A Project-based HPC Course for Single-box Computers

Carlos Bederián
Instituto de Física Enrique Gaviola
Facultad de Matemática, Astronomía, Física y Computación
CONICET
Av. Medina Allende s/n, Ciudad Universitaria
X5000HUA, Córdoba, Argentina
Email: bc@famaf.unc.edu.ar

Nicolás Wolovick
Facultad de Matemática, Astronomía, Física y Computación
Av. Medina Allende s/n , Ciudad Universitaria
X5000HUA, Córdoba, Argentina
Email: nicolasw@famaf.unc.edu.ar

***Abstract*—Throughout three iterations and six years we have developed a project-based course in HPC for single-box computers tailored to science students in general. The course is based on strong premises: showing that assembly is what actually runs on machines, dividing parallelism in three dimensions (ILP, DLP, TLP), and using them incrementally in a single numerical simulation throughout the course working in interdisciplinary pairs (CS, non-CS). The final goal is to explore how to use all the available transistors in a die. Assembly proved a great tool to show how bare-metal works, an alternative-semantics approach to programs, and a tool to demystify compiler technology. Parallelism is tackled gradually with a clear division into instruction, data, and thread parallelism. GPUs, through CUDA in particular, are used as a radically different approach to the three dimensions of parallelism. Each dimension is explored in a gradual manner, starting from a sequential toy-yet-interesting numerical simulation. After using each form of parallelism and submitting a short report, the experiences are put together in group discussion unveiling the strengths and weaknesses of each form of parallelism for each class of numerical simulation. Although there is a high variance in the students' background, CS and non-CS students pair well in project development, generating understanding and value of the disciplines. The experience proved successful, with former students producing parallel accelerated code of their own in their disciplines.**

I. Introduction

After two years of experience as a satellite course for Thomas Sterling's LSU course "High Performance Computing: Model, Methods and Means" –HPCMMM– in 2009 and 2010, and having acquired some experience in interdisciplinary work for numerical code parallelization in CPUs and GPUs, our GPGPU Computing Group started an undergraduate and graduate course for science students in Parallel Computing. Given our previous experience with HPCMMM, we cropped the contents to all the parallelism that fits in a single computer, namely Instruction Level Parallelism (ILP), Data Level Parallelism (DLP) and Thread Level Parallelism (TLP), since we thought that there was a need to dive slightly deeper than Thomas' course but in a three month period.

The course has been given in 2012, 2014 and 2016 [1]. The first edition had the basic structure, namely all the available parallelism in a box plus GPU computing, but instead of having numerical codes to parallelize in all three levels, we had assignments for each of the four topics consisting of exercise guides with synthetic problems. In 2014 we switched from exercise guides to a single numerical simulation that had to be parallelized incrementally throughout the course, first in CPU for the three dimensions, and with a CUDA port to GPUs later. The four assignments were graded in groups of two, and given the interdisciplinary aspect of our students (computer scientists, physicists, astronomers, chemists, applied mathematicians and engineers) we decided to pair CS with non-CS students with the expectation of repeating the positive experiences we had previously in our interdisciplinary work with other sciences in parallelizing numerical code. The move proved right, and we basically repeated the scheme in 2016, adding a simple yet interesting summary-class, where all the groups discussed their outcomes in each of the four assignments with teachers as moderators.

Computational resources increased in each iteration of the course. We started with a single chip 6-core Nehalem Intel Core i7, 24 GiB RAM, with two NVIDIA Tesla C2070 GPUs. In 2014 we replaced the Fermi cards with two NVIDIA Tesla K40 but keeping the CPU configuration. In 2016 our Faculty bought two dual 6-core Haswell Intel Xeon E5-2620 v3 with 128 GiB RAM each, where we fitted two NVIDIA GTX 980 and one Titan X Maxwell cards. The growth in computational power was enormous, while we had more and more up-to-date platforms. In 2016 edition we were just one year behind of current hardware.

Students increased and diversified in each course edition, from a dozen of mainly CS students in 2012, to forty this year where non-CS students accounted for 60% of the class. Non-CS students ranged physics, astronomy, mathematics, engineering (aeronautical, civil, electronic) and chemistry. These students were all doing the course as graduates, while CS students were mostly undergraduates taking the course as an elective.

We had CS undergraduate students with strong knowledge of compilation processes, programming paradigms, concurrency, operating systems, and assembly. However none of them had had a real need for GFLOPS of computational power or GiB/s of memory bandwidth so far. On the other hand

EduHPC2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3827-5/16/$31.00

astronomers, civil engineers, etc. were eager to squeeze every last drop of performance from their computational resources in order to boost their domain-specific codes in the era of massively parallel processors, but where being held off by computer architecture illiteracy. The matchmaking was then tried as a possible solution.

The authors were in charge of the course throughout the three editions, but the teaching conditions improved. We went from having this course as a side-load to the Networks course in 2012 to teaching this course exclusively in 2016, together with three alumni from the 2014 edition as *ad honorem* teaching assistants.

II. The three dimensions of parallelism

We tackled parallelism by exploring each of the three dimensions available in a single computer. First Instruction Level Parallelism (ILP) that comes from the superscalar, out-of-order computational units inside modern CPUs. Second we addressed the Data Level Parallelism (DLP) available in their vector units. Finally Thread Level Parallelism (TLP) to profit the multi-core capabilities of post-Pentium 4 single-chip CPUs. We also devoted part of the course to GPU computing. Graphic processing units have ILP, DLP and TLP, however they are implemented in ways very different from CPUs'.

We used `C99` as the language for running examples, mixed with some `FORTRAN` to attract science students using this language.

A. ILP

Although most modern central processing units are superscalar and out-of-order, this fact is not very well-known by scientific code users and writers. The compilation process including intermediate stages, is one of the main topics of this part of the course. The final goal is to teach that machine code is what the bare-metal executes and from that point, explain that the single stream of instructions is dependency-analyzed and parallelized on-the-fly by the microprocessor. Through simple but meaningful programs, we show how clear and efficient the generated assembler is. Moreover, compiling to assembly language, typically through the `-S` compiler flag, is really handy to show how optimization flags work or don't. Students understand the importance of different optimization flags and play with the idea of compiler as a *semantics preserving transformation*, that is also a valuable tool to detect correctness and performance bugs in the code.

We focus on loop unrolling as the main technique to break loop data-dependence, and we measure the impact of the applied optimizations using `perf stat` to read the instructions per cycle –IPC– counter. This tool is extremely valuable in assessing that the code plus compiler flags is doing what we expect.

Memory hierarchy is also an important topic of this section, and we stress its parallel aspect. Retrieving lines to cache is a parallelization to improve memory bandwidth. Latency aspects of each hierarchy level is also considered, and standard cache-friendly techniques are explored. We emphasize the role of microprocessor registers as the only source of non-wait memory in the hierarchy, a concept that is crucial later in GPU computing.

Parallelism is presented as a solution to the law of diminishing returns in performance per transistor area, and we used Pollack's Rule [2] to support this claim. Execution unit utilization is also covered, showing that symmetric multithreading [3] –SMT– is just a cheap way to improve execution unit utilization through explicit parallelism.

Limits and models of modern microprocessor are also presented. Floating point peak performance as well as memory bandwidth are shown for modern chips, and here the roofline model [4] fits well to tie both aspects. We go further quoting Bill Dally [5] in what is the current and future problem of HPC programmers: "*All performance is from parallelism, machines are power limited (efficiency is performance), machines are communication limited (locality is performance)*". This statement will be repeated throughout the course in each dimension of CPU and GPU parallelism.

B. DLP

The simple model of Data Level Parallelism or vector computers is presented through an exploration of the SSE/AVX Intel instruction set architecture. We show examples of reduction DLP using vertical and horizontal parallel sums. Here the important topic of non-associativity of floating point arithmetic appears. We stress the complexity of the vector ISA "*where operations are either available or not available for particular data types with little rhyme or reason*" [6].

In this sea of diversity, compiler autovectorization is presented more as a platform-independent code generator than an optimizer, since different techniques (loop peeling, memory alignment, etc.) need to be applied in order to meet the requirements of the compiler to use this part of the ISA. Autovectorized code is also vector-width independent, which is important for the current transition from 128-bit SSE vectors to the wider vectors in AVX and AVX-512.

We specially show how to write code where the control diverges between vector lanes using comparison, and masked or blended instructions. This will be important later on for GPU computing.

One class is devoted to helping the compiler autovectorizer [7], and another class is dedicated to showing how to write a stencil code similar to `heat` using DLP through Intel intrinsics [8] for SSE/AVX.

We use `sgemm` computation to demonstrate how efficient a native library can be implementing a seemingly naive problem compared to the code that people usually write, even when using good HPC practices.

C. TLP

Thread Level Parallelism is presented as the final dimension of parallelism, covering asynchronous cores of computation. The MIMD model is given as well as the main aspects of shared and coherent memory hierarchy. The examples are coded using OpenMP [9] directives and library.

In this part we also use assembly generation to show the semantics of OpenMP constructs, as well as intermediates stages of the `gcc` compiler using `-fdump-tree-omplower` and `-fdump-tree-ompexp`[1]. The tool is useful to disambiguate the natural language specification of the OpenMP standard.

To better understand scaling and efficiency concepts, some classical examples like `sgemv` and `sgemm` from BLAS level 2 and 3 are reviewed. We also compare with optimized BLAS libraries to stress the fact that it is very difficult to achieve that level of efficiency with respect to theoretical FLOPS peak.

We execute the code in different multicore platforms, ranging from old Opterons to recent Haswell architectures. These examples are the basis to discover some ccNUMA performance problems, that are exposed using `numactl` to control affinity and `numatop` to check local/remote memory access for running code. We also explore compiler autoparallelization.

D. ILP, DLP, TLP from a GPU perspective

Once the main concepts of a modern CPU architecture have been presented and practiced, the learning curve for GPU computing is somehow leveraged. We borrow concepts like vector units, the memory wall problem, MIMD, ILP, divergent lanes, but in a new perspective given by the modern post-G80 NVIDIA GPU architectures. Memory latency hiding through over-subscription of shaders fits nicely with n-way SMT. Divergent threads performance penalty is explained using the previous divergent lanes vector examples. The students can understand the GPU as a many-core computing unit (shaders), each comprising n-way SMT (block warps) and wide vector units (warp) with special hardware for handling divergent lanes transparently.

What differs is memory hierarchy, and this is one aspect we emphasize a lot showing the inverted memory hierarchy pyramid [10], and a cache that is only for spatial locality. Broadcast memory, and the shared memory as user-managed cache, are also shown as peculiarities of this memory architecture with respect to the CPU counterpart.

Since there is no common ancestor for CPUs and GPUs, we run through the historical development of GPUs. This helps to understand the idiosyncrasy behind many graphics processing units implementation decisions. This also helps to predict some aspects of the GPUs as texture interpolation ISA instructions or native computation of $\sin, \cos$ also in the ISA which are not present in the CPU counterparts.

We show the development of GPU architectures (Fermi, Kepler, Maxwell) in a similar way to what we did in CPU architectures (Nehalem, Sandy Bridge, Haswell). In the current edition of the course, all the examples have been run on Fermi, Kepler and Maxwell and this accumulation of performance profiles help to portray each architecture.

We continued exposing what is really run in the hardware, the `SASS` assembly, this time mediated by the intermediate `PTX` assembly. Again this was very useful to show improvements in the ISA that are very important to performance, for example compiler generating `ldg` instructions for cached global access and the Maxwell shared atomics. We also point out instructions without CPU counterparts like `tex2D` and `cos` that are originally tailored to the graphics pipeline.

The programming language and library is CUDA [11]. We cover standard algorithmic patterns like maps and reductions with special emphasis on how to profit from the computing and memory architecture. We display in a rather simple form, what was quoted from Bill Dally, locality is the main source of performance and by using the three level hierarchy (registers, shared and global) we show a reduction example code where an enormous boost in performance is achieved. Again `sgemm` computation is the bar to measure how hard is to achieve peak performance, and that native libraries are nearly impossible to beat in terms of efficiency. We end with a rather complex but useful parallelization, namely a scan reduction shaped by the three level memory hierarchy.

III. Projects

Most of the codes are in C, just `tiny_manna` and `tiny_sph` are in a proper subset of C++. All of them are simple enough so that the main source of computation easily fits in a page of code. The projects where chosen more by professors' familiarity with the numeric code than any other criteria, in order to effectively guide students throughout the parallelization and optimization process.

The numerical simulations are:

- `endoh1` referenced as the "Most complex ASCII fluid dynamics" and given an honorable mention at IOCCC'12 is a smoothed-particle hydrodynamics –SPH– fluid simulator for the console. The starting point for this project is based on the deobfuscated version of the IOCCC submission.
- `heat` this classical code solves the 2D heat equation numerically using a stencil of 4 neighbors for a fixed temperature boundary condition and a source point. The initial project code iterates a diffusion step until either a certain global error, or maximum number of iterations is reached. The output is given as a `ppm` image. We have used this project in two GPU schools in the past.
- `hornschunck` the starting point is the source code available from [12] for the classical version of the Horn-Schunck optical flow algorithm [13]. The program takes the maximum number of iterations and α parameter and two images, computing the optical flow `flo` output that is converted to `png` for easy visualization. The project was proposed and successfully parallelized by a 2014 alumni doing his PhD in Computer Vision and Robotics.
- `navierstokes` we start from the classical paper [14] and source code for this approximate solution to the Navier-Stokes fluid equations, with fixed inputs in order to simplify debugging. Although this is one of the most complex numerical codes, this is balanced with a nicely written related paper and code. One of the course professors parallelized in CPU and GPU a magnetic hydro-

[1]Thanks to Diego Novillo for pointing this out.

dynamic –MHD– code, so there was previous experience with this kind of numerical code.

- `scan2d` built on previous experience on GPU Computer Vision algorithms [15] we provide a sequential version of the Integral Image algorithm. The code is really small, but its strong sequential character transforms this problem in one of the hardest. Its difficulty is leveraged with in-class examples of reductions and scans, that are the main ingredients of this code. The project is provided with a single test case that was carefully chosen to cover common bugs.
- `spmv` sparse solvers are used in many fields including computational fluid dynamics and probabilistic model checking, with Sparse Matrix-Vector multiplications taking up most of the computation. Built on previous experience [16], [17], [18], we gave a page-long sequential algorithm for the Compressed Sparse Row –CSR– representation, and a set of test cases. Similar to `scan2d`, it is extremely simple, but the non-contiguous memory access and load balancing problems put a high difficulty bar on this code.
- `tiny_ising` the 2D Ising model is a stochastic program to simulate a ferromagnetic system. The grid holds spin values in $\{-1, 1\}$ and the dynamics of the system change spins in typewriter order following the Metropolis Algorithm. The code combines massive use of random number generators, 4-neighbor stencils and the computation of global grid energy. This example was built on previous experience on the Potts model [19]. The results are not deterministic, but given the stochastic nature of the code, correctness can be checked by averaging repeated runs of the parallelized code against sequential code results.
- `tiny_manna` the Manna model [20] is for a two dimension cylinder-like sandpile represented by a vector where each item contains the pile height at that point. This is known as one-dimension Manna model. The dynamics for each site spreads the sand grains stochastically to the left and right neighbors if the pile is greater than one. This process repeats for a fixed number of iterations or until it reaches a steady state, with the number of active grains being printed in each iteration. This code was communicated and parallelized by Alejandro Kolton, a physics colleague working in the Centro Atómico Bariloche.
- `tiny_mc` this is a 2D numerical simulation of a 1 W point source heating in infinite isotropic scattering medium [21]. Previous experience in this code was gained in a lab session run by the authors in a GPU school. The simulation is a typical Monte-Carlo code using lots of RNG and very little memory to accumulate the shells of energy for the photon scattering. All the photons are independent of each other. Although the result is stochastic, the implementations could be tested against the serial version by averaging.
- `tiny_sph` this code implements a smoothed particle hydrodynamics (SPH) approximation for interactive/game effects [22] with an attractive OpenGL visualization. All particles can be independently updated and the interaction is local. Correctness checking is hard since it just displays a visual simulation. This numerical code was the latest addition of the course.

In Table I we summarize main aspects of each numerical algorithm. The numerical intensity [23] is defined as the FLOPS to memory access ratio where N is the problem size. A numerical intensity of 1 means one floating point operation per memory access and this describes a memory-bound problem, while the other end of the spectrum is N showing a FLOPS-bound problem. Data dependence is the the amount of parallelism available in the numerical simulation. Usually the numerical codes on a grid have two phases, one updating values and the other computing some global quantity. The first part can be parallelized using the checkerboard technique [19] for stencils of four neighbors. We are not trying to be precise but we would rather give an idea of how hard parallelization and memory optimization is.

IV. Project Results

Although we did not record quantitative information on the outcome of the last two editions of the project (2012 edition was not project-based), we obtained interesting general information on what happened.

The ILP lab was mostly a warm-up to get acquainted with the code and try different compiler switches. As ILP is hard to program explicitly, students were not expected to unroll loops by hand or do shuffle expressions to increase the Haswell eight port utilization. Vanilla numerical codes started with no optimization flag at all, so the message was clear: there is a lot to gain by just adding `-O1`. Depending on the code, students could increase performance with a base of 2x. Another important lesson learned was system-noise and repeated executions of measurements to obtain a stable mean and the variance to quantify the quality of the measurement. Perhaps this is well-known for science students in general, but our CS students have not been exposed to experimental science. The main outcome of this part was a lin-log graph plotting normalized computation time with respect to problem size vs. problem size. The students quickly understood whether their code was memory bound if there was a normalized time increase as the problem size grows.

For DLP the picture changed, since students had to decide where to invest their time budget: rewriting the code using intrinsics or trying to help the compiler autovectorizer [7]. In 2014 and 2016, we have seen both approaches with a good degree of success. In general, groups that chose AVX intrinsics obtained good speedups of at least 2x, while autovectorization was slightly worse. It was remarkable that although vector intrinsics were really hard to code, once the code was ready, it was correct and fast. Below we will compare this to TLP. Pure CPU-bound codes, namely `tiny_mc` started to excel, gaining a linear 8x after manual vectorization on a 8-float-

TABLE I
SUMMARY OF THE NUMERICAL CODES

Code	Numerical Intensity	Data Dependence	Observations
endoh1	~ 4	Sequential/checkerboard update	Complex numbers, hard to read source
heat	~ 4	Fully parallel update, reduction	Simple
hornschunck	~ 4	Fully parallel	
navierstokes	~ 4	Sequential/checkerboard update, reduction	Difficult to test for correctness
scan2d	1	Reduction	Hard to beat sequential, needs two passes in parallel
spmv	1	Reduction	Non-contiguous memory access, load balancing
tiny_ising	~ 4	Sequential/checkerboard update, reduction	RNG
tiny_manna	1	Fully parallel	RNG
tiny_mc	N	Fully parallel update, reduction	RNG
tiny_sph	$\sim m$, $m \ll N$	Fully parallel update	Hard to modify code, non-contiguous memory access

wide vector unit. For the rest, the gain was also important, making the effort valuable.

TLP had a lot to say from the results. The summary is that memory system works as a bottleneck in symmetric multiprocessing systems. While the `tiny_mc` photon code had 12x for 12 cores, stencil and reduction codes ran into a lot of trouble trying to achieve a speedup of more than 6x with 12 cores. A group in 2014 was so frustrated that they used `likwid` to finally assess that they had reached the Stream Benchmark [24] limit for the memory bandwidth. Here the main outcome was efficiency graph with respect to problem size for increased number of cores. Comparing DLP to TLP, the OpenMP decoration were easy to write but performance tuning was a difficult task.

Finally GPU using CUDA and running on Maxwell architecture lied somewhere between vectorization and multicore parallelization with respect to difficulty and acceleration. The teachers' suggested strategy was a step by step GPU port of functions, assessing correctness in each transformation, while profiting the unified memory through `cudaMallocManaged()` to lessen the code bloat on explicit memory transfer between host and device. The strategy was useful and the professor's fears of groups not being able to code in CUDA quickly vanished. Every group obtained a correct and a performing CUDA code. Students feelings about CUDA where positive, and there was consensus that GPU programming was easier than vector intrinsics programming. From the performance aspect of E5-2620v3 vs. GTX 980, that have comparable amount of transistors, there was a mix of GPU outperforming CPU and vice-versa. The main outcome of the lab was a log-lin graph of normalized performance vs. model size, and here students quickly saw that for GPUs the more data the better, while for CPUs it was the other way round. Through experimentation the students found that CPUs are better suited for latency computing while GPUs are suited for throughput computing.

Three projects were not chosen and we think this record of failure is important to understand why these projects did not work. One of them was `endoh1`. Although it has a fancy visualization and is a very interesting piece of software, nobody picked it in either of the two project-based versions of the course. We suppose that the still complex shape of the deobfuscated code and the use of the `complex` datatype precluded its election. The second was `tiny_manna`, and the reason may be that professors did not have direct experience with the code and there was no fancy visualization, therefore no student was attracted. The third one was `tiny_sph`. The main reason was not code complexity, but code inflexibility since it was quite difficult to increase the number of particles. Besides, the code was full of non-contiguous memory access by using array of pointers as its main data structure, and throughout the course we used simple contiguous memory data structures such as array-of-structures –AoS– or structure-of-arrays –SoA–. The `tiny_sph` code should be completely refactored to use it in upcoming editions of the course.

Final projects for graduate students showed the impact of the course. -ToDo- Parallelizing R, Carde doing sparse matrix recommendation systems, matlab to C to OpenMP code Caro Daza, etc. Revisar los repos.

V. CONCLUSIONS

We based our course in our previous experience from the Thomas Sterling LSU HPCMMM course, and we developed a different flavor specifically tailored to use all available computing power in a single computer, excluding message-passing parallelism in order to be deeper in the topics discussion. Our course differs from traditional HPC courses in Argentina, in that we put the effective use of as many transistors in the die area as possible as a guiding principle.

The course interest increased and diversified throughout its editions. From a dozen of students in 2012, it doubled in 2014 and in 2016 we had an audience of forty students. The main source of students come from previous year's student recommendation, and from people knowing first hand that the instructors had real and successful experience parallelizing codes.

Given the fact that the course is for CS students finishing their undergraduate studies and science graduate students, it represents a remarkable amount for local standards.

Diversity was not a problem but a solution. The CS, non-CS pairing proved useful, no only to match the science FLOPS consumers with the people who know the platforms, but also to raise the respect of the domain knowledge in both fields. CS students understood that it is not just a matter of blind optimization and parallelization, sometimes the key is problem domain knowledge. The other way round the non-CS students

learned a lot of computational concepts and computational filigree that are key to deliver an incredible boost in their codes.

In terms of technologies and parallelism teaching we found the following. Effective use of CPUs is no less difficult than the effective use of GPUs, contrary to students previous conceptions. Explicit DLP is hard to code for but performance results are great, in comparison with directive-based TLP that is simple code decoration and a lot of memory profiling to get a decent scalability afterwards. It is easier to achieve more efficiency with respect to peak performance in GPU than in CPU. For particular problems like BLAS or FFT, there is no other choice but to use native optimized libraries.

One key factor for this highly technological course success was having up-to-date hardware. In early 2016 we had the most powerful single-box computer of Argentine Universities for students to learn parallelism.

Finally we are generating demand for HPC in the scientific community and knowledge in HPC technology in the CS students. The match-making is a long term investment to seed future interdisciplinary work in the HPC area within our University and increase the demand of supercomputing.

ACKNOWLEDGMENTS

We want to acknowledge NSF/TCPP curriculum initiative on Parallel and Distributed Computing for the 2012 Award. NVIDIA for the continuous GPU Education Center and GPU Research Center awards throughout the three editions including lots of cards. Special thanks to Chandra Cheij, the Academic Research Program Manager at NVIDIA. We also like to thank to CCAD-UNC for the Mendieta Cluster to conduct experiments and LANAIS Group FaMAF-UNC for the servers used in 2012, 2014. PROMINF and PAMEG programs are also in the thank list since they allowed our Faculty to buy two high-end servers in 2016. Thanks to the anonymous referees for suggestions and comments. Finally we would like to thank alumni Carlos Budde, Dionisio Alonso and Leandro Perona who not only provided lots of feedback in 2014, but also teaching assistance to the course in 2016.

REFERENCES

[1] C. Bederián and N. Wolovick, "Computación paralela," 2016. [Online]. Available: http://www.cs.famaf.unc.edu.ar/~nicolasw/Docencia/CP/2016

[2] M. Själander, M. Martonosi, and S. Kaxiras, *Power-Efficient Computer Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture, 2014.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 5th ed. Morgan Kaufmann, 2013.

[4] S. Williams, A. Waterman, and D. A. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[5] W. Dally, "Efficiency and programmability: Enablers for exascale," SC13, 2013.

[6] F. Giesen, "SSE: mind the gap!" 2016. [Online]. Available: https://fgiesen.wordpress.com/2016/04/03/sse-mind-the-gap

[7] LocklessInc, "Auto-vectorization with gcc 4.7," 2012. [Online]. Available: http://locklessinc.com/articles/vectorize

[8] Intel, "Intel intrinsics guide," 2016. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide

[9] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, ser. Scientific and Engineering Computation Series. MIT Press, 2008.

[10] V. Volkov, "Better performance at lower occupancy," GTC'10, 2010.

[11] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors - A Hands-on Approach*, 2nd ed. Morgan Kaufmann, 2013.

[12] E. Meinhardt-Llopis, J. S. Pérez, and D. Kondermann, "Horn-Schunck optical flow with a multi-scale strategy," *IPOL Journal*, vol. 3, pp. 151–172, 2013.

[13] B. K. P. Horn and B. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 59, no. 1-2, pp. 81–87, Feb. 1993.

[14] J. Stam, "Real-time fluid dynamics for games," GDC'03, 2003.

[15] J. Atala, C. Bederián, A. Bordese, G. Ingaramo, F. Gaich, J. Medina, M. Rosetti, J. Sánchez, M. Tealdi, and N. Wolovick, "Real-time FullHD tracking-learning-detection on a 2-SMX GPU," GTC'15, 2015.

[16] M. Tealdi, "Paralelización de algoritmos para verificación simbólica de modelos probabilísticos," Master's thesis, FaMAF, Universidad Nacional de Córdoba, 2013.

[17] G. Ingaramo, "Implementación del algoritmo Gauss-Seidel en CUDA para la verificación simbólica de modelos probabilísticos," Master's thesis, FaMAF, Universidad Nacional de Córdoba, 2013.

[18] R. S. Galeote, "Análisis de las arquitecturas gráficas emergentes mediante códigos de matrices dispersas," Master's thesis, Escuela Técnica Superior de Ingeniería Informática, Universidad de Málaga, 2013.

[19] E. E. Ferrero, J. P. D. Francesco, N. Wolovick, and S. A. Cannas, "q-state potts model metastability study using optimized GPU-based Monte Carlo algorithms," *Computer Physics Communications*, vol. 183, no. 8, pp. 1578–1587, 2012.

[20] S. S. Manna, "Critical exponents of the sand pile models in two dimensions," *Physica A*, vol. 179, no. 2, pp. 249–268, 1991.

[21] S. Prahl, "Drop-dead simple Monte Carlo codes," 2016. [Online]. Available: http://omlc.org/software/mc

[22] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Eurographics/SIGGRAPH Symposium on Computer Animation*, D. Breen and M. Lin, Eds., 2003, pp. 154–159.

[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2012.

[24] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007.