Fast Evolutionary Algorithms: Comparing High Performance Capabilities of CPUs and GPUs

Johannes Hofmann,* Dietmar Fey

Chair of Computer Architecture
University Erlangen-Nuremberg

Abstract: We use Evolutionary Algorithms (EAs) to evaluate different aspects of high performance computing on CPUs and GPUs. EAs have the distinct property of being made up of parts that behave rather differently from each other, and display different requirements for the underlying hardware as well as software. We can use these motives to answer crucial questions for each platform: How do we make best use of the hardware using manual optimization? Which platform offers the better software libraries to perform standard operations such as sorting? Which platform has the higher net floating-point performance and bandwidth? We draw the conclusion that GPUs are able to outperform CPUs in all categories; thus, considering time-to-solution, EAs should be run on GPUs whenever possible.

1 Introduction

1.1 Evolutionary Algorithms

Evolutionary algorithms (EAs) are a popular form of metaheuristic optimization algorithms. They work by maintaining a population of individuals, also called candidate solutions. The internal representation of such a candidate solution is called a chromosome or genotype. The most popular representations of individual genes are in form of floating-point numbers, integers, or bits.

A fitness function is used to assign each individual a fitness value, indicating the quality of a certain candidate solution. The algorithm repeats the following steps iteratively until a termination condition, for example reaching a fitness threshold or some runtime constraint, is met:

i. Based on their quality, parent individuals are chosen from the population.
ii. During variation, the crossover operator derives a number of offspring from the parent population; after that, the resulting offspring undergo mutation, which introduces random change into their genotypes.

*johannes.hofmann@cs.fau.de

iii. The newly created offspring are evaluated using the fitness function and, depending on the survivor selection strategy, replace less promising individuals to make up the next generation.

Although EAs are generally less time- and resource-intensive than exact methods, they can still be resource-demanding for large problems, evoking the need for efficient parallelization.

1.2 Evaluation Hardware

To ensure contemporary results, we chose Nvidia's latest Tesla K20, which is based on the new Kepler architecture, from the high-end GPU segment as reference GPU. The price of the card ($ 4,400) served as a basis for selecting a corresponding CPU counterpart. In addition to the high-end GPU, we also chose to include results of the much cheaper ($ 520) GeForce GTX 670 consumer-grade GPU.

Both GPUs feature Nvidia's new streaming multiprocessor architecture (SMX), which has seen an increase in the number of CUDA cores (roughly equivalent to CPUs' SIMD-lanes) from 32 to 192 (single-precision) resp. 16 to 64 (double-precision), offering a significant boost in performance. The consumer card comes equipped with 7 Streaming Multiprocessors (SM), which allows for a peak performance of 2.5 SP TFlop/s, while the high-end Tesla model features 13 SM, yielding a peak performance of 3.5 SP TFlop/s. The theoretical bandwidth (BW) of the GeForce and Tesla cards is 192 GB/s resp. 208 GB/s.

Key differences between the GeForce and Tesla card include the unlocked double precision floating point performance: as we can infer by comparing the ALUs of different precision, the Tesla can theoretically achieve 1/3 of its SP performance when doing double-precision calculations; the GeForce, on the other hand, can only achieve 1/24 of the SP performance, because its DP ALUs are artificially throttled by a factor of eight. Additionally, the Tesla cards are available with a larger on-board memory (up to 6 GB compared to 4 GB of single-chip GeForce cards) and offer ECC-protection for said memory.

On the CPU end, we chose a two-socket system, employing Intel Xeon E5-2687W CPUs. These processors are based on Intel's latest Sandy Bridge microarchitecture, which introduced Advanced Vector Extensions (AVX); these new vector extensions double the vector register width of Streaming SIMD Extension (SSE) from 128 bit to 256 bit, essentially doubling the peak floating-point performance. Each CPU features eight cores with 2-way SMT and has a a theoretical performance of 397 SP GFlop/s, yielding a total node performance of 794 GFlop/s. Because the CPU uses the same ALUs for single and double precision arithmetic, the DP peak performance is exactly half of the SP performance. With a peak bandwidth of 51.2 GB/s per NUMA domain, the CPU system can theoretically attain a bandwidth of 102.4 GB/s.

For quick reference, a summary of key performance data has been compiled in Table 1. Note that the number of cores refers to physical cores per node in the CPU case and to the SM count in the GPU case. To calculate the peak SP GFlop/s rate we multiply the core

Hardware	Cores	Core Clock	Peak SP/DP Perf.	Peak BW
2×Xeon E5-2687W	16	3.1 GHz	794/397 GFlop/s	102 GB/s
GeForce GTX 670	7	915 MHz	2,459/102 GFlop/s	192 GB/s
Tesla K20	13	706 MHz	3,524/1,175 GFlop/s	208 GB/s

Table 1: Key Performance Data of the Evaluated Hardware.

count, core frequency, and the SIMD-width of the vector units (8 for AVX, 192 for SMX); we also consider an additional factor of two to account for two separate vector addition and vector multiplication pipelines on the CPU side, as well as the fused multiply-add (FMA) operation on the GPU side.

2 Implementation

2.1 Random Number Generation

Evolutionary Algorithms are probabilistic algorithms that make heavy use of random numbers. Not only do they require random number generation to seed the initial population, but also for the probabilistic steps during the algorithm, e.g. random values that determine whether to mutate a gene, where to choose the crossover point, which opponents to pick for tournament selection when choosing promising parents, etc. This is why we decided to implement a very fast pseudo random number generator for both architectures. We chose a Linear Congruential Generator (LCG), because it is both simple and, using several tweaks, can be implemented efficiently in hardware. A LCG works by computing a new random number x_{i+i} from an existing random number x_i with the help of a multiplier a, an increment b, a modulus m, and a start or seed value of x_0: $x_{i+1} \equiv ax_i + b \pmod{m}$. By choosing a modulus of $m = 2^{32}$ we can ignore the modulo operation, because the registers containing our value will simply overflow if we use unsigned integers as data type. Values a and b were chosen according to literature [PTVF07] to achieve a period of m.

We started our optimization attempts with a single-core version for the CPU written in C that generates 100 billion random numbers. The runtime[1] of this initial version was 129.4 s. After devising a simple performance model, it becomes clear that this version performs almost perfectly. The general purpose integer multiplication instruction has a latency of three cycles, while the addition instruction has a latency of only one cycle. Since we have a data dependency, it will take a single core four clock cycles to generate a random number. Thus, at 3.1 GHz it should take us 129.0 s to generate 100 billion random numbers, if each one takes four cycles to compute.

The loop overhead (increment, compare, branch) can easily be handled by the core in

[1]To increase accuracy, all reported measurements in this paper were performed ten times. The relative standard deviation (RSD = $100/\overline{x}\sqrt{(n-1)^{-1}\sum_{i=1}^{n}(x_i-\overline{x})^2}$) for all measurements is below 1%.

four clock cycles, especially when taking Sandy Bridge's macro-op fusion, which allows the fusion of the compare and branch instruction into a single micro-op, into account. Table 2 contains a possible mapping of instructions to issue ports. Note that while all three listed issue ports contain an ALU and can thus perform integer additions and logical operations, only issue port one can perform integer multiplications; also, only issue port five can be used for branching (information taken from Intel's Optimization Reference Manual [Int12]).

Cycle	Port 0	Port 1	Port 5
1	`imul` (1/3) / `add`	`incr` / —	— / `jb`
2	`imul` (2/3) / `imul` (1/3)	`cmp` / —	—
3	`imul` (3/3) / `imul` (2/3)	— / `incr`	—
4	`add` / `imul` (3/3)	— / `cmp`	`jb` / —

Table 2: Possible Mapping of Instructions to Issue Ports for Hardware Thread Zero (left) and Hardware Thread One (right).

The next step to improve this single-core version is to make use of Hyperthreading (HT), also called Simultaneous Multithreading (SMT). In our code, the add operation has to wait for the multiplication to complete. In the mean time, the pipeline is unused. If we use the other hardware thread of our core to generate random numbers as well, we can improve the pipeline efficiency by 100% (see Table 2). Using this revision, our program takes 64.7 s to complete—a speedup of exactly 2.0.

So far, we have only been considering scalar operations. However, AVX offers up to 8-way SIMD with its 256 bit vector registers. Unfortunately, AVX extensions only exist for floating point operations, which is why we have to use legacy SSE instructions to vectorize integer operations. The `pmulld` instruction multiplies four 32 bit integers in the lower half of the AVX vector registers, while the `paddd` instruction will add four 32 bit integers. After making use of these functions via intrinsics, our SMT version now takes 24.3 s to finish—a speedup of 2.66 compared to the previous SMT version.

We would expect a speedup of four when replacing the scalar instructions for vectorized ones, but, as it turns out, the latencies for the vectorized instructions are somewhat different from the scalar ones. While Intel's manual claims that a `pmulld` instruction takes only three cycles to complete on the Sandy Bridge microarchitecture, a micro-benchmark devised by us to measure instruction latencies revealed that it in fact takes five clock cycles. This means that instead of waiting four clock cycles for one random number in the scalar case, we have to wait six cycles (the vectorized add instruction, like its scalar counterpart, in fact only takes one cycle) for four random numbers in the vectorized case, which explains the below-expected speedup: $(4/1)/(6/4) = 2.\overline{6}$.

As we have now exhausted all single-core optimizations for the CPU, all that is left is to parallelize our random number generator across all cores using OpenMP. Albeit our machine has a NUMA topology, this task is straightforward, since all essential data is kept in registers. Although thread-pinning was used in each benchmark using `likwid-pin`

[THW10], pinning had no measurable effect for this particular benchmark. Using all logical cores, the parallel version takes 1.52 s to complete—a speedup of 16.0 compared to the vectorized SMT version and a speedup of 85.1 compared to the initial version.

Fortunately, we don't have to repeat this refinement process for the GPU. While kernels are written in a strictly scalar fashion, the CUDA programming model will automatically distribute execution to different Thread Processors and Multiprocessors, thereby taking care of vectorization and parallelization. To optimize the throughput of our LCG, all that is left to the programmer is to find a good loop unrolling factor and total thread count. Consistent with the recommendations of the CUDA programming model, we found that a very large number of threads benefits performance. For the GPUs we used 2^{22} threads together with 512 threads per block.

The runtimes for both GPUs, as well as a summarization of the previous CPU versions is shown in Table 3. For the GPU versions, the value in parenthesis is the speedup achieved in comparison to the fully optimized CPU version.

Hardware	Single-Core	SMT	Vectorization	Multi-Core
2×Xeon E5-2687W	129.4 s	64.7 s	24.3 s	1.52 s
GeForce GTX 670	—	—	—	0.42 s (3.6)
Telsa K20	—	—	—	0.29 s (5.2)

Table 3: Runtime and Speedups (in Parenthesis) of Different Versions to Generate 100 Billion Random Numbers.

Unfortunately we cannot judge the efficiency of the GPU code generated by the compiler, because there are no instruction latencies available for Nvidia's Parallel Thread Execution (PTX) pseudo-assembly language. Considering the optimized CPU version, our previous latency-based performance model suggests we reach 99.97% of peak performance with the measured runtime of 1518067 μs. However, even with this implementation pushing the CPU's limits, the GPUs easily dominate random number generation. The Tesla is about 5 times faster than the CPU and about 1.5 times faster than the GeForce card (which is roughly equivalent to the ratio of the frequency-ALU-product of both cards); the GeForce card still is 3.6 times faster than the optimized CPU version.

2.2 Variation Operators

The variation operators—mutation and crossover—in EAs are conceptually designed to model the behaviours of their biological prototypes.

As its name suggests, the mutation operator is performing mutations on genes with some low probability p_m. This mutation can be entirely random (e.g. setting an `int` variable to a random value) or less severe (e.g. adding a random value from a normal distribution with $\mu = 0$ to a floating point value). This operator is making heavy use of random numbers, because each gene of every individual of a generation is considered individually

for mutation.

The crossover operator is mimicking the process that is taking part during meiosis in a cell. In the first step, two individuals a and b are selected from the current population. Next, a random crossover point is established using a random number. The individuals' genotypes are then cut at the crossover point; then, the tail of individual a's genotype is appended to the head of individual b's genotype and vice versa. Crossover is happening for each offspring, and usually a lot more offspring than parents are generated. This abundant genetic diversity aides our EA not to get stuck in local optima. Since this operation can be very memory intensive, a benchmark of the memory bandwidth has been performed. The results are shown in Table 4.

Hardware	Peak Bandwidth	Net Bandwidth	Efficiency
2×Xeon E5-2687W	102 GB/s	65 GB/s	64%
GeForce GTX 670	192 GB/s	115 GB/s	60%
Tesla K20	208 GB/s	128 GB/s	62%

Table 4: Peak and Net Bandwidth for Evaluated Hardware.

For the CPU, the net bandwidth was measured using likwid to compensate against non-NUMA awareness of the original STREAM benchmark [McC07]; the GPUs' bandwidths were measured using a CUDA port of the STREAM benchmark. Interestingly, the efficiency of the memory system seems to be consistent across platforms: each specimen can only achieve around 60% of the advertised peak performance. With efficiency the same across platforms, the peak performance ratio becomes the net performance ratio also, i.e. the GPUs offer about twice the bandwidth than the CPU system.

2.3 Sorting

After crossover and mutation have been performed and a number of offspring has been created and evaluated, a new generation of fixed size is chosen from the parents and offspring. Usually this choice is deterministic and only the best individuals are allowed to enter the next evolutionary cycle. To select the best individuals, we have to sort our current population and the created offspring according to their fitness values.

This is part of the evaluation we rely solely on libraries, because existing—most likely heavily optimized—implementations for standard operations such as sorting should be valid representatives for the maximum achievable performance.

We used `__gnu_parallel::sort` from glibc, `tbb::parallel_sort` from Intel's Threading Building Blocks library, and `thrust::sort` from the Thrust library [HB10]. To the best of our knowledge, the only other comprehensive library collection for CUDA is the CUDA Data Parallel Primitives (CUDPP) Library. We chose not to include results obtained with this library, because CUDPP actually uses the Thrust library as its sorting backend and results of both libraries were identical.

A compilation of our findings can be found in Figure 1. Because the population sizes can vary greatly in EAs—generic algorithms typically use several hundred to several thousand individuals while it is not uncommon for genetic programming to use populations that comprise of several million individuals—we used a logarithmic scale on the x axis of the plot.

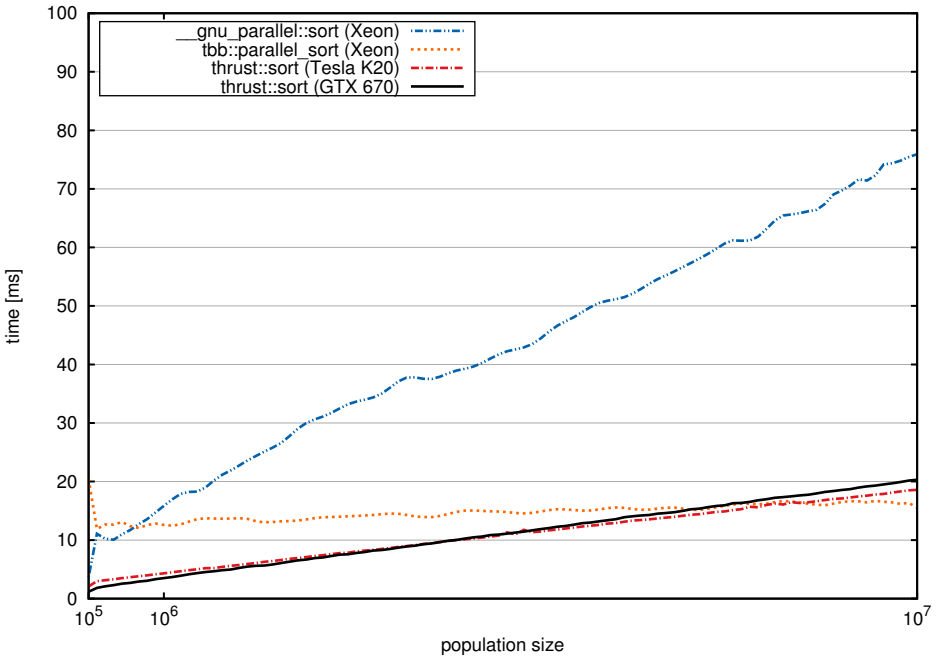

Figure 1: Time Spent Sorting Populations of Various Sizes.

The Thrust implementation appears to make good use of the underlying hardware, beating both CPU implementations for almost the entire range; there is, however, a break-even point somewhere before a population size of ten million individuals, meaning that Intel's implementation becomes faster than Thrust on the GPU for very large populations. However, we would like to note that populations beyond the ten million individual mark are found seldom in EAs. The implementation found in the GNU C Library remains behind.

2.4 Fitness Evaluation

Conceptually, fitness evaluation is at the heart of an EA. It assigns each individual a fitness value, on which

i. parent selection relies to pick promising parents; and
ii. survivor selection relies to ensure the "survival of the fittest."

The evaluation is also the most compute intensive function of an EA. Usually, evaluation involves the simulation of a physical process with some of the simulation parameters encoded in the evaluated individual's genotype. As these simulations tend do be floating-point intensive (we already dealt with integer performance in the Section 2.1), we evaluate and discuss the floating-point performance of the hardware here.

To measure the net performance for the CPU, a hand-written three-way modulo unrolled assembly kernel using the `vmulps`/`vmulpd` and `vaddps`/`vaddpd` instructions was used.

For the GPU, a PTX kernel using 16-way modulo unrolling performing only `fma.rn.f32` resp. `fma.rn.f64` instructions was used.

Hardware	Peak SP	Net SP	Peak DP	Net DP
2×Xeon E5-2687W	794	785 (99%)	397	393 (99%)
GeForce GTX 670	2,459	1,919 (78%)	102	121 (119%)
Tesla K20	3,524	2,347 (67%)	1,170	1,161 (99%)

Table 5: Peak and Net Single Precision and Double Precision GFlop/s Performance for Evaluated Hardware

Our findings are summarized in Table 5. Interestingly, the situation regarding efficiency is entirely different than our examination of the bandwidth in Section 2.2. Here, the GPUs cannot keep up with the CPU's efficiency. While the advertised theoretical single-precision performance indicates a Tesla K20 would be about 4.4 times faster than two Xeon E5-2687W CPUs, the impact of efficiency reduces the speedup to a factor of 3.0; the same reasoning applies to the slower GeForce card. Interestingly, the efficiency of double-precision arithmetic is close to the theoretical peak performance for the Tesla K20 GPU; the GeForce even exceeds the theoretical peak performance—probably the result of Nvidia favoring conservative over too aggressive throttling. Despite these facts, both GPUs still outperform the Xeons when it comes to absolute numbers of SP performance.

3 Conclusion

As we have shown in the previous Sections, the GPUs outperform a dual-socket high-end CPU system in all categories:

i. they perform random number generation about 3–4 times faster than the reference CPU node;
ii. both GPUs should be about twice as fast in variation operations, as they offer two times the net memory bandwidth;
iii. survivor or and parent selection is done approximately five times faster on the GPU, at least for typical populations sizes; and
iv. should the fitness function require floating-point operations, a GPU should offer a speedup of 2–3×.

The conclusion we can draw from this is that EAs should be run on GPUs whenever possible.

What remains to be discussed, is how easy one can tap into the performance of a platform. Compared to intrinsic or assembler programming to make use of vectorization, the CUDA programming model offers the SIMD performance "for free", i.e. without explicit optimization. The cost of this abstraction is that of efficiency, but as long as the peak performance of GPUs is a large enough factor higher than CPUs', GPUs can outperform them when it comes to evolutionary algorithms.

This fact is more and more becoming a problem when programming for CPUs. While todays' programmers are well aware of multi-core programming, very few make use of vectorization using intrinsics or even assembly code—even then, when not using them means that as much as 94% of performance will be lost in upcoming vector extensions, such as the second evolution of Intel's Advanced Vector Extensions for its Haswell microarchitecture (AVX2) or Knights Corner's Initial Many Core Instructions (IMCI).

ISPC [PM12] could be Intel's answer to that problem. It is a concept that bears striking resemblance to the CUDA programming model, in which a processor's SIMD-lanes are assigned so-called program instances (compare to CUDA threads, which run on Streaming Processors). New vector instruction sets like AVX2 and ICMI appear to work in tandem with this new programming paradigm by providing vector mask registers. These mask registers can be set by `if` statements in the code and are used to mask out SIMD lanes for instructions—a concept somewhat similar to CUDA's warp divergence, which in fact is internally implemented using the very same predication concept. Also included in these new vector instruction sets are scatter and gather operations, which aim to provide an efficient means to fill large vector registers from non-contiguous memory. Overall, these technical innovations herald the advent of vector registers of ever increasing width, a technique that just Nvidia has been successfully employing for years.

References

[HB10] Jared Hoberock and Nathan Bell. Thrust: A Parallel Template Library, 2010. Version 1.3.0.

[Int12] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-026. April 2012.

[McC07] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[PM12] Matt Pharr and William R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing Conf.*, May 2012.

[PTVF07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[THW10] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.