GPU maps for the space of computation in triangular domain problems

Cristobal A. Navarro
Department of Computer Science
Universidad de Chile
Santiago, Chile
Centro de Estudios Cientifícos (CECs)
Valdivia, Chile
Email: crinavar@dcc.uchile.cl

Nancy Hitschfeld
Department of Computer Science
Universidad de Chile
Santiago, Chile
Email: nancy@dcc.uchile.cl

***Abstract*—There is a stage in the GPU computing pipeline where a grid of thread-blocks, or *space of computation*, is mapped to the problem domain. Normally, the space of computation is a k-dimensional bounding box (BB) that covers a k-dimensional problem. Threads that fall inside the problem domain perform computations and threads that fall outside are discarded, all happening at runtime. For problems with non-square geometry, this approach makes the space of computation larger than what is necessary, wasting many threads. Our case of interest are the two-dimensional triangular domain problems, alias *td-problems*, where almost half of the space of computation is unnecessary when using the BB approach. Problems such as the *Euclidean distance map* or *collision detection* are *td-problems* and they appear frequently as part of a larger computational problem. In this work, we study several mapping functions and their contribution to a better space of computation by reducing the number of unnecessary threads. We compare the performance of four existing mapping strategies; the *bounding box* (BB), the *upper-triangular mapping* (UTM), the *rectangular box* (RB) and the *recursive partition* (REC). In addition, we propose a *map* $g(\lambda)$, that maps any λ block to a unique location (i, j) in the triangular domain. The mapping is based on the properties of the lower triangular matrix and works in *block space*. The theoretical improvement I obtained from using $g(\lambda)$ is upper bounded as $I < 2$ and the number of unnecessary blocks is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. Experimental results using different Nvidia Kepler GPUs show that for computing the Euclidean distance matrix $g(\lambda)$ achieves an improvement of up to 18% over the basic bounding box (BB) strategy, runs faster than UTM and REC strategies and it is almost as fast as RB. Performance results on shared memory 3D collision detection show that $g(\lambda)$ is the fastest map of all, and the only one capable of surpassing the brute force (BB) approach by a margin of up to 7%. These results help us realize that one of the main advantages of $g(\lambda)$ is the fact that it uses *block space mapping*, where coordinate values are small in magnitude and thread organization is not compromised, making the map stable in performance under different memory access patterns.**

I. Introduction

GPU computing is without question a well established research area [14], [11], [2], since the release of general purpose computing platforms such as CUDA [12] and OpenCL [7]. In the CUDA GPU programming model there are three constructs that allow the execution of highly parallel algorithms; (1) thread, (2) block and (3) grid. Threads are the smallest elements and they are in charge of executing the instructions of the GPU kernel. A block is an intermediate structure that contains a set of threads organized in an Euclidean space. Blocks provide fast shared memory access as well as synchronization for all of its threads. The grid is the largest construct of the GPU and it is in charge of keeping all blocks together, spatially organized for the whole execution of the kernel. These three constructs play an important role when mapping the execution resources to the problem domain, and are also necessary for the GPU to schedule and distribute the work properly among its clusters of processing cores. OpenCL chooses different names for these constructs; (1) work-element, (2) work-group and (3) work-space, respectively.

For every GPU application, there is a stage in which the grid, or *space of computation*, is mapped to a problem domain for an eventual computation later on. This map can be defined as a function $f(x) : R^k \rightarrow R^p$ that transforms each k-dimensional point $x = (x_1, x_2, ..., x_k)$ of the grid to a unique p-dimensional point of the problem domain. In other words, $f(x)$ *maps the space of computation to the problem domain*. When the problem domain is simple in shape, rectangular or square grids are good choices because the bounding box perfectly matches the domain. Rectangular or square grids are the most used ones and they are characterized for using the bounding box strategy (BB) map, where $f(x) = x$. Other problems however do not match a box shaped domain because they have a different geometry. For instance, some 2D problems have a triangular shaped domain. We call these type of problems *triangular-domain-problems* or simply *td-problems*. Building a square grid for a *td-problem* is not the best choice because it generates unnecessary thread-blocks that would need to execute if-else conditionals to discard themselves, leading to a performance penalty. The scenario is illustrated in Figure 1, where the unnecessary blocks are painted with a sad red face.

In this work we address the lack of comparisons between different existing mapping strategies for *td-problems*, presenting performance results running the same tests under the same hardware. The idea is to establish a common benchmark for all mapping strategies, using three different GPUs, and measure their performance under three cases: (1) dummy kernel, (2) Euclidean distance matrix and (3) collision detection. We have chosen these three problems because they represent different scenarios; (1) very small kernel, (2) global memory patterns and (3) shared memory patterns. In addition to this benchmark,

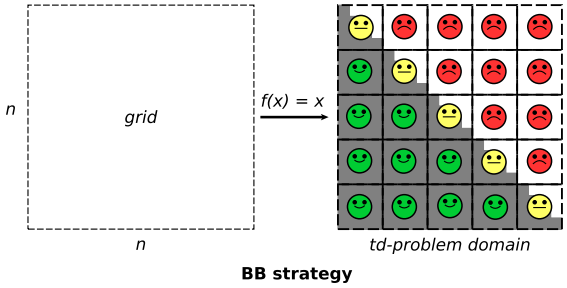

Fig. 1: The BB strategy is not the best choice for a *td-problem*.

we present a new map $g(\lambda)$, that works in *block space* and uses the *lower-triangular matrix* properties. Map $g(\lambda)$ basically makes three contributions; (1) simpler implementation than the other strategies, (2) greater square root range than in UTM [1] and (3) uncompromised thread organization per block.

The rest of the manuscript is organized as follows: Section II presents related work on GPU maps. Section III covers the formulation of $g(\lambda)$. Details about its implementation and how we chose the fastest square root function are in Section IV. In Section V we present the performance results for all existing strategies using three different Nvidia Kepler GPUs; (1) GTX 765M, (2) GTX 680 and (3) Tesla K40. Finally, in Section VI we conclude by discussing the main points of our work.

II. Related Work

The problem of improving the space of computation for *td-problems* is indeed important because any contribution on the matter will eventually, by consequence, benefit every problem of this class. Many computational problems are in fact *td-problems*; Euclidean distance maps (EDM) [10], [8], [9], collision detection [1], graph traversal through adjacency matrices [6], Cellular Automata (e.g John Conway's Game of life [3]) in triangular domains, matrix inversion [16], LU/Cholesky decomposition [4], among others.

In the field of distance maps, Ying *et. al.* have proposed a GPU implementation for parallel computation of DNA sequence distances [17] which is based on EDM. In their work, the authors mention that the problem domain is indeed symmetric and they do realize that only the upper or lower triangular part of the interaction matrix requires computation. Li *et. al.* [8] have also worked on GPU-based EDMs on large data and have also identified the symmetry involved in the computation. However, in both works there is no mention of the mapping of the space of computation.

Jung *et. al.* [5] proposed in 2008 packed data structures for representing triangular and symmetric matrices with applications to LU and Cholesky decomposition [4]. The strategy is based on building a *rectangular box strategy* (RB) for accessing and storing a triangular matrix (upper or lower). Data structures become practically half the size with respect to classical methods based on the full matrix. Originally, this strategy is aimed at the memory structure of the matrix, but it can also be applied analogously to the space of computation.

In 2009, Ries *et. al.* contributed with a parallel GPU method for the triangular matrix inversion [16]. The authors identify that the space of computation indeed can be improved by using a *recursive partition* (REC) of the grid, based on a *divide and conquer* strategy.

In 2012, Q. Avril *et. al.* proposed a GPU mapping function for collision detection based on the properties of the *upper-triangular map* [1]. The map, namely *UTM*, is a function $f(k) \rightarrow (a, b)$, where k is the linear index of a thread t_k and the pair (a, b) is a unique two-dimensional coordinate in the upper triangular matrix. The authors mention that for the square root computation they use Carmack's and Lomont's fast square root approximation (based on the Newton-Raphson approximation algorithm [15]) for speeding up the mapping function. Approximation errors are fixed by using two conditionals statements inside the kernel. Their square root implementation $sqrt(x)$ is accurate for $x \in [0, 100M]$, which according to their map function, would refer to problems in the range $N \in [0, 3000]$.

Up to date, there has not been a dedicated comparison of the different strategies proposed for improving the space of computation in *td-problems*. In the best case we can find a comparison between Avril's map and the BB strategy [1] where the authors report a speedup of almost $2X$, however the filtering is done by thread ID instead of by block coordinates, making the BB map slower.

The triangular map can still be improved to work for problem sizes of $N < 30000$ and also to be able to use the GPU shared memory, through a blockwise formulation.

III. The Blockwise Triangular Map

A. Formulation

It is important first of all to distinguish between two types of mappings that can be performed on the GPU; (1) threadwise mapping and (2) blockwise mapping. Threadwise mapping is where each thread uses its own unique index as the parameter for the mapping function, just as in the UTM map [1]. In blockwise mapping threads use their block index to map all of them to a specific location, followed by a local shift according to the relative position in the block. We have chosen to continue with blockwise mapping and its advantages are explained later on.

Let N be the problem size (assuming data-parallel elements), A a *td-problem* of size $N(N+1)/2$, $n = \lceil N/\rho \rceil$ the number of blocks needed to cover the data along one dimension and ρ the number of threads per block per dimension, or *dimensional block-size* (for simplicity, we assume all dimensions are the same). A BB strategy would simply build a square grid, namely G_{BB}, of nxn blocks and put conditional instructions to cancel the computations outside the problem domain. A finer analysis tells that $n(n+1)/2$ blocks are sufficient to cover the problem domain of A:

$$A = \begin{vmatrix} 0 & & & & \\ 1 & 2 & & & \\ 3 & 4 & 5 & & \\ \dots & \dots & \dots & \dots & \\ \frac{n(n-1)}{2} & \frac{n(n-1)}{2}+1 & \dots & \dots & \frac{n(n+1)}{2}-1 \end{vmatrix} \quad (1)$$

Note : for the rest of the paper we will refer to our mapping approach as **LTM** for **lower triangular mapping**, or simply $g(\lambda)$.

The idea is to build a two-dimensional balanced grid G_{LTM} that covers the lower-triangular matrix, just using $n(n+1)/2$ blocks. By balanced grid, we mean $n' = \lceil \sqrt{n(n+1)/2} \rceil$ blocks per dimension. Figure 2 illustrates G_{LTM} and how it is smaller than G_{BB} from Figure 1, while still providing the necessary number of blocks to cover the problem domain. The

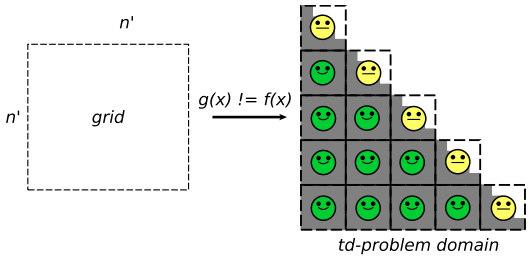

Fig. 2: The LTM strategy uses just the necessary number of blocks to cover the problem domain.

result is a reduction from $n(n-1)/2 \in \mathcal{O}(n^2)$ to $n/2 \in \mathcal{O}(n)$ unnecessary blocks.

The next step is to formulate $g(\lambda) = (i, j)$ where (i, j) are the coordinates in problem space and λ is the index of block $B_{x,y}$ computed as $\lambda = x + yn'$ in *block space*.

Theorem 1: For any block $B_{x,y}$ with $\lambda = x + yn'$, its mapping function $g(\lambda)$ is:

$$g(\lambda) = (i, j) = \left(\left\lfloor \sqrt{\frac{1}{4} + 2\lambda} - \frac{1}{2} \right\rfloor, \lambda - i(i+1)/2 \right) \quad (2)$$

: The λ index can be expressed as a triangular number with an unknown summation limit x.

$$\lambda = \sum_{k=1}^{x} k \quad (3)$$

The index of the far left block that lies on the same row of the λ-th block corresponds to the sum in the range $[1, i]$. Similarly, the index of the far left block of the $(i+1)$-th row is a sum in the range $[1, i+1]$. For all λ indices under the same row i, the range of the summation lies in a range $[1, i+\epsilon]$, where $\epsilon < 1$:

$$\sum_{k=1}^{i} k \leq \lambda = \sum_{k=1}^{i+\epsilon} k < \sum_{k=1}^{i+1} k \quad (4)$$

Therefore $x \in \mathbb{R}$ and $i = \lfloor x \rfloor$. Since $\sum_{k=1}^{x} k = x(x+1)/2$, x is found by just solving a second order equation with coefficients $a = 1$, $b = 1$ and $c = -2$:

$$x^2 + x - 2\lambda = 0 \quad (5)$$

The solution of interest is:

$$x = \frac{\sqrt{1 + 8\lambda} - 1}{2} = \sqrt{1/4 + 2\lambda} - 1/2 \quad (6)$$

And the i coordinate of the λ-th block is computed as:

$$i = \lfloor x \rfloor = \left\lfloor \sqrt{1/4 + 2\lambda} - 1/2 \right\rfloor \quad (7)$$

Coordinate j is the distance from the λ-th block to the left most block in the same row:

$$j = \lambda - \frac{i(i+1)}{2} \quad (8)$$

■

If the diagonal is not needed, then $g(\lambda)$ becomes:

$$g(\lambda) = (i, j) = \left(\left\lfloor \sqrt{\frac{1}{4} + 2\lambda} + \frac{1}{2} \right\rfloor, \lambda - i(i+1)/2 \right) \quad (9)$$

When comparing LTM and its closest counterpart UTM [1], we identify three improvements: (1) $g(\lambda)$ uses fewer floating point operations than in UTM since it uses the lower-triangular mapping. (2) LTM maps blocks and not threads as in UTM. Since $g(\lambda)$ is a map of blocks and the number of blocks is $n = N/\rho$, the square root gives smaller approximation errors, allowing accurate computation for larger values of N. (3) Thread organization is not compromised in LTM, while in UTM it is.

B. Bounds on the improvement factor

The reduction of unnecessary blocks from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ may suggest that the improvement could reach in the best case a $2X$ factor. For this to be possible, one would need to measure just the mapping time, so that necessary and unnecessary blocks do the same jobs, and assume that the mapping function $g(\lambda)$ is as cheap as in the BB strategy. In the following analysis we analyze the improvement factor considering a more realistic scenario where $g(\lambda)$ has a higher cost than the BB map, due to the square root computation involved.

The LTM strategy depends strongly on the square root which in theory is asymptotically $O(M(n))$ [18] where $M(n)$ is the cost of multiplying two numbers of n digits. Considering that real numbers are represented by a finite number of digits (*i.e.,* floating point numbers with a maximum of m digits), then all basic operations cost a fixed amount of time, leading to a constant cost $M(m) = C_s \in \mathcal{O}(1)$. All other computations are elemental arithmetic operations and will be taken as an additional cost of $C_a \in \mathcal{O}(1)$. The LTM strategy has a cost of $\tau = C_s + C_a = \mathcal{O}(1)$ for each mapped thread. On the other hand, the BB strategy checks for each thread if $B_y <= B_x$ in order to continue or not, also leading to a constant cost of $\beta \in \mathcal{O}(1)$.

For this particular case, asymptotic analysis does not give precise information about the improvement factor, since both the LTM and BB strategies lie in the same complexity order (*i.e.,* $n(n+1)/2 \in \mathcal{O}(n^2)$) therefore the improvement should be a constant factor. We proceed to a finer non-asymptotic analysis in order to find the constants involved in the improvement factor.

Let $|G_{BB}|$ and $|G_{LTM}|$ be the number of blocks for the BB and LTM strategies, respectively, and ρ the number of threads per block per dimension as mentioned earlier. It is indeed evident that β is cheaper than τ, therefore $\tau = k\beta$ with a constant $k \geq 1$. The improvement factor I can be obtained by dividing the total cost of BB by LTM across their entire grids:

$$I = \frac{\beta |G_{BB}| \rho^2}{\tau |G_{LTM}| \rho^2} = \frac{\beta n^2}{\tau n(n+1)/2} = \frac{2\beta n^2}{\tau n^2 + \tau n} \quad (10)$$

As shown in (10), the improvement does not depend on the threads, but instead, on the blocks. For large n we have:

$$I = \frac{2\beta n^2}{\tau n^2 + \tau n} \approx \frac{2\beta}{\tau} \tag{11}$$

If $\tau \geq 2\beta$, performance is equal to or worse than that of BB. A real improvement is achieved when:

$$\beta \leq \tau < 2\beta \tag{12}$$

By using the relation $\tau = k\beta$ in (11) we get that:

$$I \approx 2/k \tag{13}$$

Since $k > 1$, the range of I is:

$$0 < I < 2 \tag{14}$$

Any value in the range $1 < I < 2$ means an improvement in performance and a value in the range $0 < I < 1$ will mean a slowdown respect to the BB strategy. Constant k can be interpreted as the cost and overhead of the mapping function. A value of $k \approx 1$ means that the maximum possible improvement is achieved; $I_{max} \approx 2$, under large n. In practice, a value of $k \approx 1$ is too optimistic and would not occur. Our hypothesis is that actual hardware could give a value in the range $1.5 \leq k < 2.0$ which would correspond to $1.00 < I \leq 1.33$. Any value of $k \geq 2$ will lead to no improvement at all, resulting in slower performance than the BB strategy. It is important to put emphasis on the fact that C_a (arithmetic operations) will not have much room for optimization as C_s. Therefore, getting the maximum possible value of I will finally depend on how fast the square root can be.

IV. Implementation of LTM

In this section we choose the best of three square root implementations to be used in the LTM map. We also explain in general terms the idea behind the other mapping strategies we have also implemented for later comparison.

A. Choosing the best square root

The performance of the LTM strategy depends strongly on how fast the computation of index i is. More precisely, the computation of the square root as mentioned earlier. We made three versions of the LTM map, using different square root implementations, and computed the improvement factor with respect to the BB strategy. The first one, named *LTM-X*, uses the default $sqrtf(x)$ function from CUDA and it is the simplest one.

The second implementation of LTM, named *LTM-N*, computes the square root by using three iterations of the Newton-Raphson method [18], [15]. We use the implementation of Carmack and Lomont. This implementation has proved to be effective for applications that allow small errors. The initial value used is the magic number '0x5f3759df' (this initial guess became known when 'Id Software' released Quake 3 source code back in the year 2005). We have found that adding a constant of $\epsilon = 10^{-4}$ to the result of the square root can fix approximation errors in the range $N \in [0, 30720]$.

The third implementation, named *LTM-R*, uses the hardware implemented reciprocal square root, $rsqrtf(x)$:

$$\sqrt{x} = \frac{x}{\sqrt{x}} = x \cdot rsqrtf(x) \tag{15}$$

In terms of simplicity, *LTM-R* is similar to *LTM-X*, the only difference is that it adds $\epsilon = 10^{-4}$ at the end to fix approximation errors, just like in *LTM-N*.

We measured the improvement factor of each implementation with respect to the BB strategy by running a dummy kernel that computes the i, j indices and writes the sum $i + j$ to a constant location in memory. It is necessary to perform at least one memory access otherwise the compiler can optimize the code removing part of the cost of mapping. Figure 3 shows the improvement factor as $I = BB/LTM$ using the three different implementations, running on three different Nvidia Kepler GPUs; GTX 680, GTX 765M and Tesla K40. We have included the BB map for reference, as an horizontal black line at $I = 1$. On the right side of Figure 3 we compare the number of unnecessary blocks between BB and LTM. From the results, we observe that *LTM-X* is slower than BB when running on the GTX 680 and GTX 765M, achieving $I_{680} \approx 0.87$ and $I_{765M} \approx 0.83$, respectively. For the same two GPUs, *LTM-N* achieves an improvement of $I_{680} \approx 1.025$ and $I_{765M} \approx 0.96$ which is practically the performance of BB. On the other hand, *LTM-R* achieves improvements of $I_{680} \approx 1.15$ and $I_{765M} \approx 1.1$. From these results, we observe that using the inverse square root is the best option for the GTX 680 and GTX 765M. For the case of the Tesla K40, we observe that all three implementations achieve an improvement of $I_{K40} \approx 1.08$, allowing to choose *LTM-X* without any performance penalty.

B. Implementing the other strategies

We have implemented the other mapping strategies; BB, RB, REC and UTM following the details provided by the authors [5], [1], [16]. To each implementation we added the following restriction: the map cannot use any memory that grows as a function of N. This means no auxiliary array such as lookup tables are allowed; constants are allowed though. We have put this restriction in order to guarantee that GPU memory is dedicated to the application problem.

For the *bounding box* (BB) strategy, blocks above the diagonal are discarded at runtime, without needing to compute a thread coordinate. This is done by checking if $B_x > B_y$ for every thread. For the rest of the threads, the coordinate is computed. The condition $i > j$ is still performed to discard threads on blocks where $B_x = B_y$. This implementation of BB is faster than computing the thread coordinate first and filtering afterwards.

The *rectangular box* (RB) strategy is based on the work of Jung *et. al.* [5]. This method takes the sub-triangular portion of the threads where $t_x > N/2$, rotates it CCW and places it above the diagonal to form a rectangular grid (see Figure 4). The original work was actually a memory packing technique aimed at the data structures and not a GPU map, however the main idea of the strategy is suitable as a mapping function. We remove the lookup texture used in the original implementation and perform the coordinate mapping arithmetically at runtime. All threads below the diagonal just need to map to $i = t_y - 1$, while j remains the same. Threads in or above the diagonal map to $i = N - t_y - 1$ and $j = N - i - 1$. This mapping is for even values of N. The case of odd N is the same idea but partitioning at $\lfloor N/2 \rfloor$. A remarkable feature of the RB map is that the number of unnecessary blocks is asymptotically $\mathcal{O}(1)$.

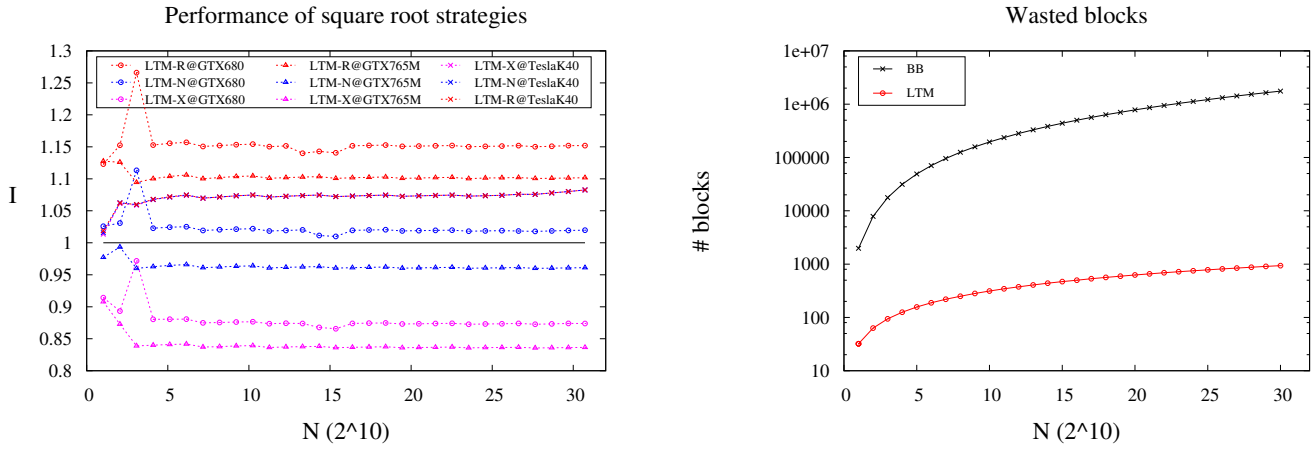

Fig. 3: On the left, the performance of the different square root strategies and on the right the number of unnecessary blocks using $\rho = 16$.

The *recursive partition* (REC) strategy was proposed for the GPU [16] for matrix inversion. In this method, the size of the problem is defined as $N = m2^k$ where k and m are positive integers and m is a multiple of ρ (the block-size). The idea is to do a binary *bottom-up* recursion of k levels, similar to *merge-sort* (see Figure 4). At each level, a grid of blocks is launched for parallel execution, a total of k times. This method requires an additional pass for computing the blocks at the diagonal. More details of how the grid is built and how blocks are distributed are well explained in [16]. In the original work, the mapping of blocks to their respective locations at each level is achieved by using a lookup table stored in constant memory. In this work, we do the mapping at runtime as in RB.

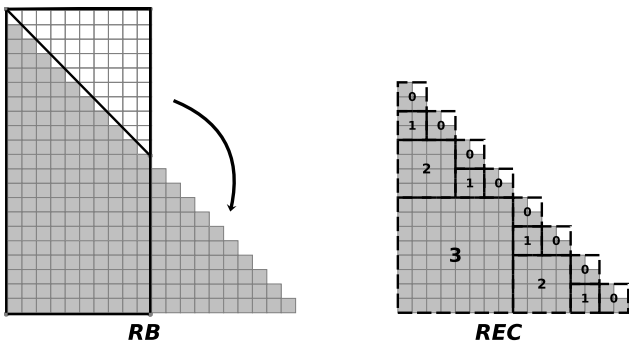

Fig. 4: The RB and REC strategies.

The *upper-triangular mapping* (UTM) was proposed by Avril *et. al.* [1] for performing efficient collision detection on the GPU. This method is very similar to LTM; given a problem size N and a thread index k, its unique pair (a, b) is computed as $a = \lfloor \frac{-(2n+1)+\sqrt{4n^2-4n-8k+1}}{-2} \rfloor$ and $b = (a+1) + k - \frac{(a-1)(2n-a)}{2}$. The UTM strategy uses the idea of mapping to the upper-triangular matrix without the diagonal. Mapping to the upper-triangular matrix still allows solving lower-triangular shaped domains, and *vice-versa* via transposition. An important difference between UTM and LTM is that UTM uses linear thread organization and the map works in *linear thread space*, where very large numbers need to be computed in the square root. On the other hand LTM uses a two-dimensional thread organization and the map works in *linear block space*, making the square root to work with smaller numbers than what UTM uses.

V. Experimental results

Our experimental design consists of measuring the performance of LTM and compare it against all existing strategies; *bounding box* (BB), *upper-triangular mapping* (UTM) [1], *rectangular box* (RB) [5] and the *recursive partition* (REC) [16]. We checked that the outputs for each strategy were always correct. Three tests are performed to each strategy; (1) the dummy kernel, (2) EDM and (3) Collision detection. Test (1) just writes the (i, j) coordinate into a fixed memory location. The purpose of the dummy kernel is to measure just the cost of the strategy and not the cost of the application problem. Test (2) consists of computing the Euclidean distance matrix (EDM) using four features. The purpose is to measure what is the performance of all strategies when solving a global memory based problem. Test (3) consists of performing collision detection of N spheres with random radius inside a unit box. The goal of this last test is to measure the performance of the kernels using a shared memory approach.

The reason why we have chosen these tests is because they are simple enough to study their performance from a GPU map perspective and use different memory access paradigms such as global memory and shared memory. Based on these arguments, we expect that the performance results obtained by the three tests will give insights on what would be the behavior for more complex problems that fall into one of the two memory access paradigms. Furthermore, we have decided to run the tests on three different GPUs in order to obtain metrics general enough that can provide performance patterns. The details of the hardware and the maximum number of simultaneous blocks (*sblocks*) for each GPU is listed in table I. Performance results for the dummy kernel, EDM and collision detection in 1D/3D are presented in Figure 5. On each graphic we plot the performance of all four strategies as different dashed line colors, while the point symbol corresponds to the

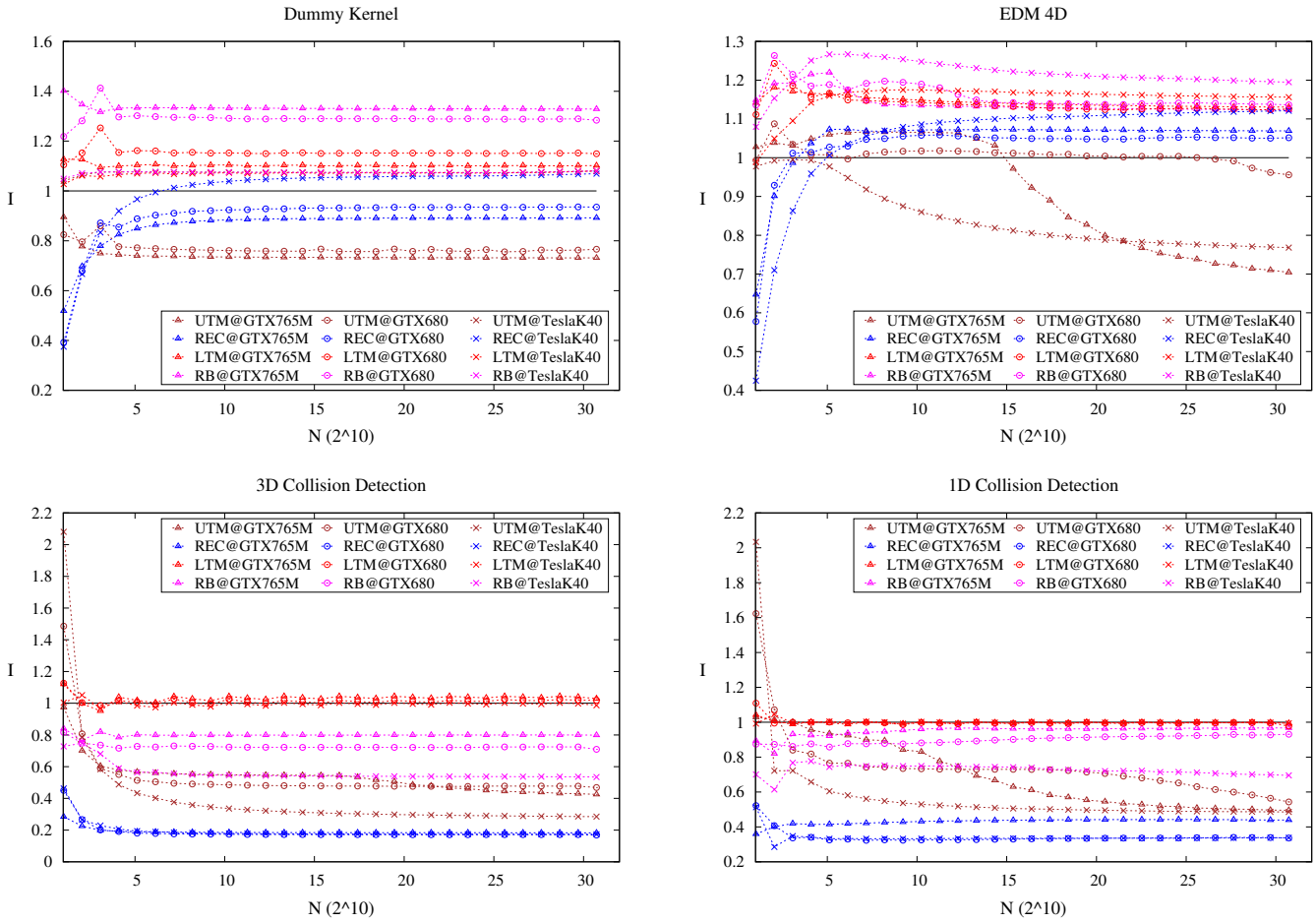

Fig. 5: Improvement factors for the dummy kernel, Euclidean distance matrix and collision detection.

TABLE I: Hardware used for experiments.

Component	Description
CPU	Intel(R) Core(TM) i7-3770K @ 3.50GHz
RAM	32GB DDR3 1333MHZ
GPU_1	Geforce GTX 680 (GK104, 2GB, 1536 cores, sblocks = 128)
GPU_2	Geforce GTX 765M (GK106, 2GB, 768 cores, sblocks = 80)
GPU_3	Tesla K40 (GK110, 12GB, 2880 cores, , sblocks = 240)
API	CUDA 5.5

different GPU chosen for that test. The performance of each mapping strategy is given in terms of its improvement factor I with respect to the BB strategy (*i.e.,* the black and solid horizontal line fixed at $I = 1$). Values that are located above the horizontal line represent actual improvement, while curves that fall below the horizontal line represent a slowdown with respect to the BB strategy. For the dummy kernel test we observe that the RB strategy is the fastest one achieving up to 33% of improvement with respect to BB when running on the GTX 765M. LTM comes in the second place achieving a stable improvement of 18% when running on the GTX 680. The REC and UTM strategies performed slower than BB for the whole range of N. We note that this test running on the Tesla K40 does not provide any clear performance difference between the mapping strategies once $N > 15000$, since they all converge to a 7% of improvement with respect to BB.

For the EDM test, we observe that RB is again the fastest map achieving an improvement of up to 28% with respect to BB when running on the Tesla K40 GPU. In second place comes LTM with a stable improvement of 18% and third the REC map with an improvement that reaches up to 12% for the largest N. The performance of the UTM strategy is lower than BB and unstable for all GPUs. At this point, we have to consider that UTM was designed to work in the range $N \in [0, 3000]$, where it actually does perform better than BB offering up to 4% of improvement over BB. For this test, performance differences did manifest for all GPUs, even on the Tesla K40.

For the collision detection test we have included two cases; 3D and 1D collision detection. From all the map strategies, only LTM manages to perform better than BB, offering an improvement of up to 7%. Indeed, the performance scenario changes drastically in the presence of a different memory access pattern such as shared memory; the RB strategy, which was the best in global memory, now performs slower than BB. The case is similar with the REC map which now performs much slower. It is important to mention that the UTM map is the only strategy that cannot use a 2D shared memory pattern

because the mapping works in *linear thread space*. At low N, UTM achieves a 100% of improvement because of the different memory approach used. However as N grows, its performance is over passed by the rest of the strategies that use shared memory. For the case of 1D collision detection, results are not so beneficial for the mapping strategies and in the case of LTM performance is in the limit of not becoming an improvement. The 1D results show that in low dimensions the benefits of a mapping strategy can be not as good as in high dimensions.

VI. Conclusions

We have studied the benefits of GPU maps for triangular domain problems, alias *td-problems*. These maps allow the use of a smaller space of computation compared to the bounding box strategy (BB) in order to solve a problem. By using a smaller space of computation, the number of unnecessary threads is reduced and the GPU kernel can execute faster. Our proposed GPU map, namely LTM for *lower triangular map*, uses a $g(\lambda)$ function to map the space of computation to any *td-problem*. The main advantage of LTM is that it works in *block space*, not affecting thread organization and allowing problem sizes to be at least in the range of $N \in [0, 30720]$, which is ten times the range achieved in the upper triangular map (UTM) [1]. It is important to mention that the problem size must be large enough to generate more blocks than the number of blocks the GPU can handle in parallel. Assuming a warp of 32 threads, a value of $N \geq 800$ already generates more than double the number of blocks in parallel for any of the GPUs we used. For smaller values of N, the BB strategy will still remain useful.

When comparing all mapping strategies under the same performance tests, we found that the rectangular box (RB) and LTM are the best mapping strategies for a global memory based problem such as EDM, achieving 28% and 18% of improvement, respectively. The recursive partition (REC) strategy managed to provide up to 12% of improvement for the global memory problem. The UTM strategy achieved less performance than the BB method and could not provide exact results when $N > 3000$. For shared memory collision detection we found that no strategy but LTM managed to perform better than BB, by 7%. The reason why LTM could still provide an improvement under a different memory access pattern is because LTM maps in *block space* thus it does not compromise thread organization. The benefits of *block space mapping* are better manifested under shared memory algorithms or thread coarsening techniques [13].

The implementation of the LTM strategy is extremely short in code and totally detached from the problem, making it easy to adopt. When choosing the best implementation of square root, we found that the reciprocal square root was the most convenient option for GK104 and GK106 GPUs such as the GTX 680 and GTX 765M, respectively. When running the LTM map on Nvidia's Tesla K40 GPU, which is a GK110 architecture, we found that the computation of the square root can be performed just using the default function $sqrtf(x)$ without any performance penalty. This makes the implementation of LTM even simpler.

The performance of mapping strategies for *td-problems* still varies depending on the GPU used and on the way memory is accessed, making it hard to choose the best map. Nevertheless, we have found some important performance patterns among our results; the RB and LTM maps are good for global memory problems while the LTM map is the only one good for a shared memory based problem. This scenario puts LTM in advantage over the other maps, since it is the only strategy that can work for both global and shared memory based problems.

Acknowledgment

The authors would like to thank *CONICYT* for funding the PhD program of Cristóbal A. Navarro. This work was partially supported by the FONDECYT project No 1120495.

References

[1] Q. Avril, V. Gouranton, and B. Arnaldi. Fast collision culling in large-scale environments using gpu mapping function. In *EGPGV*, pages 71–80, 2012.

[2] L. M. Cristobal A. Navarro, Nancy Hitschfeld-Kahler. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Commun. Comput. Phys.*, 15:285–329, 2014.

[3] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, Oct. 1970.

[4] F. Gustavson. New generalized data structures for matrices lead to a variety of high performance algorithms. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 2328 of *Lecture Notes in Computer Science*, pages 418–436. Springer Berlin / Heidelberg, 2006.

[5] J. H. Jung and D. P. O'Leary. Exploiting structure of symmetric or triangular matrices on a gpu. Technical report, 2008.

[6] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Software, Environments, Tools. Society for Industrial and Applied Mathematics, 2011.

[7] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[8] Q. Li, V. Kecman, and R. Salman. A chunking method for euclidean distance matrix calculation on large dataset using multi-gpu. In *Proceedings of the 2010 Ninth International Conference on Machine Learning and Applications*, ICMLA '10, pages 208–213, Washington, DC, USA, 2010. IEEE Computer Society.

[9] D. Man, K. Uda, Y. Ito, and K. Nakano. A gpu implementation of computing euclidean distance map with efficient memory access. In *Proceedings of the 2011 Second International Conference on Networking and Computing*, ICNC '11, pages 68–76, Washington, DC, USA, 2011. IEEE Computer Society.

[10] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano. Implementations of parallel computation of euclidean distance map in multicore processors and gpus. In *Networking and Computing (ICNC), 2010 First International Conference on*, pages 120–127, 2010.

[11] J. Nickolls and W. J. Dally. The gpu computing era. *IEEE Micro*, 30(2):56–69, Mar. 2010.

[12] Nvidia-Corporation. *Nvidia CUDA C Programming Guide*, 2012.

[13] L. Nyland, M. Harris, and J. Prins. Fast N-Body Simulation with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, Aug. 2007.

[14] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[15] H. A. Peelle. To teach Newton's square root algorithm. *SIGAPL APL Quote Quad*, 5(4):48–50, Dec. 1974.

[16] F. Ries, T. De Marco, M. Zivieri, and R. Guerrieri. Triangular matrix inversion on graphics processing unit. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 9:1–9:10, New York, NY, USA, 2009. ACM.

[17] Z. Ying, X. Lin, S. C.-W. See, and M. Li. Gpu-accelerated dna distance matrix computation. In *Proceedings of the 2011 Sixth Annual ChinaGrid Conference*, CHINAGRID '11, pages 42–47, Washington, DC, USA, 2011. IEEE Computer Society.

[18] T. J. Ypma. Historical development of the Newton-Raphson method. *SIAM Rev.*, 37(4):531–551, Dec. 1995.