Boosting quantum evolutions using Trotter-Suzuki algorithms on GPUs

Carlos S. Bederián[1] and Axel D. Dente[1,2]

[1] Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, Ciudad Universitaria, 5000 Córdoba, Argentina.
[2] Instituto de Física Enrique Gaviola (CONICET - U.N.C.)

Abstract. The evolution calculation of quantum systems represents a great challenge nowadays. Numerical implementations typically scale exponentially with the size of the system, demanding high amounts of resources. General Purpose Graphics Processor Units (GPGPUs) enable a new range of possibilities for numerical simulations of quantum systems. In this work we implemented, optimized and compared the quantum Trotter-Suzuki algorithm running on both CPUs and GPUs.

1 Introduction

In the physics of macroscopic systems, the evolutions are typically ruled by classical equations (Newton's three laws of motion), where the numerical computation scales linearly with the size of the system. However, in the microscopic world, where the evolutions are governed by quantum mechanics (Schrödinger equation), the computation scales exponentially. Thus, the search of algorithms capable to perform numerical evolutions of quantum systems has become a necessity. The exact numerical computation of a quantum evolution on discrete systems implies a digitalization procedure with $2^N \times 2^N$ matrices, where N is the number of sites in the system. From this diagonalization it is observed that the computational time increases exponentially. In order to deal with this problem, the Trotter-Suzuki Algorithm (TSA)[1, 2] implements an approximated method based on coarse-grain evolutions, taking advantage of the exponential operator properties to perform a partition of the dynamics into smaller piece evolutions which can we made independently, allowing parallelization. This algorithm has been previously applied to the study of the dynamical behavior of quantum chaotic systems [3, 4].

A high performance implementation of the Trotter-Suzuki algorithm is highly useful for numerical studies in quantum chaotic systems and time reversal simulations [3, 4]. Additionally, the use of pair partitioning methods in classical oscillators [5] enables the implementation of this algorithm (with few changes) for the study of sound propagation in one and two dimensional systems.

In this work we present a parallelization on CPUs and GPUs of the TSA algorithm used to study the evolution of gaussian wave packets within two dimensional (2D) tight binding systems.

2 Trotter-Suzuki Algorithm

The heart of the TSA is the decomposition of dynamics into sets of small unitary evolutions. The evolution of a quantum state within a time independent hamiltonian $\mathcal{H}$ is written as follows,

$$|\Psi(t)\rangle = \exp\left(-\frac{i}{\hbar}\mathcal{H}t\right)|\Psi(t=0)\rangle\,, \tag{1}$$

where $|\Psi(t=0)\rangle$ is the initial state.
Under separable conditions, the Hamiltonian $\mathcal{H}$ can be written as a sum of $\mathcal{H}_1$ and $\mathcal{H}_2$. Thus, Ec.1 is rewritten as follows,

$$|\Psi(t)\rangle = \exp\left(-\frac{i}{\hbar}\mathcal{H}_1 t\right)\exp\left(-\frac{i}{\hbar}\mathcal{H}_2 t\right)\exp\left(-\frac{i}{\hbar}[\mathcal{H}_1,\mathcal{H}_2]t\right)|\Psi(t=0)\rangle\,, \tag{2}$$

where $[\mathcal{H}_1,\mathcal{H}_2]$ is the commutator operator between $\mathcal{H}_1$ and $\mathcal{H}_2$.

By performing a time evolution with $t << 1$, $\exp\left(-\frac{i}{\hbar}[\mathcal{H}_1,\mathcal{H}_2]t\right)$ is approximated by 1,

$$|\Psi(t)\rangle \simeq \exp\left(-\frac{i}{\hbar}\mathcal{H}_1 t\right)\exp\left(-\frac{i}{\hbar}\mathcal{H}_2 t\right)|\Psi(t=0)\rangle\,. \tag{3}$$

In this way, Ec. 3 shows that the quantum dynamics is separated as a composition of two evolutions.

Taking into account the approximation of Ec.3 and using the appropriate $\mathcal{H}_1$ and $\mathcal{H}_2$, it is possible to construct the total quantum dynamics by splitting it into small sub-system evolutions.

2.1 Trotter Suzuki approximation in tight binding chains

In this section we will show the application of the TSA to a tight binding chain (1D system). In a simplistic way, the Hamiltonian for this type of systems is represented by a set of sites with nearest-neighbor interactions forming a linear chain. To obtain the total dynamics of a gaussian packet in this system we apply a finite number of Δt steps. The smaller the value of Δt, the more accurate the approximation of the quantum dynamics.

The TSA divides the evolution from t to $t + \Delta t$ into two-sites "grains"(see Eq. 3), which can be performed independently. Each of these two-sites evolutions is calculated from the exact solution to the following hamiltonian,

$$\mathcal{H} = \begin{pmatrix} 0 & -V \\ -V & 0 \end{pmatrix}, \tag{4}$$

where V is the coupling element between the two sites.
The solution for this system is found by using Eq. 1,

$$\begin{aligned} |\Psi(t+\Delta t)\rangle_1 &= \cos(V\frac{\Delta t}{\hbar})\,|\Psi(t)\rangle_1 + i\sin(V\frac{\Delta t}{\hbar})\,|\Psi(t)\rangle_2 \\ |\Psi(t+\Delta t)\rangle_2 &= i\sin(V\frac{\Delta t}{\hbar})\,|\Psi(t)\rangle_1 + \cos(V\frac{\Delta t}{\hbar})\,|\Psi(t)\rangle_2 \end{aligned} \tag{5}$$

From Eq. 5 we can observe that the evolution from t to $t+\Delta t$ requires the values of both sites at the previous step, so it is crucial to synchronize all the "grains" at each temporal step.
From now on, we will use this two-sites exact evolution as the building block for solving more complex quantum dynamics.

In Fig 1 we show the procedure for a Δt step in a tight binding chain, where the separation of the problem into two-site evolutions is schematized. It is important to note that one temporal Δt evolution is achieved by performing two computational steps, and after each of these the TSA requires the synchronization of the whole system.

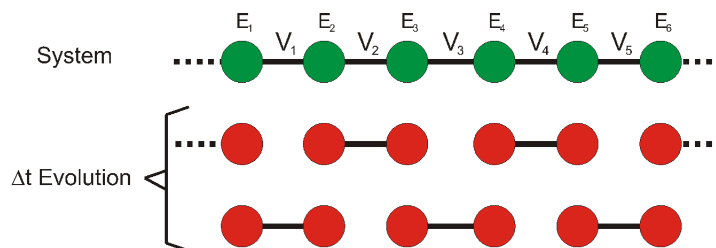

Fig. 1. Scheme for the 1D evolution. The circles represent the sites of the system, and the black lines their couplings. The red circles paired by black lines are two-site evolutions. In this figure it is possible to observe that the evolution of each pair is independent from the other pairs.

2.2 TSA in 2D tight binding

To perform a Δt evolution in a 2D system it is necessary to increase the number of computational steps in order to account for every two-site combination and at the same time avoid synchronization problems. In order to get a better approximation, we have developed code for the second order approximation of the TSA method, which can be found in Ref. [1]. In Fig. 2 we schematized the procedure to perform a Δt step.

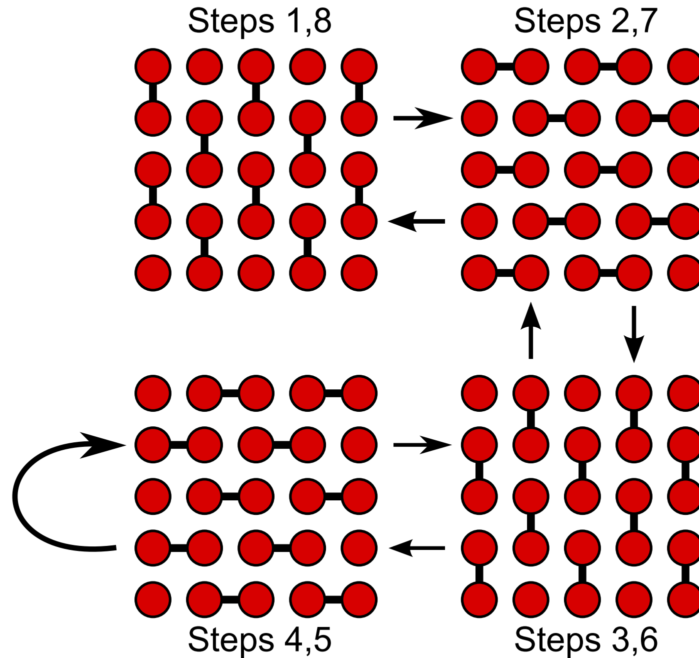

Fig. 2. Scheme for the 2D second order Δt evolution.

3 GPGPU introduction and terminology

GPUs are gaining popularity in the High Performance Computing field as cost-effective high throughput accelerators for massively parallel problems. While CPUs dedicate most of their transistor count to improve sequential code performance, GPUs take a different approach housing hundreds of simple execution units which run one thread each at a time.

The work to be done by each thread is specified by a function called *kernel*. All threads run the same kernel. Execution units share instruction fetch and decode hardware like vector processors do, running bundles of threads called *warps*[3], however each thread has its own instruction pointer to allow divergence on branches. This model is called SIMT (Single Instruction, Multiple Thread) and provides more flexibility than SIMD units, but must be used carefully as threads in a warp whose instruction pointer differs from the one that is currently executing must remain idle, reducing throughput.

Multiprocessors contain several execution units and schedule warps to run on them. Warps are grouped into thread *blocks* that can be synchronized using block-wide barriers and cooperate with other threads in the block through shared resources on the multiprocessor they're running in. These resources include a L1

[3] AMD calls them *wavefronts*.

cache and a low latency, user managed *shared memory* [4]. All multiprocessors in a GPU can access the *global memory* on the board (through the L2 cache), which is also an endpoint for host-GPU communication through the PCI Express bus. GPUs don't have prefetching hardware and their caches are relatively small, so the latency on accesses to global memory is hidden by sending a thread block that's waiting for data to sleep and running another block resident on the same multiprocessor in the meantime.

4 Implementation

It's easy to see that the step formula in Eq. 5 has the following form for each pair of adjacent sites (p, q),

$$\begin{aligned} p' &= a_{(p,q)}p + ib_{(p,q)}q \\ q' &= a_{(p,q)}q + ib_{(p,q)}p \end{aligned} \tag{6}$$

where $a_{(p,q)} = \cos(V_{(p,q)}\frac{\Delta t}{\hbar})$ and $b_{(p,q)} = \sin(V_{(p,q)}\frac{\Delta t}{\hbar})$ are constant and can be precalculated for each pair. All our implementations are based on this formula. We will be using the same a and b on every coupling to avoid polluting the caches with coupling values for easier analysis.

4.1 Naive CPU approach

Our first approach was a naive reference implementation of each computational step in a time step. Each computational step is performed in a single pass over the matrix of sites that updates one pair of sites at a time. As the nodes (p, q) in each pair are adjacent, we can loop over the nodes with the lower index in each pair and add a constant shift value to its index to obtain its peer. The shift value is 1 for steps along the x axis, and the row stride for steps along the y axis. If we colored the nodes that the loop steps on, this algorithm paints a checkerboard pattern on the grid.

Since updates are independent, the outer loop over the rows of the matrix was parallelized using OpenMP, resulting in linear scaling with close to perfect efficiency for small system sizes (see Fig.3). For large system sizes and thread counts, performance plummets once the system no longer fits in the processor caches.

4.2 Naive GPU approach

Our GPGPU efforts began with a straight port of the original CPU code to a CUDA C kernel. We ran one thread per pair of nodes with each thread starting on the nodes along the checkerboard pattern mentioned earlier. Each thread reads the node it's on and its peer, updates their values and writes them back to global memory.

[4] AMD calls it *local data store*.

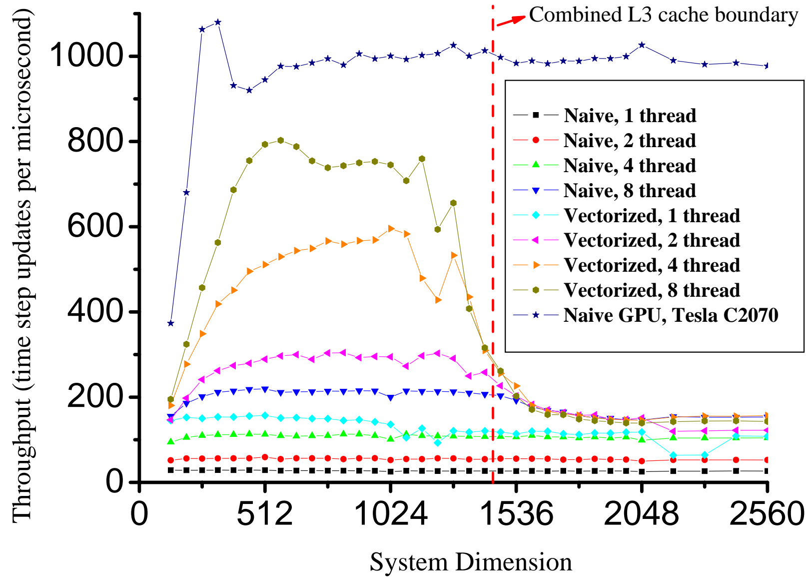

Fig. 3. Early implementation results, single precision.

Originally this code was ran on GeForce GTX 280 boards which are very sensitive to the patterns in which warps access global memory, and as these strided accesses to memory weren't fully *coalesced* [5] this resulted in very modest speedups when compared to CPU code. Fermi-based GPUs read full cache lines and serve execution units from cache like CPUs do, which improves the performance of this code significantly.

Previous studies on GPU stencil computations[6, 7] have been focused on optimizing memory access patterns to avoid excessive refetching on large stencils, but the TSA algorithm only requires one read per element per computational step. However, [6] points out that further improvements can be obtained by performing multiple steps in shared memory, which will be discussed later.

4.3 Vectorized CPU approach

Since the next value of each node is calculated in the same fashion for every node, we set out to improve our initial approach through vectorization[8, 9]. Computation steps that pair sites along the x axis can be processed without

[5] Adjacent and aligned on a 128 bit boundary.

problems using vectors of arbitrary even widths in the following way,

$$\begin{aligned}
\hat{v} &= \langle p_0, q_0, p_1, q_1, ..., p_n, q_n \rangle \\
\hat{t} &= \langle q_0, p_0, q_1, p_1, ..., q_n, p_n \rangle \\
\hat{a} &= \langle a_0, a_0, a_1, a_1, ..., a_n, a_n \rangle \\
\hat{b} &= \langle b_0, b_0, b_1, b_1, ..., b_n, b_n \rangle
\end{aligned}$$

$$\langle p'_0, q'_0, p'_1, q'_1, ..., p'_n, q'_n \rangle = \hat{a} \otimes \hat{v} \oplus \hat{b} \otimes \hat{t}$$

where $\hat{v}$ is read sequentially from each matrix row, $\hat{t}$ can be obtained from $\hat{v}$ through shuffling operations, and $\otimes$ and $\oplus$ are the pointwise product and sum of vectors[6] respectively.

On the other hand, computation steps that pair sites along the y axis need to read two rows simultaneously and update every other value in each row, which is detrimental to memory bandwidth and cache utilization and involves unnecessary amounts of shuffling, as half the data in each vector read needs to be rearranged. This motivated a change in memory layout.

As mentioned earlier, if we apply a checkerboard-like red-black coloring to the nodes on the grid, we can see that computational steps update pairs of sites that are on different colors. The index shift for steps along the y axis remains at one row stride, which is now halved, but on the x axis the shift can be either 0 or 1 depending on whether the row number is even or odd.

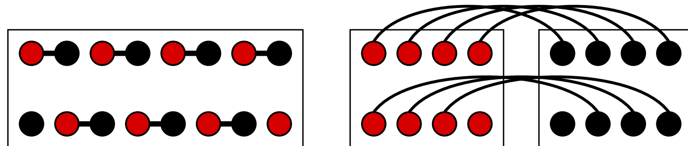

Fig. 4. Red-black split of a horizontal computational step.

With this memory layout we can run computational steps by traversing the lower-indexed color matrix and pairing each of its elements with its peer on the opposite color after applying the appropriate index shift for the current computational step and row. Since the index shift $s \in \{s_{even}, s_{odd}\}$ for a particular step is constant for every element in an even or odd row, a step can be vectorized using

[6] As these are operations on complex numbers, there's an abuse of notation for simplicity. Calculating a complex product on packed data takes extra vector instructions, so real and imaginary data are stored on separate matrices and vectors.

$$\begin{aligned}
\hat{rd} &= \langle rd_0, rd_1, rd_2, ..., rd_n \rangle \\
\hat{bk} &= \langle bk_{0+s}, bk_{1+s}, bk_{2+s}, ..., bk_{n+s} \rangle \\
\hat{a} &= \langle a_0, a_1, a_2, ..., a_n \rangle \\
\hat{b} &= \langle b_0, b_1, b_2, ..., b_n \rangle
\end{aligned}$$

$$\begin{aligned}
\langle rd'_0, rd'_1, rd'_2, ..., rd'_n \rangle &= \hat{a} \otimes \hat{rd} \oplus \hat{b} \otimes \hat{bk} \\
\left\langle bk'_{0+s}, bk'_{1+s}, bk'_{2+s}, ..., bk'_{n+s} \right\rangle &= \hat{a} \otimes \hat{bk} \oplus \hat{b} \otimes \hat{rd}
\end{aligned}$$

This method has no shuffling overhead and, given a matrix with a leading dimension length that is a multiple of 16 bytes, we can use aligned vector reads on one or both matrices depending on the shift required.

The implementation of this approach using SSE intrinsics runs up to 260% faster in single precision mode (Fig.5) and 60% faster in double precision than the reference implementation, but as soon as the system size grows too large for the caches to hold the performance drops to the same level as the regular code. (observe the critical point in Fig.5)

4.4 Tackling the memory bottleneck

While our dual Xeon setup has 16MB split between two 8MB L3 caches that can house up to 1448×1448 single precision systems, Fermi's 768KB L2 cache isn't nearly as spacious. As GPUs rely on massive parallelism to hide memory access latencies and fill their large amounts of execution units, small systems that do fit inside the GPU cache can't spawn enough threads to perform optimally, resulting in the small bump seen on small systems in Fig.3. This placed our early GPU kernels barely ahead of the vectorized CPU code.

The key factor to these performance issues is the very low number of operations per element per computational step, which results in execution unit stalls because the memory subsystem can't keep them fed with a steady stream of data. We can estimate the minimum number of instructions needed per element to avoid stalls on uncached memory accesses by dividing the peak throughput of the processor times twice the size of the elements (one read, one write) by the memory subsystem bandwidth. The estimations for the hardware we tested is shown in Table 1.

The three instructions executed on each element per computational step are far lower than our estimated ideal ratios, so we focused our efforts on increasing this number. The only way to do more work on each element read from memory is to run multiple computational steps on it at a time. Suppose we do two computational steps at a time. Expanding formula 6 for one of the nodes in a pair we have:

$$\begin{aligned}
p'' &= a_{(p,q)} p' + ib_{(p,q)} q' \\
&= a_{(p,q)} (a_{(p,r)} p + ib_{(p,r)} r) + ib_{(p,q)} (a_{(q,s)} q + ib_{(q,s)} s)
\end{aligned}$$

Table 1. Computational resources

System	Memory subsystem	Bandwidth (GBPS)	GFLOPS		Instructions/element	
			float	double	float	double
Dual Xeon X5550[7]	Dual triple-channel DDR3-1066	51.2	170.24	85.12	26.6	26.6
Tesla C2070[8]	384 bit 3132MHz GDDR5	150.3	1030.4	515.2	54.8	54.8
GeForce GTX 580[8]	384 bit 4002MHz GDDR5	192.4	1581.1	197.63	65.8	16.4

where r and s are p's and q's peers on the first step respectively. Although this approach has exponential growth of the number of instructions per element, most of the terms are recalculated several times in one pass through the whole matrix.

We can also split the system into small blocks and evolve each block multiple steps at a time. Work on the block isn't constrained by memory bandwidth as long as blocks fit inside on-chip memory. However, splitting the problem presents new difficulties due to the way in which the stencil changes on each step.

Suppose that a block has been read to on-chip memory and evolved a number of steps. We can't write the results back to the same matrix from where it was read, as blocks adjacent to it still need some values on the boundary between the blocks for their own evolution. This is fixed through double buffering, allocating two matrices instead of one and going back and forth between the two, reading from one and writing on the other.

As we try to perform more computational steps on a block, the amount of nodes external to the block that are required to perform a step on nodes near the boundaries of the block increases. The external nodes are required at varying computational steps, so these nodes also must be updated. These nodes generate a *halo* around a block that must also be read into on-chip memory and updated, but that at later steps become invalid because their own halos aren't present. One of the halo patterns can be seen in Fig. 5.

To minimize the overhead of reading and updating halo values, we must pick a combination of block size and computational steps performed (which determines the minimal halo size) that is optimal. The 24 instructions involved in performing all eight steps in a time step on a block are close to our instruction per element estimation for CPUs, while requiring a manageable 6 halo columns and 8 halo rows per block (which makes slightly taller blocks more efficient in terms of overhead than square blocks), so the block size was determined by taking the largest matrix that fits in on-chip memory and subtracting the halo from it. Duplicating the number of steps to reach a number closer to a GPU's estimated optimal instructions per element would result in a huge halo with prohibitive overhead so we used the same number of steps on both architectures.

The multiple step strategy, combined with the tunable block size that depends on the hardware's cache size, put the algorithm in the family of cache-aware stencil computations[10]. Further flexibility could be obtained by switching the strategy from fixed blocking to cache-oblivious trapezoids[11, 12].

[7] ICC 11.1, openSuSE 11.2 x86-64

[8] NVCC 3.2, Debian 6.0.1

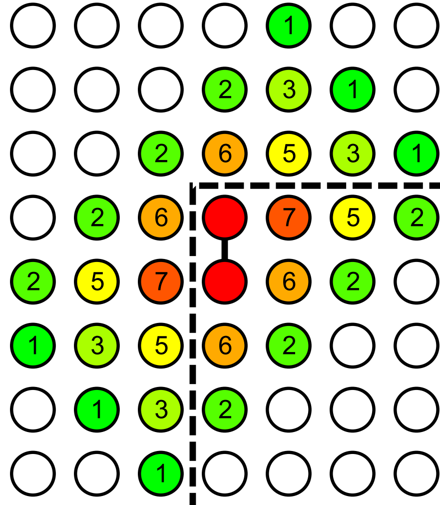

Fig. 5. Halo required by a full step update on a block boundary. Each neighbor is labeled with the last step number in which its value contributes to the update. There are two different halo patterns per computational step, one for each column or row in a 2×2 sub-block.

4.5 Full time step GPU approach

In Fermi hardware, shared memory holds up to 48KB per multiprocessor, or 6144 single precision complex numbers. We chose to run two 32×16 thread blocks on each multiprocessor to allow the GPU to hide the memory latency, limiting our block size to 3072 elements. As said earlier tall blocks have less halo overhead, so we decided on a width of one full GPU warp (32 threads) for SIMT purposes which results in blocks and thread blocks being 96 and 16 elements tall respectively. This puts our overhead at

$$\frac{halo}{block} = \frac{3072 - block}{block} = \frac{3072 - 26 \times 88}{26 \times 88} = 34\%.$$

The kernel reads six 32×16 sub-blocks from global memory into shared memory. The thread block is then arranged in a 32×32 checkerboard pattern similar to the naive implementation, and each thread updates a pair of elements in each third of the block. Finally, the halo is discarded and six 26×16 writes (except for the last sub-block, which is 8 elements tall) are done to global memory, the two matrix pointers are swapped on kernel completion and the process is repeated for the next time step.

Even with the considerable amount of overhead, this implementation runs up to 200% faster (Fig.6) than the original GPU implementation. The double precision results in Fig.7 show our GTX 580 being held back by its lower peak double precision GFLOPS and performing similarly to the memory constrained Tesla C2070 which has less memory bandwidth, confirming that our instructions

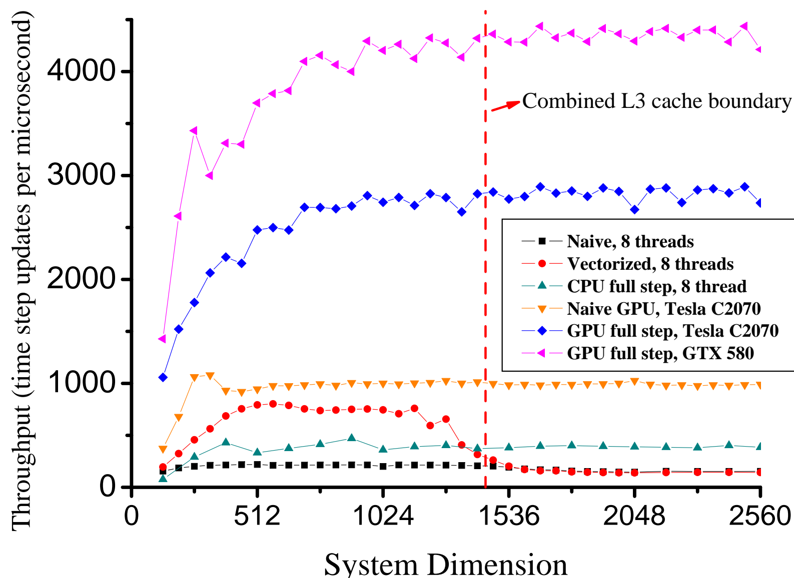

Fig. 6. Full time step results, single precision.

per element ratio theory is sound and that the kernel we've written is well over the actual ratio.

4.6 Full time step CPU approach

CPU caches aren't manageable in software[9], but we can assume that code with strong temporal and spatial locality will be cached properly, so we allocate one static 32KB buffer per core, which is the size of our processor's L1 data cache. Each core then picks a row of blocks and proceeds in similar fashion to the GPU code, copying blocks and their halos to its static buffer, running the computational steps on it with our vectorized code, and writing the block back to the matrices.

To make vectorization easier we widened the halo to 8 columns, making the overhead for tall and wide blocks the same. This makes square blocks optimal, so we decided to use blocks of 64×64 single precision elements, resulting in an overhead of

$$\frac{halo}{block} = \frac{4096 - block}{block} = \frac{4096 - 56 \times 56}{56 \times 56} = 30\%$$

As we can see in Fig. 6 the code still performs up to 15% faster when the problem fits in cache, but increasing the number of computational steps any

[9] Except for prefetch instructions.

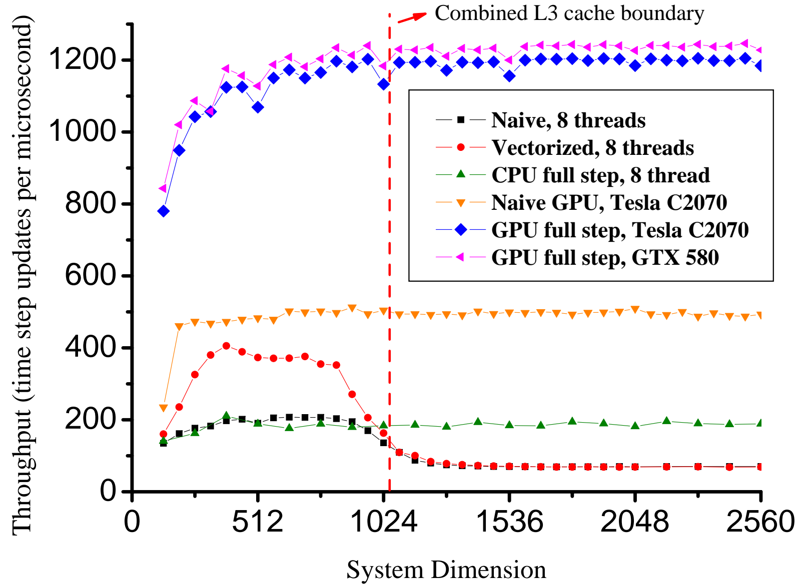

Fig. 7. Full time step results, double precision.

further would also enlarge the halo, reducing efficiency. Comparing the peak throughput obtained when the vectorized code fits in cache with the full time step code's throughput (taking into account its overhead) we can say that there's still room for improvement in CPU performance on large systems for this particular approach, but the 180% speedup over regular bandwidth constrained code paints a better picture of the capabilities of the platform than earlier implementations.

5 Conclusions and future work

In this work we implemented the Trotter-Suzuki algorithm for both CPUs and GPUs in a 2D tight binding system. We studied the speedup obtained through different strategies and compared them with their GPGPU ports, and found that running multiple computational steps at a time is the best option for TSA implementations on both platforms.

In the comparison between CPU and GPU implementations, our testing revealed that the latter have the best performance. The speedups obtained on GPUs against high end CPU setups range from 2.5x when the whole system fits in CPU caches, to 11x for large systems.

Further work can be done on improving the current CPU strategy for lower computation overhead on halo management, or replacing it completely with a cache-oblivious algorithm. Distributing the problem across many GPUs on a

single host or across many hosts on a low latency network should also increase performance, although the tight coupling at boundaries requires careful consideration. As mentioned earlier, the code can also be adapted for sound propagation simulation with minor changes.

Acknowledgements We thank H. L. Calvo, H. M. Pastawski and N. Wolovick for useful discussions. We acknowledge the financial support of CONICET, ANPCyT, SECyT-UNC and NVIDIA.

References

1. H. De Raedt, Ann. Rev. of Comp. Physics IV **107** (1996).
2. H. De Raedt and K. Michielsen, quant-ph/0406210. Handbook of Theoretical and Computational Nanotechnology. Quantum and Molecular Computing, Quantum Simulations (American Scienti.c Publishers, 2006).
3. H.L. Calvo and H.M. Pastawski, EPL **89**, 60002 (2010).
4. H.L. Calvo, R.A. Jalabert, and H.M. Pastawski, Phys. Rev. Lett. **101**, 240403 (2008).
5. H.L. Calvo, E.P. Danieli and H.M. Pastawski, Time reversal mirror and perfect inverse filter in a microscopic model for sound propagation. Physica B **398**, 317 (2007). cond-mat/0702301
6. Paulius Micikevicius, 3D finite difference computation on GPUs using CUDA. Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pages 79–84 (2009).
7. Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf and Katherine Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. Proceedings of the 2008 ACM/IEEE conference on Supercomputing, pages 4:1–4:12 (2008).
8. Samuel Larsen and Saman Amarasinghe, Exploiting superword level parallelism with multimedia instruction sets. Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, pages 145–156 (2000).
9. Alexandre E. Eichenberger, Peng Wu and Kevin O'Brien, Vectorization for SIMD architectures with alignment constraints. Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, pages 82–93 (2004).
10. Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf and Katherine Yelick, Implicit and explicit optimizations for stencil computations. Proceedings of the 2006 workshop on Memory system performance and correctness, pages 51–60 (2006).
11. Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran, Cache-Oblivious Algorithms. 40th Annual Symposium on Foundations of Computer Science (1999).
12. Robert Strzodka, Mohammed Shaheen, Dawid Pajak and Hans-Peter Seidel, Cache oblivious parallelograms in iterative stencil computations. Proceedings of the 24th ACM International Conference on Supercomputing, pages 49-59. ACM (2010).