand. Since then, we have instead seen an increase in parallelism. Whilst one previously used the increasing number of transistors for executing instructions more efficiently, the extra transistors today are spent on creating multi-core designs.

Simultaneously as we have seen a growing parallelism in CPUs, we have also seen alternative architectures emerge. Around the year 2000, researchers started exploring how GPUs could be used to solve non-graphics problems. GPUs utilize a SIMD (single-instruction-multiple-data) type of execution model. SIMD was originally developed in the 1970s for vector supercomputers³. Although SIMD machines that can execute up to 64,000 instructions in parallel were developed, such computers were very specialized and expensive. In comparison, parallel computers based on several main-stream processors running independent tasks offered more flexibility at a lower cost. With the development of GPUs, a cheap, powerful SIMD based accelerator became easily accessible. Programming the GPU for non-graphic tasks was originally an error prone and cumbersome process, but showed that GPUs could solve a multitude of problems faster than the CPU. Since then, GPUs have become highly programmable using modern C-based languages, and have received widespread adoption. In fact, three of the worlds five fastest supercomputers today use GPU acceleration (<http://www.top500.org/>), and there is an increasing number of libraries, such as MAGMA and CULA sparse, and commercial software products, such as Adobe Photoshop and MATLAB, that incorporate GPU acceleration.

The reason for the widespread adoption of GPUs is twofold. The first reason is that GPUs are inexpensive and readily available in everything from laptops to supercomputers. The second reason is that they offer an enormous performance, especially when considering performance per watt or performance per dollar. This difference between the GPU and the CPU is due to their differing design intents. The CPU is a highly complex processor, and modern CPUs can have over two billion transistors⁴. However, most of these transistors are spent on caches, complex logic for instruction execution and latency hiding, and operating system functionality, leaving only a small percentage for computational units. GPUs, on the other hand, have up to three billion transistors, a slight increase compared to CPUs, and spend most of these transistors on computational units. This means that GPUs cannot replace CPUs, as they do not contain enough complex functionality, but can at the same time offer an extreme floating point performance. A further difference between these architectures, is that CPUs are optimized for single thread

³ Vector-computers execute the same instruction on each element of a vector.

⁴ For example, the Intel Core i7-3960X holds 2.27×10^9 transistors

performance, meaning it is very efficient at making one task run quickly. GPUs, on the other hand, are designed for throughput instead of single thread performance, meaning it can perform a lot of computations fast, but the speed of each computation might be slower.

The most recent trend in modern computer architectures is the incorporation of GPU cores and CPU cores on the same physical chip. This combines the best of both worlds by incorporating traditional CPU cores, which are efficient for serial tasks, and GPU cores, which are efficient for throughput tasks. There are also other alternatives to GPUs for accelerated computing. For example, in 2006 Sony, Toshiba and IBM released the Cell processor (Chen et al. 2007) used in both the PlayStation 3 and the first petaflops supercomputer (Barker et al. 2008). This processor was based on using one traditional CPU core coupled with eight lightweight accelerator cores all on the same chip, and it delivered unprecedented performance. However, the programming model was cumbersome and has been openly criticized by many, and there has not been an updated version yet, making it a one-off architecture. Another alternative is to use FPGAs (field programmable gate arrays). FPGAs are essentially reprogrammable integrated circuits that offer an extreme performance per watt ratio, as you only use power on actual computation. However, as with application specific integrated circuits (ASICs), programming them is both cumbersome and error prone as one has to consider details, such as timings etc. Nevertheless, over the last five years, there has been a tremendous development in programmability through the development of C-like languages. However, programming FPGAs is still a challenging process.

Development of modern GPU technology

GPUs were originally designed for offloading demanding graphics functions from the CPU to a dedicated co-processor. As such, it originally accelerated a fixed set of graphics operations, such as vertex transformations and lighting calculations of a 3D game world. In the early days of GPU computing, one had to use these graphics specific APIs, such as OpenGL (Shreiner et al. 2012) or DirectX (Luna 2012) to perform computations, see also Fig. 2. This was a cumbersome and error prone process, as one had to rephrase the problem into operations on graphical primitives. As a trivial example, the addition of two matrices could be performed by creating a window with one *pixel* per output element, and *rendering* one quadrilateral that covered the whole window. This quadrilateral would then be *textured* with two textures, in which the matrix values would be represented as a color, and the GPU would add these colors together unknowing that it was performing a matrix addition. For more complex algorithms, such as matrix multiplication or Gaussian elimination, however, this process becomes quite difficult.

The earliest GPUs that accelerated a fixed set of graphics functions used the so-called *fixed function graphics pipeline*, and around 2003 parts of this pipeline became programmable with the release of the NVIDIA GeForce 256 GPU and the C for Graphics (Cg) (Fernando and Kilgard 2003) language. Figure 3 shows this programmable graphics pipeline, in which the input is a set of *vertices* in 3D space

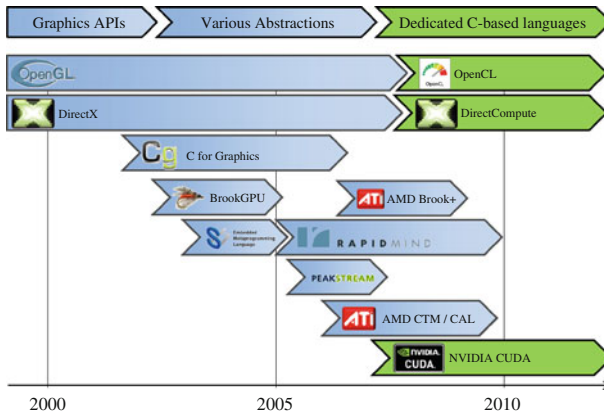


Fig. 2 Evolution of GPU programming languages. Initially, the graphics card was programmed using dedicated graphics languages, but since 2007 general purpose languages ,such as CUDA, DirectCompute, and OpenCL have appeared

that typically represent triangles of a 3D model. These vertices are then first transformed into the so-called *clip space*, essentially the world as seen from the camera, by the *vertex shader*⁵. This is a programmable stage, meaning that we can calculate the new position of the vertex using a program. After vertices have been transformed into clip space, the GPU typically creates triangles from them in the *primitive assembly* stage, and removes triangles that are not seen by the camera in the *primitive processing stage*. Then, the GPU converts the triangles into fragments in a process called *rasterization*. Fragments are candidates for an output pixel, and each triangle that covers a pixel position gives rise to one fragment. This means that we may have multiple fragments per pixel, for example, if two triangles cover the same pixel. All these fragments are then processed by the fragment shader, which determines the color of the fragment using for example textures⁶ and lighting calculations. Finally, all these fragments enter the framebuffer operations stage, which determines the final pixel color from all the input fragments. This stage can give rise to transparency, by blending the fragments from two overlapping triangles, or simply to choose the one closest to the camera, see Fig. 4. The latter is done using the depth test.

In 2007, however, NVIDIA released the first general-purpose language for programming GPUs, called CUDA. This release met the demands of researchers who up until then had either used the graphics APIs, or the various abstractions of these, and subsequently we saw a whole new level of interest in GPU computing. Since then, two alternatives to CUDA have emerged, namely OpenCL and DirectCompute. All these are quite similar when it comes to the basic programming concept, which is often referred to as data parallel execution.

⁵ The vertex shader typically uses a modelview matrix and a perspective matrix to transform the vertices from *object space* to *clip space*.

⁶ A texture is a 2D image that typically is shown on a 3D surface to increase realism.

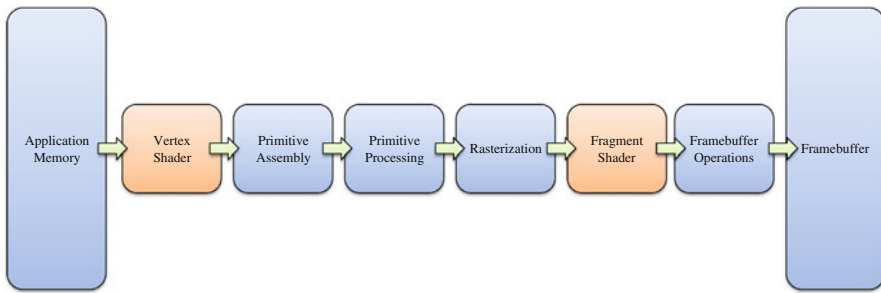


Fig. 3 The programmable OpenGL graphics pipeline with programmable stages marked in orange. Input to the pipeline are *vertices* that often represent triangles in 3D, and the output is a 2D image on screen

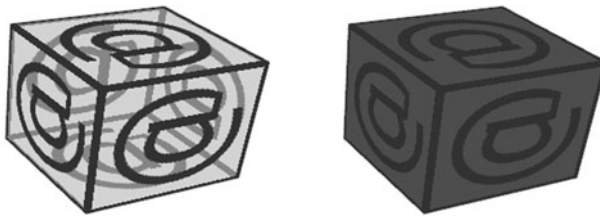


Fig. 4 Illustration of the framebuffer operations stage of the graphics pipeline. The *left figure* uses *blending* of the primitives for each fragment, and the *right* selects the fragment closest to the camera using the depth test

Programming example in CUDA

To illustrate the execution model of GPUs and how they are programmed using CUDA, we present a simple prototype of a local search procedure running on the GPU. This programming example is designed for clarity and used to demonstrate how GPUs work, and thus we have neglected important performance optimizations. A more thorough discussion on algorithm design and optimization can be found in (Schulz 2013).

A local search procedure starts with a given (feasible) *initial solution* to the problem at hand as the current solution. In our example, it is an instance of the well-known Euclidean Travelling Salesman Problem: Given a number of cities, find a permutation with minimal total traveling cost of visiting all cities in the order of the permutation. Traveling cost between cities is defined as the Euclidean distance. Local search with steepest descent (or best improvement) then examines all solutions that lie within a certain neighborhood of the current solution. The best improving neighboring solution is accepted as the new current solution, and the local search procedure continues in the same way. The procedure stops when there is no improving neighbor, i.e., a local minimum for the defined neighborhood is found.

Typically, the neighborhood is not defined explicitly as a set of solutions, but described implicitly in terms of a type of change to the current solution defined by a

neighborhood operator. Given a current solution and the operator, the neighborhood is generated by applying the operator in all possible ways to the solution. Each individual change is called a *move*. In our example, we use a simple swap neighborhood where a move simply exchanges the position of two specific cities in the permutation. In our representation, we keep the first city fixed to avoid rotating the solution.

The quality of a move can be expressed by the difference, or *delta* value, between the cost of the neighboring solution the move leads to and the cost of the current solution. Hereby a negative delta value means that the neighboring solution has less cost than the current one, i.e., for minimizing problems like the TSP it is better.

Let us start our example by showing how this can be implemented as a traditional CPU algorithm. A fixed random permutation is our initial solution. Let n be the numbers of cities in the problem instance to be solved, leading to a swap neighborhood size of $(n - 1)(n - 2)/2$ moves. We systematically generate these moves, evaluate each of their incremental cost, and select the best move as follows. We include a feasibility check of each move to illustrate where such a test can be performed (here on the CPU and later on the GPU), although a swap move for the TSP will always be feasible.

```

1 //Loop through all possible moves
2 for (int k=0;k<num_moves;++k) {
3     //Generate move number k
4     Move move = generate_move(k, num_nodes, &solution[0])
5
6     if (is_feasible(move)) {
7         //Evaluate cost of the move
8         float delta = cost(move);
9
10        //Save the move that improves the solution the most
11        if (delta < min_delta) {
12            best_move = move;
13            min_delta = delta;
14        }
15    }
16 }
```

After having evaluated the full neighborhood, we apply the best move, or exit if we have found a local minimum:

```

1 while (true) {
2     //Find the best move as shown above
3     ...
4
5     //If no moves improve the solution, we have reached local minimum
6     if (min_delta > -1e-7) {
7         break;
8     }
9
10    //Otherwise, apply the best move to current solution
11    apply(best_move);
12 }
```

This problem is well suited for execution on GPUs, due to its highly parallel nature: the evaluation of each move can be performed independently of all other

moves. However, finding the move that improves the solution the most is a serial process. Let us start showing how the evaluation can be done in parallel on the GPU. We start by first allocating storage space for the solution on the GPU, and copy the initial configuration to the GPU as well:

```

1 //Allocate data on the GPU
2 err = cudaMalloc(&solution_gpu , solution_size_in_bytes);
3 if (err != cudaSuccess) {
4     cout << "Could not allocate GPU memory for solution" << endl;
5     exit(-1);
6 }
7
8 //Copy data from the CPU to the GPU
9 err = cudaMemcpy(solution_gpu , &solution[0] ,
10                 solution_size_in_bytes , cudaMemcpyHostToDevice);
11 if (err != cudaSuccess) {
12     cout << "Could not copy solution to GPU memory" << endl;
13     exit(-1);
14 }

```

Similarly, we allocate space for the city coordinates on the GPU and copy them from the CPU to the GPU. We can now write a *kernel* that evaluates the cost of moves, and stores this on the GPU. A kernel is a function that is invoked by a large number of threads in parallel on the GPU. Our approach is to write a kernel that evaluates in each thread a subset of the total number of moves, and stores the best move of the subset in main GPU memory (which must be allocated similarly to `gpu_solution`):

```

1  __global__
2  void evaluate_moves_kernel(unsigned int* solution ,
3                           float* city_coordinates ,
4                           float* best_subset_deltas ,
5                           unsigned int* best_subset_moves ,
6                           unsigned int num_nodes ,
7                           unsigned int num_moves_per_thread) {
8
9      //Compute the thread ID
10     unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
11     const unsigned int num_moves = (num_nodes-2)*(num_nodes-1)/2;
12
13     float min_delta = 0.0;
14     const unsigned int first_move = tid*num_moves_per_thread;
15     unsigned int best_move = first_move;
16
17     //Loop through the subset and find best move
18     for (int i=first_move; i<first_move+num_moves_per_thread; ++i) {
19         if (i < num_moves) {
20             Move move = generate_move(i, num_nodes, solution);
21             if (is_feasible(move)) {
22                 float move_cost = cost(move, city_coordinates);
23                 if (move_cost < min_delta) {
24                     min_delta = move_cost;
25                     best_move = i;
26                 }
27             }
28         }
29     }
30
31     //Write best move cost and index to GPU main memory
32     best_subset_deltas[tid] = min_delta;
33     best_subset_moves[tid] = best_move;
34 }

```

Here, the keyword `__global__` marks the function as a kernel, and the number of parallel invocations is determined by the *grid* and *block* configuration. The global CUDA variable `blockDim.x` contains our one-dimensional block size, `threadIdx.x` the index of the thread inside its block, and `blockIdx.x` the index of the block inside the grid. A block is simply a collection of threads, and a grid is a collection of blocks. In our example, we have chosen a total of 8,192 threads split into blocks consisting of 128 threads, giving us a total of 64 blocks. These numbers are somewhat arbitrarily chosen, but still follow some fundamental guidelines. The block size should be a multiple of 32, as the GPU executes 32 threads in SIMD⁷ fashion, and we want enough blocks to occupy all of the 16 multiprocessors on current GPUs.

The next thing we now need to do, is to reduce the best moves for the 8,192 different subsets into the best global move, and apply this move. We can do this in another *kernel*, but this time, we only invoke one block consisting of 512 threads. This is because threads within one block can cooperate, whilst different blocks are independent. In the first part of the kernel, we use parallel reduction⁸ in *shared memory*⁹ to find the best move and we then apply this move:

```

1  __global__
2  void apply_best_move_kernel(unsigned int* solution ,
3                             float* deltas ,
4                             unsigned int* moves,
5                             unsigned int num_nodes,
6                             unsigned int deltas_size) {
7      // Computation of minimum delta removed for brevity
8      ...
9      // now: deltas_shmem[0] contains minimal delta
10     //                                     (in shared memory)
11     //     moves_shmem[0] contains corresponding best move
12     //                                     (in shared memory)
13
14     //Let thread 0 apply the move
15     if (threadIdx.x == 0) {
16         Move move = generate_move(moves_shmem[0], num_nodes, solution);
17         if (deltas_shmem[0] < 0) {
18             apply_move(move);
19         }
20
21         //Write the best delta to main GPU memory
22         deltas[0] = deltas_shmem[0];
23     }
24 }

```

With these two GPU kernels, we can find the best move for the current configuration in parallel and then also apply it. What remains is the CPU logic for *launching* these kernels, and stopping execution when no moves improve the solution:

⁷ SIMD stands for single instruction multiple data.

⁸ Reduction is a standard SIMD and, thus, GPU operation which computes the repeated application of a binary operator to all elements in parallel. In our example, the binary operator chooses the move with smaller delta and, thus, the reduction returns the best move.

⁹ Shared memory is a kind of programmable cache or scratch-pad memory on the GPU.

```

1  while (true) {
2      Move best_move;
3      float min_delta = 0.0;
4
5      //Evaluate moves in parallel
6      evaluate_moves_kernel<<<evaluate_grid, evaluate_block>>>(
7          solution_gpu, coordinates_gpu, deltas_gpu,
8          moves_gpu, num_nodes, num_moves_per_thread);
9
10     //Find the best move and apply it
11     apply_best_move_kernel<<<apply_grid, apply_block>>>(
12         solution_gpu, deltas_gpu, moves_gpu,
13         num_nodes, num_evaluate_threads);
14
15     //Copy the smallest delta from the GPU to the CPU.
16     err = cudaMemcpy(&min_delta, &deltas_gpu[0],
17         sizeof(float), cudaMemcpyDeviceToHost);
18     if (err != cudaSuccess) {
19         cout << "Could not copy minimum delta to CPU" << endl;
20         exit(-1);
21     }
22
23     // If no moves improve the solution, we have reached local minimum
24     if (min_delta > -1e-7) {
25         break;
26     }
27 }

```

Both the GPU and the CPU version of this code end up with the same solution in the same number of iterations¹⁰, but there is a dramatic difference in execution speed. For 1,000 cities, the GPU version takes just over 2 s to find the local minimum, whilst the CPU uses over 175 s to complete the same task, a more than 80-fold increase in speed.

Our parallel local search on the GPU was able to achieve a 80 times speed increase compared to the CPU, a figure that is representative for many publications. However, this “speedup” is nothing more than an indication that the GPU has a potential. It is highly likely that both the GPU and the CPU are operating at only a fraction of peak performance, and it is still a major challenge to optimize both the CPU and the GPU version. In “[Profiling the local search example](#)”, we will show that our approach in fact far from utilizes the full potential of the GPU.

Development strategies

GPU programming differs from traditional multi-core CPU programming, because the hardware architecture is dramatically different. It is rather simple to get started with GPU programming, and it is often relatively easy to get speedups over existing CPU codes. But these first attempts at GPU computing are often sub-optimal, and do not utilize the hardware to a satisfactory degree. Achieving a scalable high-performance code that uses hardware resources efficiently is still a difficult task that can take months and years to master.

¹⁰ The GPU version is compiled for compute capability 2.0.

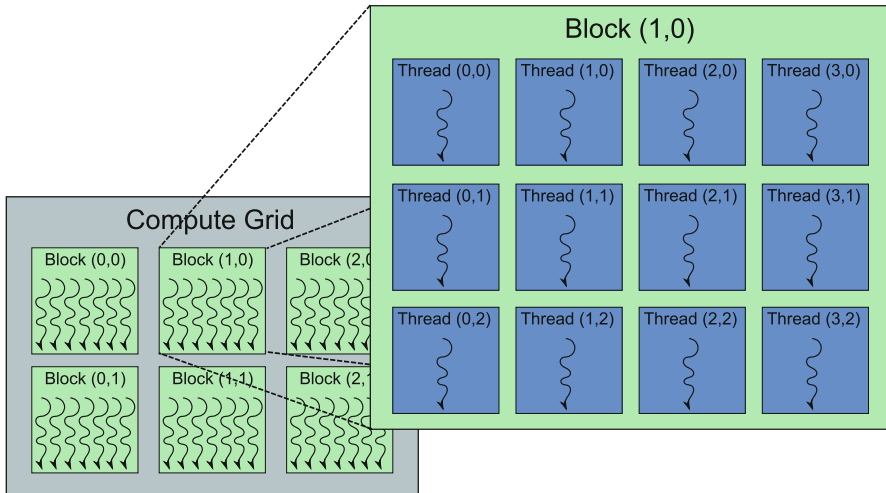


Fig. 5 The CUDA concept of a grid, blocks, and threads. The domain consists of distinct blocks, which again are made up of a set of threads that can communicate and cooperate. Each thread in the global grid can be identified uniquely by the use of its block index in combination with its thread index

In this section, we present techniques for achieving a high resource utilization when it comes to GPUs¹¹. These techniques target NVIDIA GPUs using CUDA, but as both the programming model and hardware is similar for other GPUs and languages, many of these techniques are also applicable in a broader context.

The GPU execution model

The execution model of the GPU is based around the concept of launching a kernel on a grid consisting of blocks as shown Fig. 5. Each block is composed of a set of threads. All threads in the same block can synchronize and cooperate using fast shared memory. These blocks are executed by the GPU, so that a block runs on a single multiprocessor. However, we can have far more blocks than we have multiprocessors, since each multiprocessor can execute multiple blocks in a time-sliced fashion. The grid and block can be one-, two-, and three-dimensional, and determine the number of threads that will be used. Each thread has a unique identifier within its block, and each block has a unique identifier within the grid. By combining these two, we get a unique global identifier per thread.

Latency hiding and thread performance

The GPU uses the massively threaded execution model to hide memory latencies. Even though the GPU has a vastly superior memory bandwidth compared to CPUs,

¹¹ The information in this section is gathered from many sources, including books, documentation, manuals, conference presentations, and on Internet fora. However, most of the optimization techniques presented can be found in (NVIDIA 2011; Micikevicius 2010a, b; Brodtkorb et al. 2012b).

it still takes on the order of hundreds of clock cycles to transfer a single element from main GPU memory. This latency is hidden by the GPU as it automatically switches between threads. Once the current thread stalls on a memory fetch, the GPU activates another waiting thread in a fashion similar to Hyper-Threading (Marr et al. 2002) on Intel CPUs. This strategy is most efficient when there are enough available threads to completely hide the memory latency, however, meaning we need a lot of threads. As there is a maximum number of threads a GPU can support concurrently, we can calculate how large a percentage of this figure we are using. This is referred to as the *occupancy* of the GPU, and is a rough measure of how well the GPU program is at hiding memory and other latencies. As a rule of thumb it is good to keep a relatively high occupancy, but a higher occupancy does not necessarily equate higher performance: Once all latencies are hidden, a higher occupancy may actually degrade performance as it also affects other performance metrics.

Hardware support for multiple threads is available on Intel CPUs as Hyper-Threading, but a GPU thread operates quite differently from these CPU threads. One of the differences from traditional CPU programming is that the GPU executes instructions in a 32-way SIMD fashion, in which the same instruction is simultaneously executed in 32 neighboring threads, called a warp. This is illustrated in Fig. 6, in which different code paths are taken by different threads within one warp. This means that all threads within a warp must execute both parts of the branch, which in the worst case slows down the program by a factor 32. Conversely, the cost of such an if-statement is minimal when all threads in a warp take the same branch.

Sorting is one technique that can be used to avoid expensive branching within a kernel: by sorting the different elements according to the branch we make sure the threads within each warp all execute their code without diverging. Another way of preventing branching is to perform the branch once on the CPU instead of once for each warp on the GPU. This can be done, for example, using templates: by replacing the branch variable with a template variable, we can generate two kernels, one for condition true, and one for condition false, and let the CPU select the correct kernel. The use of templates in this example is not particularly powerful, as the overhead of running a simple if-statement in the kernel would be small. When there are a lot of parameters, however, there can be a large performance gain from using template kernels (Harris 2011; Brodtkorb et al. 2012a). Another example of the benefit of

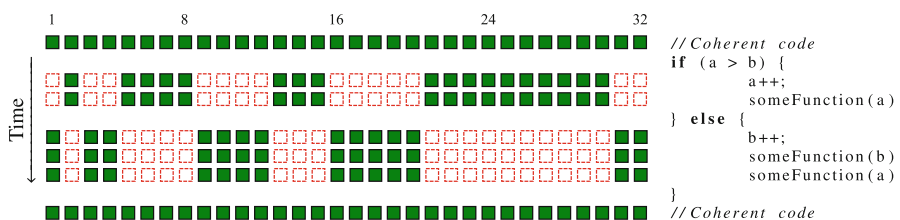


Fig. 6 Thread divergence on 32-wide SIMD GPU architectures. All threads perform the same computations, but the result is masked out for the *dashed boxes*

kernel template arguments is the ability to specify different shared memory sizes at compile time, thus allowing the compiler to issue warnings for out-of-bounds access. The use of templates can also be used to perform compile-time loop unrolling, which has a great performance impact. By having separate kernels for different for-loop sizes, performance can be greatly improved.

Memory guidelines

The memory wall, in which transferring data to the processor is far more expensive than computing on that data, has halted the performance increase of CPU programs for a long time. It can also be a major problem on GPUs, which makes memory optimizations important. The first rule in optimizing memory is to reuse data and keep it as close as possible to the processor. The memory hierarchy on GPUs consists of three main memories, listed in decreasing order by speed: registers, shared memory, and global memory. The use or misuse of these can often determine the efficiency of GPU programs.

Registers are the fastest memory units on a GPU, and each multiprocessor on the GPU has a large, but limited, register file. This register file is divided amongst threads residing on that multiprocessor, and are private for each thread. If the threads in one block use more registers than are physically available, registers will also spill to the L1 cache¹² and global memory, which means that when you have a high number of threads, the number of registers available to each thread is very restricted. This is one of the reasons why a high occupancy may actually hurt performance. Thus, thread-level parallelism is not the only way of increasing performance. It is also possible to increase performance by decreasing the occupancy to allow more registers per thread.

The second fastest memory on the GPU is the *shared memory*. Shared memory is a very powerful tool in GPU computing because it allows all threads in a block to share data. Shared memory can be thought of as a kind of programmable cache, or scratchpad, in which the programmer is responsible for placing data there explicitly. However, as with caches its size is limited (up to 48 KB), which can be a limitation on the number of threads per block. Shared memory is physically organized into 32 banks that serve one warp with data simultaneously. For full speed, each thread must access a distinct bank, which can be achieved, for example, if the threads access consecutive 32-bit elements.

The third type of memory on the GPU is the *global memory*. This is the main memory of the GPU, and even though it has an impressive bandwidth, it has a high latency as discussed earlier. The latencies are preferably hidden by a large number of threads, but there are other pitfalls. First of all, just as with CPUs, the GPU transfers full cache lines¹³ across the memory bus (called coalesced reads). Transferring a single element, therefore, consumes the same bandwidth as

¹² Global memory is cached by several caches on the GPU. The L1 cache is the fastest (and smallest) cache in the cache hierarchy, followed by the L2 cache which is larger but slower.

¹³ Caches transfer continuous regions of memory from RAM called cache lines (128 bytes on Fermi class GPUs). These cache lines increase the read performance when the processor requests neighboring elements.

transferring a full cache line as a rule of thumb. To achieve full memory bandwidth, we should therefore program the kernel such that warps access continuous regions of memory. Furthermore, we want to transfer full cache lines, which is done by starting at a quad word boundary (the start address of a cache line), and transfer full quadwords (128 bytes) as the smallest unit. The address alignment is typically achieved by padding arrays. Alternatively, for non-cached loads, it is sufficient to align to word boundaries and transfer words (32 bytes). To fully occupy the memory bus the GPU also uses memory parallelism, in which a large number of outstanding memory requests are used to occupy the bandwidth. This is both a reason for a high memory latency, and a reason for high bandwidth utilization.

In addition to the above-mentioned memory areas, the NVIDIA GPUs of the recent Fermi architecture have hardware L1 and L2 caches. The L2 cache size is fixed and shared between all multiprocessors on the GPU, whilst the L1 cache is per multiprocessor. The L1 cache can be configured to be either 16 or 48 KB, at the expense of shared memory. The L2 cache, on the other hand, can be turned on or off at compile-time, or by using inline PTX assembly instructions in the kernel. The benefit of turning off the L2 cache is that the GPU is allowed to transfer smaller amounts of data than a full cache line, which will often improve the performance for sparse and other random access algorithms.

In addition to the L1 and L2 caches, the GPU also has caches related to traditional graphics functions. The constant memory cache is one example, which is typically used for arguments sent to a CUDA kernel. It is a cache tailored for broadcast, in which all threads in a block access the same data. The GPU also has a texture cache that can be used to accelerate reading global memory. However, the L1 cache has a higher bandwidth, so the texture cache is mostly useful if combined with texture functions, such as linear interpolation between elements.

Further guidelines

The CPU and the GPU operate asynchronously because they are different processors. This enables simultaneous execution on both processors, which is a key ingredient of heterogeneous computing: the efficient use of multiple different computational resources by letting each resource perform the tasks for which it is best suited. In the CUDA API, this is exposed as *streams*. Each stream is an in-order queue of operations that will be performed by the GPU, including memory transfers and kernel launches. A typical use-case is that the CPU schedules a memory copy from the CPU to the GPU, a kernel launch, and a copy of results from the GPU to the CPU. The CPU then continues processing simultaneously as the GPU executes its operations, and synchronization is only performed when the GPU results are needed. There is also support for multiple streams, which can execute simultaneously as long as they obey the order of operations within their respective streams. Current GPUs support up to 16 concurrent kernel launches (NVIDIA 2011), which means that we can both have data parallelism, in terms of a computational grid of blocks, and task parallelism, in terms of different concurrent kernels. GPUs furthermore support overlapping memory copies between the CPU and the GPU and kernel execution. This means that we can simultaneously copy data from the CPU to

the GPU, execute 16 different kernels, and copy data from the GPU back to the CPU if all these operations are scheduled properly to different streams. In practice, however, it can be a challenge to achieve such high levels of task parallelism.

When transferring data between the CPU and the GPU, it can be beneficial to use so-called page-locked memory. Page locked memory is guaranteed to be continuous and in physical RAM (not swapped out to disk, for example), and is thus not pageable by the operating system. However, page-locked memory is scarce and rapidly exhausted if used carelessly. A further optimization for page-locked memory is to use write-combining allocation. This disables CPU caching of a memory area that the CPU will only write to, and can increase the bandwidth utilization by up to 40 % (NVIDIA 2011). It should also be noted that enabling error-correcting code (ECC) memory will negatively affect both the bandwidth utilization and available memory, as ECC requires extra bits for error control.

CUDA supports a *unified address space*, in which the physical location of a pointer is automatically determined. That is, data can be copied from the GPU to the CPU (or the other way round) without specifying the direction of the copy. While this might not seem like a great benefit at first, it greatly simplifies code needed to copy data between CPU and GPU memories, and enables advanced memory accesses. The unified memory space is particularly powerful when combined with mapped memory. A mapped memory area is a continuous block of memory that is available directly from both the CPU and the GPU at the same time. When using mapped memory, data transfers between the CPU and the GPU are automatically executed asynchronously with kernel execution when possible.

The most recent version of the CUDA API has become thread safe (NVIDIA 2011), so that one CPU thread can control multiple CUDA contexts (e.g., one for each physical GPU), and conversely multiple CPU threads can share a single CUDA context. The unified memory model together with the new thread safe context handling enables much faster transfers between different GPUs. The CPU thread can simply issue a direct GPU–GPU copy, bypassing a superfluous copy in CPU memory.

Profile driven development

A quote often attributed to Donald Knuth is that “premature optimization is the root of all evil” (Knuth 1974). The lesson in this statement is to make sure that the code produces the correct results before trying to optimize it, and optimize only where it will matter. Optimization always starts with identifying the major bottlenecks of the application, as performance will increase the most when removing these. However, locating the bottleneck is hard enough on a CPU, and can be even more difficult on a GPU. Optimization should also be considered a cyclic process, because after having found and removed one bottleneck, we need to repeat the process to find the next bottleneck in the application. This cyclic optimization can be repeated until the kernel operates close to the theoretical hardware limits or all optimization techniques have been exhausted.

To identify the performance bottleneck in a GPU application, it is important to choose an appropriate performance metric, and compare attained performance to the

theoretical peak performance. When programming GPUs there are several bottlenecks one can encounter. For a GPU kernel there are essentially three main bottlenecks: the kernel may be limited by instruction throughput, memory throughput, or latencies. It may, however, also be that CPU–GPU communication and synchronization is a bottleneck, or that other overheads dominate the run-time.

When profiling a CUDA kernel, there are two main approaches to locate the performance bottleneck. The first and most obvious is to use the CUDA visual profiler. The profiler is a program that samples different hardware counters, and the correct interpretation of these numbers is required to identify bottlenecks. The second option is to strategically modify the source code in an attempt to single out what takes most time in the kernel.

The visual profiler can be used to identify whether a kernel is limited by bandwidth or arithmetic operations. This is done by simply looking at the instruction-to-byte ratio, or in other words finding out how many arithmetic operations your kernel performs per byte it reads. The ratio can be found by comparing the instructions issued counter (multiplied with the warp size, 32) to the sum of global store transactions and L1 global load miss counters (both multiplied with the cache line size, 128 bytes), or directly through the instruction/byte counter. Then we compare this ratio to the theoretical ratio for the specific hardware the kernel is running on, which is available in the profiler as the ideal instruction/byte ratio counter.¹⁴

Unfortunately, the profiler does not always report accurate figures as the number of load and store instructions may be lower than the actual number of memory transactions (e.g., it depends on address patterns and individual transfer sizes). To get the most accurate figures, we can compare the run-time of different versions of the kernel: the original kernel, one *Math* version in which all memory loads and stores are removed, and one *Memory* version in which all arithmetic operations are removed, see Fig. 7. If the *Math* version is significantly faster than the original and *Memory* kernels, we know that the kernel is memory bound, and conversely for arithmetics. This method has the added benefit of showing how well memory operations and arithmetic operations overlap.

To create the *Math* kernel, we simply comment out all load operations, and move every store operation inside conditionals that will always evaluate to false. We do this to fool the compiler so that it does not optimize away the parts we want to profile, since the compiler will strip away all code not contributing to the final output to global memory. However, to make sure that the compiler does not move the computations inside the conditional as well, the result of the computations must also be used in the condition as shown in Fig. 8. Creating the *Memory* kernel, on the other hand, is much simpler. Here, we can simply comment out all arithmetic operations, and instead add all data used by the kernel, and write out the sum as the result.

If control flow or addressing is dependent on data in memory, as is often the case in discrete optimization, the method becomes less straightforward and requires special care. A further complication with modifying the source code is that the

¹⁴ The Visual Profiler 4.0 computes the instruction/byte ratio.

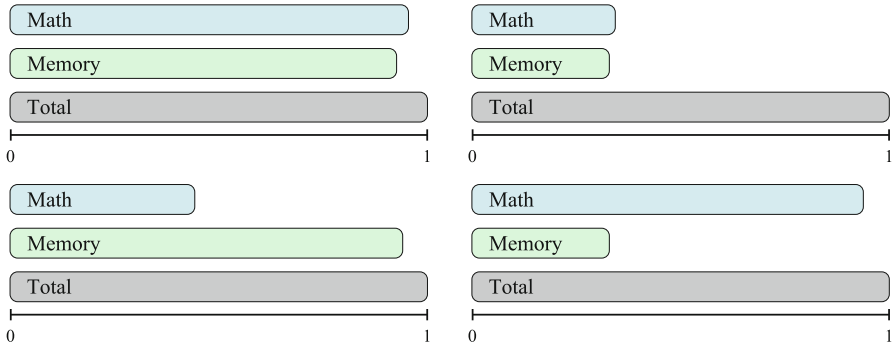


Fig. 7 Run-time of modified kernels which are used to identify bottlenecks: (*top left*) a well-balanced kernel, (*top right*) a latency bound kernel, (*bottom left*) a memory bound kernel, and (*bottom right*) an arithmetic bound kernel. “Total” refers to the total kernel time, whilst “Memory” refers to a kernel stripped of arithmetic operations, and “Math” refers to a kernel stripped of memory operations. It is important to note that latencies are part of the measured run-times for all kernel versions

```

1  __global__ void main(..., int flag) {
2      float result = ...;
3      if(1.0f == result * flag)
4          output[i] = value;
5  }
```

Fig. 8 Compiler trick for arithmetic only kernel. By adding the kernel argument *flag* (which we always set to 0), we disable the compiler from optimizing away the if-statement, and simultaneously disable the global store operations

register count can change, which again can alter the occupancy and thereby invalidate the measured run-time. This can be solved by increasing the shared memory parameter in the launch configuration of the kernel, `someKernel<<<grid_size, block_size, shared_mem_size, ...>>>(…)`, until the occupancy of the unmodified version is matched. The occupancy can easily be examined using the profiler or the CUDA occupancy calculator.

When a kernel appears to be well balanced (i.e., neither memory nor arithmetics appear to be the bottleneck), it does not necessarily mean that it operates close to the theoretical performance numbers. The kernel can be limited by latencies, which typically are caused by problematic data dependencies or the inherent latencies of arithmetic operations. Thus, if your kernel is well balanced, but operates at only a fraction of the theoretical peak, it is probably bound by latencies. In this case, a reorganization of memory requests and arithmetic operations can be beneficial: the goal should be to have many outstanding memory requests that can overlap with arithmetic operations.

Debugging

Debugging GPU programs has become almost as easy as debugging traditional CPU programs as more advanced debugging tools have emerged. Many CUDA

programmers have encountered the “unspecified launch failure”, which used to be notoriously hard to debug. Such errors were typically only found by either modification and experimenting, or by careful examination of the source code. Today, however, there are powerful CUDA debugging tools for commonly used operating systems.

CUDA-GDB is available for Linux and Mac, can step through a kernel line by line at the granularity of a warp, e.g., identifying where an out-of-bounds memory access occurs, in a similar fashion to debugging a CPU program with GDB. In addition to stepping, CUDA-GDB also supports breakpoints, variable watches, and switching between blocks and threads. Other useful features include reports on the currently active CUDA threads on the GPU, reports on current hardware and memory utilization, and in-place substitution of changed code in running CUDA application. The tool enables debugging on hardware in real-time, and the only requirement for using CUDA-GDB is that the kernel is compiled with the `gG` flags. These flags make the compiler add debugging information into the executable, and the executable to spill all variables to memory.

Parallel NSight is a plug-in for Microsoft Visual Studio and Eclipse which offers conditional breakpoints, assembly level debugging, and memory checking directly in the IDE. It furthermore offers an excellent profiling tool, and is freely available to developers. Debugging used to require two distinct GPUs (one for display, and one for running the actual code to be debugged), but this requirement has been lifted as of version 2.2. Support for Linux and the Eclipse development IDE was also released with version 2.2, making Parallel NSight an excellent tool on all platforms.

Profiling the local search example

For illustrative purposes, we will profile the local search code described in “[Programming example in CUDA](#)” and show how we determine its performance using Parallel NSight and the Visual Profiler tool. It is often good to get an overview of the application by generating a timeline of the different GPU operations and measure a set of metrics for the kernels, as shown in Fig. 9. The Visual Profiler also offers the option of displaying averaged measurements for each kernel, as shown in Fig. 10. It is often possible to identify application bottlenecks by examining these different measurements, and a few selected measurements are presented in Table 1. The table shows that the neighborhood evaluation kernel takes the most time, and if we double the problem size it completely dominates the run-time. This means that we should focus our optimization efforts on this kernel first.

The first thing we can look at for this kernel is the achieved FLOPS counter, which indicates a performance of 142 gigaflops. The hardware maximum is over 1 teraflop, meaning we are way off. However, if our kernel is memory bound, this might still be ok, as measuring gigaflops for a memory bound kernel makes little sense.

We have to acknowledge that our problem is quite small in terms of memory usage. For each node in our problem, we need 12 bytes of storage (4 bytes for its place in the solution, 2×4 bytes for the 2D coordinates), yielding a total of 12 KB.

Table 1 Selected profiling results for the local search example on a GeForce GTX 480

	nbh. evaluation	apply move
Avg. duration (μ s)	459.7	274.8
Achieved FLOPS: single GFLOPS	141.84	0
Registers/thread	23	18
Achieved occupancy	0.3	0.3
Static shared memory	0	4,096
Avg. dynamic shared memory	0	0
Instructions issued	6,452 288	49,327
Instruction replay overhead (%)	52.1	16
Branch efficiency (%)	99.8	94.7
Warp execution efficiency (%)	97	96.5
DRAM read throughput (bytes/s)	896	458
Global memory load efficiency (%)	6.9	0.9
DRAM write throughput (MBytes/s)	135.6	0.3
Global memory store efficiency (%)	100	12.5
Global store transactions	570	11
L1 global hit rate (%)	100	96.3
L1 global load miss	1,395	1,131
L2 hit rate reads (%)	100	100

The GTX480 has an L1 cache which holds 16 KB by default, more than enough to hold our whole problem. This is clearly visible in the profiling by a 100 % L1 Global Hit Rate counter as shown in Table 1. Each value is only read once from the global memory (DRAM), which explains the very low DRAM read throughput and efficiency. Unfortunately, this does not mean that our memory access pattern is well designed in general. In fact, reading the coordinates of a node in the cost computation means reading data at a random location, as the node is specified by a permutation (the solution). The effects of this can be observed when studying the instruction replay overhead. If threads within a warp cause non-coalesced reads, several instructions are necessary to read all needed data from memory, or as in our case, the L1 cache. For a coalesced read only one instruction would be necessary.

Each thread writes only the best move of the ones it has evaluated to global memory. As neighboring threads write to neighboring memory locations, we get coalesced writes and an excellent global memory store efficiency of 100 %. Good news also come from the problem of warp divergence. The branch efficiency counter is at 99.8 %, which means that of all branches taken, virtually none were divergent.

Summing up, we know now that our example has several shortcomings that limit its performance, and is operating far from the peak performance of the hardware. Some of these issues are simpler to address than others, but all require some redesign of the original local search algorithm. Therefore, we should revisit the algorithm with our new insight to achieve higher performance. In the [Appendix](#), we apply some simple adjustments to the kernels to achieve a considerable improvement.

Summary and conclusion

Over the last decade, we have seen that GPUs have gone from curious hardware being exploited by a few researchers, to mainstream processors that power the worlds fastest supercomputers. The field of discrete optimization has also joined the trend with an increasing level of research on mapping solution methods for these problems to the GPU. In the foreseeable future, it is clear that GPUs and parallel computers will play an important role in all of computational science, including discrete optimization, and it is important for researchers to consider how to utilize these kinds of architectures.

This paper is Part I in a series of two papers. Here, we introduce graphics processing units in the context of discrete optimization. We give a short historical introduction to parallel computing, and the GPU in particular. We also show how local search can be written as a parallel algorithm and mapped to the GPU with an impressive speed-up, yet there is still room for major improvements, as the implementation far from utilizes the full potential of the GPU. We also discuss general development and optimization strategies for writing algorithms that may approach peak performance. In Part II (Schulz et al. 2013), we give a literature survey focusing on the use of GPUs for routing problems.

Acknowledgments The work presented in this paper has been partially funded by the Research Council of Norway as a part of the Collab project (Contract Number 192905/I40, SMARTRANS), the DOMinant II project (Contract Number 205298/V30, eVita), the Respons project (Contract Number 187293/I40, SMARTRANS), and the CloudViz project (Contract Number 201447, VERDIKT).

Appendix: Profiling and improving the local search example

In this appendix we will show, how some simple adjustments in the kernels improve the GPU implementation considerably. The final kernels will not be rigorously profiled and optimized as the focus here is on illustrating some of the different aspects of GPU programming discussed in the paper.

Before we continue, we would like to briefly mention compute capability (CC). If GPU kernels are compiled for CC 1.x, the floating point arithmetic will not conform to the IEEE standard. This causes slightly different results in the distance computations, leading to a different solution on the GPU than on the CPU. With CC 2.0 and higher, however, floating point arithmetic on the GPU became IEEE compliant. For this reason, we will in the following discussion use results from compilation for CC 2.0. The arguments hold for compilation for CC 1.x and the profiling results are very similar. All profiling is done on a Geforce GTX 480, which is a Fermi class GPU.

From “[Profiling the local search example](#)” we know that our achieved occupancy is very low, only around 33 %. At the same time, the move evaluation kernel takes more time than the application kernel which chooses and applies the best move. The times spent on evaluation, application and other tasks are illustrated in Fig. 11 for a problem with 1,000 and 2,000 nodes, respectively. Using NSight, we profile the evaluation kernel with an occupancy experiment that tells us that in our chosen configuration we can have up to 8 blocks per SM yielding a total of

$8 \times 128 \times 15 = 15,360$ threads. So the reason for our low occupancy is actually too few threads, as we only have 8,192 and, thus, the GPU does not have enough active warps to hide latency. The kernel needs 23 registers per thread, so an occupancy of 100 % is not possible on our GPU. With some small experimentation, we find that using 20,160 threads spread over 105 blocks (7 per SM) with 192 threads each gives the best achieved occupancy of 78.38 % (compared to a theoretical limit of 87.5 % for this setup). Figure 11 shows that this GPU version 2 yields a faster move evaluation kernel, but a slower move application kernel. The reason for the latter is that choosing the best move now includes reducing 20,160 rather than 8,192 moves to the best one. For a problem of 1,000 nodes this increase in the application kernel actually dominates the whole runtime, such that the second GPU version is actually slower than the first version. However for larger problems (e.g., 2,000 nodes) the decrease in evaluation time leads to an improvement in overall runtime. As the GPU is intended for solving large problems, we will continue to focus on the evaluation kernel which still dominates the run time for bigger problems (see right part of Fig. 11).

In “Profiling the local search example”, we observed that although our memory access pattern in the evaluation kernel is not ideal, access to global memory is actually efficient as the whole problem of 1,000 nodes fits into the L1 cache. Clearly, for a problem of size 2,000 nodes this will no longer be the case with a 16 KB L1 cache, as illustrated in Fig. 12. However, on the Fermi architecture, we can configure the L1 cache to be either 16 or 48 KB. Setting it to 48 KB enables us to have problems of size up to about 4,000 nodes in the L1 cache. The positive effect can clearly be seen in GPU version 3 in Fig. 11 for the problem with 2,000 nodes.

The memory experiment in Fig. 12 also points out our bad memory access pattern (for loading). In average, we have 14.75 transactions per request. Ideal would be one (coalesced reads only), worst would be 32 (completely random memory access). As those are transactions between cache and registers, this number does not change with the increased L1 cache. So we need to improve our reading pattern from memory. The computation of the cost of one move includes two types of reads: find the node at a given position in the solution and get the coordinates of

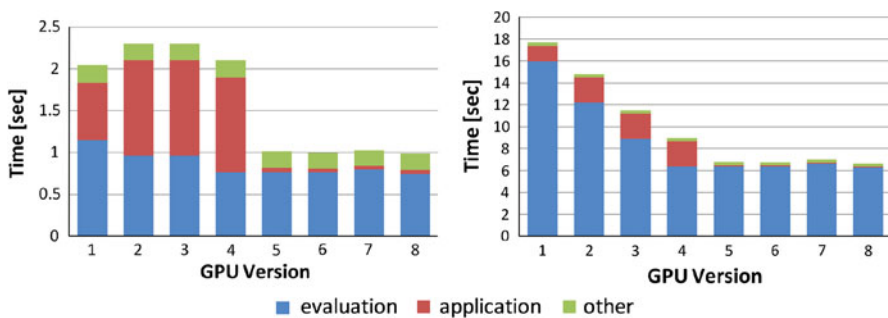


Fig. 11 Time spent on move evaluation, move choice and application and other tasks during local search for different versions of the GPU kernels on a problem with (left) 1,000 nodes over 2,500 iterations and (right) 2,000 nodes over 5,000 iterations

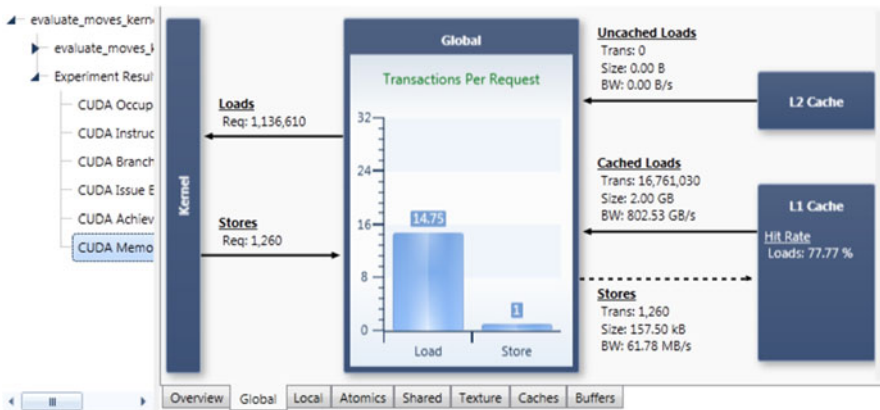


Fig. 12 Memory experiment statistics for global memory access for the evaluation kernel of GPU Version 2 for a problem with 2,000 nodes

this node. Actually this has to be done not just for one but several nodes. Nevertheless, as the solution can be any permutation of the nodes, it is impossible to predict which node will be at which position in the solution. Hence it is difficult to improve reading the coordinates of the nodes. However, we have control over the positions in the solution, or more exactly, we know which positions in the solution a move needs to access. So far, we split the lexicographically ordered moves into consecutive parts of a certain length, and one thread in the evaluation kernel evaluates the moves in its part.

```

1 //Loop through the subset and find best move
2 for (int i=first_move; i<first_move+num_moves_per_thread; ++i)
3 { ... }

```

In this way, we ensure that the moves are evaluated and compared to each other in exactly the same order as on the CPU. However, if two moves are equally good, it does not matter which of them is taken. So we can split the neighborhood into different parts, which enables a more efficient memory access pattern. In most cases, the first node to swap in the moves i and $i + 1$ will be the same and the second node in move $i + 1$ will be the neighboring node in the solution to the second node of move i . Hence, it makes sense to let neighboring threads k and $k + 1$ evaluate the moves i and $i + 1$ so we get coalesced memory access with respect to finding the node in the solution. Moreover, since one of the nodes is the same, this will also benefit the coordinate access for this node. We thus change the neighborhood part a thread has to evaluate from the moves $i, i + 1, \dots, i + N$ to the moves $j, j + M, j + 2M, \dots, j + NM$ where M is the grid size.

```

1 //Loop through the subset and find best move
2 for (int i=tid; i<number_moves; i += grid_size)
3 { ... }

```

The benefit is clear, as our transactions per request now reduce to 10.23 for the problem of size 2,000. The effects in running time both for the evaluation kernel and in total are again shown in Fig. 11 for this GPU version 4.

Profiling the GPU version 4 indicates that the memory access pattern is still the limiting factor for the evaluation kernel. Unfortunately it is hard to optimize this pattern further, since the access to the node coordinates goes through a, from a GPU point of view, random permutation. For this reason, we will stop here focusing on the evaluation kernel and instead concentrate on move reduction and application. For small problems, the move application kernel dominates the runtime. Similarly to the evaluation kernel, we have a very bad achieved occupancy of 2.2 %, which stems from the fact that we in total only have one block running on the GPU. This was done to keep the reduction code easy and readable and we will, therefore, not change this. The profiling tells us in addition, that again the memory access pattern is bad with around 17 transactions per global memory load request. When deciding on the best move, each thread in the application kernel first reads through a set of delta values.

```

1 //Reduce num_nodes_ elements in each thread
2 for (unsigned int i=tid*threads; i<(tid+1)*threads; ++i)
3 { ... }
```

This is the same type of bad memory access pattern used before in the evaluation kernel, hence we can apply the same remedy by changing which delta values are accessed by which thread.

```

1 //Reduce elements in each thread
2 for (unsigned int i=tid; i<deltas_size_; i += threads)
3 { ... }
```

Figure 11 shows that this drastically improves the runtime of the application kernel (GPU version five).

Reduction is a standard technique in GPU literature that has been studied with respect to good implementations (Harris 2011). With the improved memory access pattern, we now implement most of the ideas suggested in (Harris 2011) except loop unrolling. In GPU version 6 we, therefore, include loop unrolling in the reduction, but the effect on the runtime is only marginal in our case (see Fig. 11).

In the above paragraphs, we have shown that profiling and applying only minor changes to the original code resulted in a considerable speedup of the runtime for both larger and smaller problems. But how big problems can we actually solve? So far we only considered up to 2,000 nodes. How long does it take to evaluate the swap neighborhood for 10,000 nodes on the GPU? Running the program for a TSP of this size leads to a problem; both the CPU and the GPU version terminate with an error. The cause is a problem in our mapping from the linear move index to the indices of the nodes to be swapped. It includes floating point arithmetic, which on a computer never is exact. For such a big problem, the error in floating point computations causes the resulting indices to be invalid in a few cases, causing the

1,5	1,4	1,3	1,2		1,6	1,5	1,4	1,3	1,2
2,5	2,4	2,3	4,5		2,6	2,5	2,4	2,3	5,6
3,5	3,4	3,4	3,5		3,6	3,5	3,4	4,5	4,6
4,5	2,3	2,4	2,5		4,6	4,5	3,4	3,5	3,6
1,2	1,3	1,4	1,5		5,6	2,3	2,4	2,5	2,6
					1,2	1,3	1,4	1,5	1,6

Fig. 13 Improved mapping between linear index and node indices for (*left*) even and (*right*) odd number of nodes. The gray marked cells are the ones used

termination of the programs. On the CPU, the solution is simple. In fact, if one would implement local search for this neighborhood on the CPU without thinking about the GPU, most people would not use a mapping but instead two loops, where the first runs through the valid indices for the first node and the second loop through the valid ones for the second node.

```

1 for (unsigned int x=1; x+2 <= num_nodes; ++x) {
2     for (unsigned int y=x+1; y+1 <= num_nodes; ++y) {
3         ...
4     }
5 }

```

This eliminates the need for the demanding floating point arithmetic and, thus, not only enables very large problem sizes but in addition improves the runtime on the CPU. The latter fact illustrates also the limited usefulness of our GPU vs. CPU speedup statement mentioned earlier when introducing the “[Programming example in CUDA](#)”. For a more detailed discussion about usefulness and limits of such GPU vs. CPU speedup measurements see “Lessons for future research” in the second part of this paper (Schulz et al. 2013).

For the GPU, the floating point arithmetic problem is not so easy to solve. The square root in the mapping comes from solving a quadratic problem. Goldberg (1991) suggests to rearrange the quadratic formula to avoid catastrophic cancellation errors. This is done in GPU version 7, but in our case it does unfortunately not eliminate all possible sources of catastrophic cancellation. Hence, this GPU version still terminates with an error for a TSP of size 10,000. In (Schulz 2013), a different mapping between the linear move index and the node indices is suggested. The idea of this mapping is to consider the lexicographically ordered indices as a triangle and then cut the triangle top and rotate it to create a rectangular indexing scheme. The same idea can alternatively be described as considering the lexicographical triangle as half of a rectangle and filling the other half with the same indexing scheme in reverse order as done in Fig. 13. Then a simple rectangular mapping can be used to compute for a move its cell in the table. From this it is easy to deduce the node indices. This mapping consists only of integer arithmetic, hence eliminating any floating point errors. Our GPU version 8 using this mapping is able to evaluate 500 iterations for a TSP with size 10,000 in roughly 22.5 s.

In this appendix, we showed how repeated profiling and modifying can lead to a significant improvement in the efficiency of a GPU implementation. This just

emphasizes the importance of considering the architecture of the GPU when programming it. At the same time, we illustrated that comparing the CPU vs. the GPU can be unfair. Finally, the mapping between move number and thread number showed the importance of rethinking and adjusting an algorithm when implementing it on the GPU.

References

- Barker K, Davis K, Hoisie A, Kerbyson D, Lang M, Pakin S, Sancho J (2008) Entering the Petaflop Era: the architecture and performance of roadrunner. *Supercomputing*
- Bixby RE (2002) Solving real-world linear programs: a decade and more of progress. *Operations Res* 50:3–15
- Brodtkorb AR, Dyken C, Hagen TR, Hjelmervik JM, Storaasli O (2010) State-of-the-art in heterogeneous computing. *Sci Progr* 18(1):1–33
- Brodtkorb AR, Sætra ML, Altinakar M (2012a) Efficient shallow water simulations on GPUs: implementation, visualization, verification, and validation. *Comput Fluids* 55(0):1–12
- Brodtkorb AR, Sætra ML, Hagen TR (2012b) GPU programming strategies and trends in GPU computing. *J Parallel Distrib Comput*
- Chen T, Raghavan R, Dale J, Iwata E (2007) Cell broadband engine architecture and its first implementation: a performance view. *IBM J Res Dev* 51(5):559–572
- Fernando R, Kilgard M (2003) *The Cg tutorial: the definitive guide to programmable real-time graphics*. Addison-Wesley Professional, Boston
- Goldberg D (1991) What every computer scientist should know about floating-point arithmetic. In: *Numerical Computation Guide*, Appendix D, pp 171–264
- Harris M (2011) *NVIDIA GPU Computing SDK 4.1: optimizing parallel reduction in CUDA*
- Intel (2012) Intel's microprocessor export compliance metrics. <http://www.intel.com/support/processors/sb/CS-017346.htm>
- Knuth DE (1974) Structured programming with go to statements. *Comput Surv* 6:261–301
- Luna F (2012) *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning and Information, Boston
- Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, Upton M (2002) Hyper-threading technology architecture and microarchitecture. *Intel Technol J* 6(1):1–12
- Menabrea LF (1842) Sketch of the analytical engine invented by Charles Babbage. *Bibliothèque Universelle de Genève*
- Micikevicius P (2010a) Analysis-driven performance optimization. In: 2010 GPU Technology Conference, session 2012 (Conference presentation)
- Micikevicius P (2010b) Fundamental Performance Optimizations for GPUs. In: 2010 GPU Technology Conference, session 2011 (Conference presentation)
- NVIDIA (2011) *NVIDIA CUDA Programming Guide 4.1*
- NVIDIA (2012) <http://nvidia.com>
- Owens J, Houston M, Luebke D, Green S, Stone J, Phillips J (2008) GPU Computing. *Proc IEEE* 96(5):879–899
- Partyka J, Hall R (2012) Vehicle routing software survey—on the road to innovation. *OR MS Today* 39(1):38–45
- Schulz C (2013) Efficient local search on the GPU—investigations on the vehicle routing problem. *J Parallel Distrib Comput* 73(1):14–31 (Metaheuristics on GPUs)
- Schulz C, Hasle G, Brodtkorb AR, Hagen TR (2013) GPU Computing in discrete optimization—Part II: survey focused on routing problems. *EURO J Transp Logist*
- Shreiner D, Group TKW, Licea-Kane B, Sellers G (2012) *OpenGL programming guide: the official guide to learning OpenGL*, 8th edn. Addison-Wesley, Boston
- (2011) Top 500 supercomputer sites. <http://www.top500.org/>