SIGRAD 2014
M. Obaid, D. Sjölie, E. Sintorn and M. Fjeld (Editors)

Accelerated Computation of Minimum Enclosing Balls by GPU Parallelization and Distance Filtering

Linus Källberg & Thomas Larsson

Mälardalen University, Sweden

Abstract

Minimum enclosing balls are used extensively to speed up multidimensional data processing in, e.g., machine learning, spatial databases, and computer graphics. We present a case study of several acceleration techniques that are applicable in enclosing ball algorithms based on repeated farthest-point queries. Parallel GPU solutions using CUDA are developed for both low- and high-dimensional cases. Furthermore, two different distance filtering heuristics are proposed aiming at reducing the cost of the farthest-point queries as much as possible by exploiting lower and upper distance bounds. Empirical tests show encouraging results. Compared to a sequential CPU version of the algorithm, the GPU parallelization runs up to 11 times faster. When applying the distance filtering techniques, further speedups are observed.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming, I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

1. Introduction

The minimum enclosing ball has found widespread use in many different areas of computer science. Depending on the specific context, the dimensionality of the balls varies broadly. Low-dimensional cases are common in computer graphics, computer-aided design, and geographic information systems. Higher-dimensional cases arise frequently in multidimensional index methods and classification based on support vector machines. In these cases, data sets with dimensions in the range 10 to 10000 are common [TKK07].

In the plane, the minimum enclosing circle problem is about fitting a circle to embrace a set of points as tightly as possible. This is a mathematical optimization problem, where the task is to find the unique center such that the maximum distance from the center to any input point is minimized. More generally, in any dimension d, the minimum enclosing ball (MEB) is sought, and the solution is usually denoted $B^* = (c^*, r^*)$, where c^* is the unique center that gives the smallest radius r^*. The MEB problem bears many names, e.g., the smallest bounding sphere or 1-center problem, and the literature is vast [PS85].

Due to the high computational complexity of finding the exact MEB, a number of approximation algorithms have been proposed in the literature [BC03, KMY03, Yıl08]. In a recent publication, Larsson and Källberg [LK13] proposed an efficient algorithm to compute a $(1+\varepsilon)$-approximation of B^* in $O(\frac{dn}{\varepsilon} + \frac{d}{\varepsilon^3})$ time for n input points in dimension d. This algorithm, named FastApxBall, computes a sequence of approximations $B_i = (c_i, r_i)$ such that $r_i \leq r^*$, until a solution is reached whose radius can be enlarged to cover the entire point set without exceeding $(1+\varepsilon)r^*$. Pseudocode is shown in Figure 1. On Lines 1–3, the current approximation is initialized. On Line 4, a subset C of input points, known as a core-set, is initialized. Then in each iteration of the loop beginning on Line 5, the point $q \in P$ located farthest from the current center point is found and inserted into the core-set. The point q as well as $h = \|q - c\|$, where $\|\cdot\|$ denotes the Euclidean norm, are also used on Lines 8 and 9 to update the current solution in each iteration.

The purpose of the subroutine SolveApxBall, invoked on Line 11, is to further refine the current solution by considering only the (at most $2/\varepsilon$) points currently in the core-set (see [LK13] for more details). In practice, this reduces the

FastApxBall(P,ε)
input: $P = \{p_1, p_2, ..., p_n\}$, $\varepsilon > 0$
output: A $(1+\varepsilon)$-approximation B of B^*
1. $q', h' \leftarrow$ FindFarthestPoint(p_1, P)
2. $q, h \leftarrow$ FindFarthestPoint(q', P)
3. $(c, r) \leftarrow ((q'+q)/2, h/2)$
4. $C \leftarrow \{q', q\}$
5. **loop** $2/\varepsilon$ **times** (at most)
6. $q, h \leftarrow$ FindFarthestPoint(c, P)
7. **if** $h \leq r(1+\varepsilon)$ **then exit loop**
8. $r \leftarrow (\frac{r^2}{h} + h)/2$
9. $c \leftarrow q + \frac{r}{h}(c-q)$
10. $C \leftarrow C \cup \{q\}$
11. $(c, r) \leftarrow$ SolveApxBall($(c, r), C, \varepsilon/2$)
12. **return** (c, h)

Figure 1: *The algorithm* FastApxBall.

number of passes considerably, especially in low to moderate dimensions with $n \gg d$. As noted by Larsson and Källberg, it is possible to remove Line 11 from the code to get a modified version of the algorithm, called SimpleApxBall, with time complexity $O(\frac{dn}{\varepsilon})$. Interestingly, it has been shown [KL] that this algorithm is equivalent to Yıldırım's first algorithm [Yıl08].

Since FindFarthestPoint takes $O(dn)$ time in each of the $O(\frac{1}{\varepsilon})$ iterations, the total time spent in this subroutine is $O(\frac{dn}{\varepsilon})$, as captured by the left term in the time complexity of FastApxBall and the remaining term in the time complexity of SimpleApxBall. In practice, these repeated farthest-point queries constitute the main bottleneck in both algorithms. Fortunately, however, due to its high degree of data parallelism, the farthest-point query makes a good candidate for acceleration using parallel computing.

In this paper, GPU parallelization strategies are proposed as well as two algorithmic distance filtering approaches, which are applicable in both serial and parallel settings to reduce the cost of the dominating distance computations. The proposed techniques should be applicable in several other MEB algorithms that perform repeated farthest-point queries, such as [Gär99, BC03, KMY03, Yıl08].

2. Farthest-point queries on the GPU

We let the input points and the center point be represented on the GPU as the following $n \times d$ matrix and d-ary column vector:

$$P = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,d} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,d} \end{bmatrix}, \quad c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_d \end{bmatrix},$$

where $p_{j,k}$ and c_k denote the k-th coordinate of p_j and c, respectively. To realize the FindFarthestPoint subroutine efficiently, a straightforward solution is to first compute all of the n squared distances $\|p_j - c\|^2$ in parallel into a vector e, and then find the maximum of e in a separate parallel reduction step. Thus, e is given by

$$e = \begin{bmatrix} (p_{1,1}-c_1)^2 & + & \cdots & + & (p_{1,d}-c_d)^2 \\ (p_{2,1}-c_1)^2 & + & \cdots & + & (p_{2,d}-c_d)^2 \\ \vdots & & \ddots & & \vdots \\ (p_{n,1}-c_1)^2 & + & \cdots & + & (p_{n,d}-c_d)^2 \end{bmatrix}.$$

Note that computing squared as opposed to exact distances is preferable since it avoids taking n square roots. The computation of e can be further simplified by first rewriting each element as

$$e_j = \|p_j - c\|^2 = \|p_j\|^2 + \|c\|^2 - 2\langle p_j, c\rangle,$$

where $\langle \cdot, \cdot \rangle$ denotes the Euclidean inner product. Since the term $\|c\|^2$ is the same in all distances during a pass, it can be disregarded without altering which point is returned as the farthest from the reduction step. Furthermore, the term $\|p_j\|^2$ remains constant throughout all passes of the MEB algorithm; thus, it can be precomputed for each input point into an n-ary vector u, which is then reused in every pass. Using these simplifications, the distance computation step amounts to a GEMV (general matrix-vector multiplication) kernel:

$$e = u - 2Pc.$$

Efficient GPU implementations of GEMV are commonly found in GPU implementations of BLAS (Basic Linear Algebra Subprograms), such as cuBLAS and MAGMA [DDG*]. Similarly, there are efficient GPU implementations of the reduction step available, e.g., from the Thrust template library for CUDA [BH12].

2.1. Tailor-made kernels

The above solution based on GEMV can be expected to work well in many situations. However, to support the distance filtering techniques that will be described in Section 3, we implemented two tailor-made kernels in CUDA that allow for masking out certain rows of the matrix P in the distance computations. To retain some degree of data locality in this case, we let P be stored in row-major order. This way, any point can be loaded from the global memory in a coalesced fashion into the SMs, provided the dimension is large enough and the matrix is allocated with a properly set pitch.

In the first kernel, a tile mesh is superimposed on the matrix such that each tile covers $t_x \times t_y$ elements. Each thread block is mapped to a number of such tiles, determined by the parameters w_x and w_y for the x and y directions, respectively. Thus, in general, a thread block processes

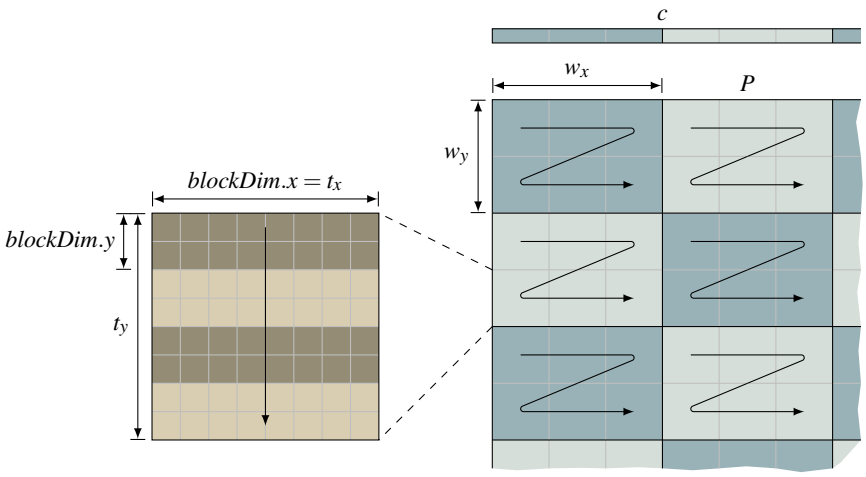

Figure 2: *Illustration of the assignment of matrix tiles to thread blocks and elements within the tiles to threads. In this example, each block processes $w_x = 3$ tiles in the horizontal direction and $w_y = 2$ tiles in the vertical direction. Since $t_x = t_y = 8$ and $blockDim.y = 2$, each thread must iterate four times for a full tile to be processed.*

$(w_x \times w_y) \times (t_x \times t_y)$ elements of the matrix and $w_x \times t_x$ elements of the center point in total in each invocation, although some blocks may do less work depending on the matrix dimensions. Furthermore, depending on the *blockDim* settings, the thread blocks might need to iterate within each tile to process all of its elements. To keep the parameter space manageable, we let the tiles be squares with side t_d, i.e., $t_x = t_y = t_d$, and we let $blockDim.x$ be fixed to t_d. This leaves the parameters t_d, $blockDim.y$, w_x, and w_y that can be varied to tune the performance of the kernel. More details on the used tuning process are given in Section 4.

As illustrated in Figure 2, the thread blocks process their assigned tiles in a left-to-right, top-to-bottom order. Within each tile, the elements are processed from top to bottom. With these traversal orders, each thread can maintain $t_y/blockDim.y$ partial sums of its computed squared differences by storing them in registers. After the last tile in the current row has been processed, all threads in the block store their partial sums into a matrix of size $t_y \times t_x$ in shared memory. Each row of this matrix is then summed up, which reduces it to a column vector of t_y partial sums. Then, before moving on to the next row of tiles, these values are added to the output vector e, which was initialized to 0 before invoking the kernel. Note that in cases when there are more than one thread block working on the same rows of P, i.e., when $w_x \times t_x < d$, the latter addition to e must be done using an atomic add operation, since another thread block might attempt to update the same elements of e simultaneously.

Otherwise, when there is only one block in the horizontal dimension of the grid of thread blocks, a regular add operation can be used instead. However, a more significant optimization is possible in this case: since each distance is computed by a single thread, the reduction needed to find the largest distance can be integrated with the distance computation kernel. This fuses two kernel calls (first computing the distances, then finding the maximum) into one. Moreover, it reduces the amount of data that needs to be written out to global memory, since it becomes unnecessary to store all the distances to the vector e. Thus, we made the kernel detect this special case, so that a local reduction is performed by each thread block to find the maximum of its computed distances. Then, before returning, an inter-block reduction is performed in a scalar in global memory using atomic operations.

Whenever the matrix dimensions are not multiples of the tile dimensions, the last row and/or column in the grid of thread blocks will have tiles to process that are not completely filled. This is handled as a special case in the kernel, so that the overhead from the necessary conditional statements is isolated to the affected thread blocks. Since there are not enough elements in these partially filled tiles to occupy all threads in these blocks, parts of the thread blocks become inactive. As long as both n and d are large enough to give mostly filled tiles overall, this reduction in thread utilization is amortized sufficiently over the processing of the filled tiles. However, whenever either n or d is very small, measures must be taken to avoid poor performance due to low utilization. Since we focus mainly on cases where $d \ll n$ here, we consider only cases where d is small, i.e., when P has a tall shape. Our second kernel is specifically optimized for such matrices. When executing this kernel, we relax the requirement on square tiles and fix only $t_x = d$. As before, we set $blockDim.x = t_x$, which in this case automatically implies $w_x = 1$. Thus, the remaining tunable launch parameters for this kernel are t_y, $blockDim.y$, and w_y. Note that there

may be more than one warp working on each row of the matrix in this case, when d is neither a multiple nor a divisor of the warp size 32. For example, in $d = 3$, each warp spans $\lfloor \frac{32}{3} \rfloor = 10$ full rows as well as parts of one or two other rows. This keeps all threads of the block fully occupied, and due to the row-major organization of P, it also increases the locality of the data accessed by each warp.

3. Distance filtering

Evaluating the distance metric for pairs of points is an expensive operation, particularly in high dimensions. Therefore, attempting to reduce the number of exact distance computations by utilizing approximate distance measures may speed up the processing substantially. By exploiting knowledge from already computed similar distances together with the triangle inequality, safe bounds can be derived to give simple conditions for when an exact distance computation can be skipped.

Of course, using the triangle inequality to filter out distance computations is not a new idea. Such approaches are well-known in data mining where similarity queries are common (see, e.g., [BK73, BS98, BEKS01]). Hjaltason and Samet describe several general rules which can be applied to prune the search space under varying circumstances depending on what distances are known and what distance calculations are attempted to be avoided [HS03].

The filtering techniques we describe below are designed for the specific case of MEB computation. Intuitively, this is a situation where this type of filtering can be expected to pay off very well. The sequence of generated center points sweeps out a path in the neighborhood of the optimal center c^*, and overall the distance between each pair of consecutive center points tends to decrease with each pass. Thus, searching for the farthest point from c_{i+1} is expected to be very similar to searching for the farthest point from c_i. To study the benefit of reducing the number of full distance computations between points during the farthest-point queries, we describe three different filtering schemes for caching and reusing computed distances between the passes of the MEB algorithm. To make the discussion more general, we will use the notation $D(\cdot,\cdot)$ to represent any distance measure that satisfies the triangle inequality.

3.1. Triangle inequality filtering

During the first farthest-point query of the MEB algorithm, the distances from the first center point to all input points are computed and stored in a simple auxiliary array of size $O(n)$, hereafter called the distance array. Then in all the subsequent passes, the distances cached in this array are used to derive an upper bound on the distance to each point in P from the current center point. Before a pass is started, say pass $i+m$, the distances from c_{i+m} to all previous center points are computed. Also, a lower bound f on the actual

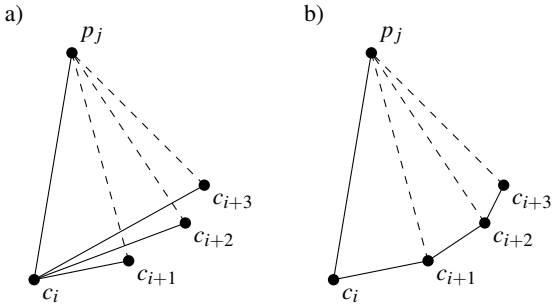

Figure 3: *Upper bounds on the unknown distances, shown as dashed lines, are derived from the known distances, shown as solid lines. a) The exact distance from c_i to the current center point is recomputed in each pass. b) The accumulated movement from c_i is used instead.*

farthest distance $\max(D(c_{i+m}, p \in P))$ is needed. We simply let $f = r_{i+m}$, where r_{i+m} is the current radius. Assuming the distance from an earlier center point c_i to a point p_j is currently stored in position j of the distance array, the following condition can be tested before evaluating the exact distance $D(c_{i+m}, p_j)$:

$$D(c_i, p_j) + D(c_i, c_{i+m}) < f.$$

When this is fulfilled, it is impossible that $D(c_{i+m}, p_j) \geq f$ holds, and the computation of the exact distance can be skipped. This is an immediate consequence of the triangle inequality, which states that

$$D(c_i, p_j) + D(c_i, c_{i+m}) \geq D(c_{i+m}, p_j).$$

This type of filtering is illustrated in Figure 3a., where the distance to a point p_j is computed in pass i, and then the triangle inequality gives approximate distance measures to p_j in the next three passes. Given that the filtering condition succeeds, the unknown distances shown with dashed lines are never computed.

Whenever an actual distance has to be computed for p_j, it is also cached in the distance array simply by overwriting the previously cached value. Note that each stored distance is associated with a certain center point c_i, which must be identified during filtering. Therefore, the value of i is cached as well together with each computed distance. Furthermore, if a computed distance is the largest encountered so far, the point is stored as the current farthest point. When all points have been processed this leaves us with the true farthest point, as well as a distance array that has been updated for all points where the filtering failed.

Two possible drawbacks of this technique are that the distances between all possible pairs of center points are computed, and that the complete sequence of computed center points must be stored. In the algorithms considered here, this filtering method thus requires $O(d/\varepsilon^2)$ time and $O(d/\varepsilon)$

storage. Depending on which MEB algorithm is used, this may degrade the theoretical time complexity of the algorithm. Nevertheless, for sufficiently large point sets and reasonable ε values, this overhead is expected to have little effect on the run time in practice, even in cases when the filtering effectiveness is low.

3.2. Filtering with accumulated distances

An alternative filtering method can be designed, which avoids the above-mentioned potential problems of the previous approach by using a less tight upper bound. In this approach, the distance array stores upper bounds on the distances to earlier center points as opposed to the exact distances. As before, the distance array is initialized with the exact distances from the first query point to all input points. Then in each pass, all these cached distances are incremented with the movement of the center point from the previous pass. Thus, assuming c_i is the last center point from which the distance to a point p_j was computed, the following filtering condition is tested in pass $i+m$:

$$D(c_i,p_j)+D(c_i,c_{i+1})+D(c_{i+1},c_{i+2})+\ldots \\ \ldots+D(c_{i+m-1},c_{i+m})<f,$$

where the sum in the left-hand side represents the value stored for point p_j. When the condition fails, the exact distance is computed and the accumulated bound in the distance array is reset. In Figure 3b, this type of filtering is illustrated. The distance from c_i to p_j is computed in pass i, and then during the following three passes, the consecutive movements of the center points are accumulated in the distance array and used to find approximate distance measures to p_j. Again, given that the filtering condition is fulfilled, the unknown distances to p_j in the next three passes are never computed.

Clearly, the accumulation tends to make the approximate distances less tight compared to the those achieved by the first approach described in the previous section. But similarly to the previous approach, they should improve gradually as the movement of the center point decreases.

Of course, the main advantage of this approach is that it avoids the $O(d/\varepsilon^2)$ computation cost and $O(d/\varepsilon)$ storage cost introduced by the first method above. Thus, incorporating this acceleration technique in farthest point-based MEB algorithms will not affect their theoretical asymptotic time complexity. Of course, storing the auxiliary distance array still introduces an $O(n)$ storage cost.

3.3. Cauchy-Schwarz inequality filtering

Nielsen and Nock [NN04] proposed a filtering criterion based on an upper bound on the Euclidean distance, derived as

$$\begin{aligned}\|p_j-c_i\| &= \sqrt{\|p_j\|^2+\|c_i\|^2-2\langle p_j,c_i\rangle} \\ &\leq \sqrt{\|p_j\|^2+\|c_i\|^2+2\|p_j\|\|c_i\|}.\end{aligned} \tag{1}$$

The bound follows from the Cauchy-Schwarz inequality, which states that for two vectors u and v, it holds that $|\langle u,v\rangle|\leq\|u\|\|v\|$. By computing $\|p\|$ for each $p\in P$ at the beginning of the algorithm, and recomputing $\|c_i\|$ before each farthest-point query, the upper bound in Equation 1 can be computed in constant time per point. As before, if the upper bound is smaller than the lower bound on the searched farthest distance, there is no need to compute the exact distance.

In general, this method can be expected to be less effective in pruning distance computations than the methods of Sections 3.1 and 3.2. To see why, consider that the bound in Equation 1 too expresses the triangle inequality:

$$\begin{aligned}\|p_j-c_i\| &\leq \sqrt{\|p_j\|^2+\|c_i\|^2+2\|p_j\|\|c_i\|} \\ &= \sqrt{(\|p_j\|+\|c_i\|)^2} \\ &= \|p_j\|+\|c_i\|.\end{aligned}$$

In this interpretation, instead of utilizing center points computed in earlier passes of the algorithm, this filtering method repeatedly applies the triangle inequality to the origin o, in addition to p_j and c_i. Thus, expressed with a general distance measure D, the bound becomes

$$D(c_i,p_j)\leq D(o,p_j)+D(o,c_i).$$

Consequently, the distance $D(c_i,p_j)$ is approximated well only when at least one of $D(o,p_j)$ and $D(o,c_i)$ are sufficiently small. Thus, in order to get effective filtering, either most of the input points must be clustered close to the origin, or the sequence of generated center points must be near the origin. In other cases, the distances tend to be largely overestimated by this bound, leading to poor filtering.

3.4. Distance filtering on the GPU

All of the above distance filtering methods require an additional filtering step before invoking the distance computation kernel in each pass. In this step, it is determined in parallel which of the points in P need to have their distances computed exactly. Although the details of how this is determined differs between the methods, the result is a vector b of n binary values, where $b_j=1$ if the distance to point p_j needs to be computed, and $b_j=0$ otherwise. The vector b then goes through a parallel compaction step, which turns it into a vector x containing all indices j such that $b_j=1$. For example, given a vector

$$b=(0,1,0,1,1,0,0,1),$$

the compaction yields

$$x = (2,4,5,8),$$

assuming 1-based indexing. The vector x is then provided as an argument to the distance computation kernel, which processes the rows of the matrix P indirectly through x. In the case of the filtering methods of Sections 3.1 and 3.2, the vector x is then used once again to store the computed distances back into the right positions of the distance array.

4. Experiments

The discussed techniques were evaluated in practice on a laptop equipped with an Intel i7-3820QM processor (2.7 GHz) as well as an Nvidia Quadro K4000M GPU, whose specifications are listed in Figure 4. Sequential CPU implementations, written in C++ and compiled with Visual Studio 2012, as well as parallel GPU implementations, based on CUDA version 6.0, were tested. In addition, all three of the distance filtering methods discussed in Section 3 were evaluated in both the CPU and the GPU versions. Finally, two implementations following the approach outlined in Section 2 were included, using the column-major and row-major (transposed) versions of SGEMV (single-precision GEMV) in cuBLAS. Throughout all experiments, single-precision floating-point was used, and the approximation quality of the balls was set to $\varepsilon = 10^{-3}$. No robustness issues were observed in the experiments.

To tune the performance of our hand-written kernels, we executed an automatic process reminiscent of auto-tuning (cf. [Sør12, DO12]) to find good choices for the kernel launch parameters, as well as to select between our two distance computation kernels. For each pair of d and n included in the experiments, we executed both kernels using different sets of parameters and kept track of the most efficient configuration. The first kernel was run using all combinations of the following parameters:

$$t_d \in \{16,32,64\},$$
$$blockDim.y \in \{\frac{t_d}{1}, \frac{t_d}{2}, \frac{t_d}{4}, \ldots, 1\},$$
$$w_x, w_y \in \{1,2,4,\ldots,512\}.$$

The kernel designed for tall matrices was evaluated with the following parameters:

$$t_y \in \{16,32,64,\ldots,512\},$$
$$blockDim.y \in \{\frac{t_y}{1}, \frac{t_y}{2}, \frac{t_y}{4}, \ldots, 1\},$$
$$w_y \in \{1,2,4,\ldots,512\}.$$

Of course, many combinations of the parameters above are not valid, either due to hardware limitations or because they are not applicable to the particular problem size, so these were skipped. The parameters selected for each problem size were then used regardless of whether filtering was enabled or not.

Architecture	Kepler
SMs	5
CUDA cores	960
Clock rate	600 MHz
Memory clock rate	1.4 GHz
Memory bus width	256 bits

Figure 4: *Specifications for Nvidia's Quadro K4000M GPU.*

4.1. Moderate to high dimensions

The results from SIMPLEAPXBALL and FASTAPXBALL for moderate- to high-dimensional cases are shown in the left and right columns of Table 1, respectively. Nielsen and Nock's distance filtering method is denoted by NN, and our own filtering methods from Sections 3.1 and 3.2 are denoted by TI and TI2, respectively. For each of the selected pairs of d and n, the algorithms were executed ten times on different input sets randomly generated with a uniform distribution in $[-1,1]^d$. The table shows average figures from all ten runs: the number of passes k, the total number of full distance computations (d.c.) performed during the farthest-point queries as a fraction of $k \times n$ (which is the number of such computations in the non-filtering version), the execution time in seconds, and the speedup factor s. The latter parameter gives the average of the speedup factors computed relative to the sequential CPU version that does not use distance filtering.

The non-filtering GPU adaptations of the algorithms show speedups of up to 11×. The cuBLAS implementations exhibit good performance as long as the dimension is quite high, especially the column-major version. Judging from the case $d = 10$, however, the GEMV kernels in cuBLAS seem less optimized for tall matrices. The GPU implementations based on our custom kernels are significantly faster than both of the cuBLAS implementations in this case.

In $d = 10$, the tall matrix kernel was selected in the tuning process, with the parameters t_y, $blockDim.y$, and w_y chosen as shown in parentheses for that case in Table 1. In the remaining cases, the more general kernel proved to be more efficient. Similarly, the parameters t_d, $blockDim.y$, w_x, and w_y selected for this kernel for each problem size are shown in parentheses in the table. In general, the former kernel seems to be the most efficient in dimensions up to $d \approx 20$, as indicated by additional test runs. Notice that the launch parameters for the more general kernel were consistently selected so as to give only one thread block in the horizontal direction, i.e., such that $t_x \times w_x \geq d$. It seems natural that this is an efficient division of labor among the thread blocks, as it allows each thread to sum up its computed squared differences along an entire tile row using registers before the reduction in shared memory takes place. Thus, with only one thread block in the horizontal direction, the number of such

SIMPLEAPXBALL			
$d = 10, n = 10^6, k = 728.1$			
algorithm	**d.c.**	**time**	s
CPU	1.000	6.650	1.0
CPU, NN	0.000	0.724	9.2
CPU, TI	0.004	1.384	4.8
CPU, TI2	0.021	1.764	3.8
cuBLAS, r.m.	1.000	5.752	1.2
cuBLAS, c.m.	1.000	2.446	2.7
GPU $(128, 16, 16)$	1.000	0.877	7.6
GPU, NN	0.003	0.467	14.2
GPU, TI	0.005	0.594	11.2
GPU, TI2	0.022	0.632	10.6
$d = 100, n = 10^5, k = 769.7$			
algorithm	**d.c.**	**time**	s
CPU	1.000	5.888	1.0
CPU, NN	0.166	1.685	3.7
CPU, TI	0.014	0.272	21.7
CPU, TI2	0.047	0.605	9.7
cuBLAS, r.m.	1.000	1.219	4.8
cuBLAS, c.m.	1.000	0.855	6.9
GPU $(32, 8, 4, 32)$	1.000	0.962	6.1
GPU, NN	0.195	0.568	10.5
GPU, TI	0.020	0.473	12.5
GPU, TI2	0.055	0.520	11.4
$d = 500, n = 10^5, k = 618.5$			
algorithm	**d.c.**	**time**	s
CPU	1.000	25.106	1.0
CPU, NN	0.885	22.811	1.1
CPU, TI	0.040	1.301	19.3
CPU, TI2	0.094	2.837	8.8
cuBLAS, r.m.	1.000	2.954	8.5
cuBLAS, c.m.	1.000	2.314	10.9
GPU $(32, 8, 16, 32)$	1.000	2.376	10.6
GPU, NN	0.906	3.090	8.1
GPU, TI	0.060	0.836	30.2
GPU, TI2	0.114	0.941	26.7
$d = 1000, n = 10^4, k = 407.9$			
algorithm	**d.c.**	**time**	s
CPU	1.000	3.346	1.0
CPU, NN	1.000	3.362	1.0
CPU, TI	0.120	0.493	6.8
CPU, TI2	0.199	0.710	4.7
cuBLAS, r.m.	1.000	0.478	7.0
cuBLAS, c.m.	1.000	0.422	7.9
GPU $(32, 8, 32, 16)$	1.000	0.395	8.5
GPU, NN	1.000	0.547	6.1
GPU, TI	0.174	0.590	5.7
GPU, TI2	0.249	0.484	6.9
$d = 5000, n = 10^4, k = 348.3$			
algorithm	**d.c.**	**time**	s
CPU	1.000	14.354	1.0
CPU, NN	1.000	14.404	1.0
CPU, TI	0.302	4.655	3.1
CPU, TI2	0.402	5.844	2.5
cuBLAS, r.m.	1.000	1.609	8.9
cuBLAS, c.m.	1.000	1.346	10.7
GPU $(32, 8, 256, 16)$	1.000	1.333	10.8
GPU, NN	1.000	1.668	8.6
GPU, TI	0.434	1.990	7.2
GPU, TI2	0.516	1.489	9.6

FASTAPXBALL			
$d = 10, n = 10^6, k = 14.9$			
algorithm	**d.c.**	**time**	s
CPU	1.000	0.137	1.0
CPU, NN	0.002	0.025	5.5
CPU, TI	0.222	0.063	2.2
CPU, TI2	0.344	0.105	1.3
cuBLAS, r.m.	1.000	0.126	1.1
cuBLAS, c.m.	1.000	0.059	2.3
GPU $(128, 16, 16)$	1.000	0.019	7.4
GPU, NN	0.150	0.026	5.1
GPU, TI	0.260	0.023	6.0
GPU, TI2	0.352	0.024	5.6
$d = 100, n = 10^5, k = 56.2$			
algorithm	**d.c.**	**time**	s
CPU	1.000	0.440	1.0
CPU, NN	0.361	0.227	2.0
CPU, TI	0.191	0.129	3.4
CPU, TI2	0.280	0.183	2.4
cuBLAS, r.m.	1.000	0.107	4.1
cuBLAS, c.m.	1.000	0.082	5.4
GPU $(32, 8, 4, 32)$	1.000	0.080	5.5
GPU, NN	0.519	0.086	5.1
GPU, TI	0.270	0.065	6.8
GPU, TI2	0.342	0.069	6.4
$d = 500, n = 10^5, k = 127$			
algorithm	**d.c.**	**time**	s
CPU	1.000	5.237	1.0
CPU, NN	0.953	5.093	1.0
CPU, TI	0.187	1.167	4.5
CPU, TI2	0.274	1.674	3.1
cuBLAS, r.m.	1.000	0.703	7.5
cuBLAS, c.m.	1.000	0.576	9.1
GPU $(32, 8, 16, 32)$	1.000	0.560	9.4
GPU, NN	0.968	0.749	7.0
GPU, TI	0.274	0.370	14.2
GPU, TI2	0.349	0.393	13.3
$d = 1000, n = 10^4, k = 149.1$			
algorithm	**d.c.**	**time**	s
CPU	1.000	1.348	1.0
CPU, NN	1.000	1.359	1.0
CPU, TI	0.277	0.489	2.8
CPU, TI2	0.369	0.599	2.3
cuBLAS, r.m.	1.000	0.307	4.4
cuBLAS, c.m.	1.000	0.288	4.7
GPU $(32, 8, 32, 16)$	1.000	0.269	5.0
GPU, NN	1.000	0.328	4.1
GPU, TI	0.394	0.339	4.0
GPU, TI2	0.469	0.312	4.3
$d = 5000, n = 10^4, k = 269.3$			
algorithm	**d.c.**	**time**	s
CPU	1.000	11.690	1.0
CPU, NN	1.000	11.736	1.0
CPU, TI	0.364	4.843	2.4
CPU, TI2	0.462	5.780	2.0
cuBLAS, r.m.	1.000	1.853	6.3
cuBLAS, c.m.	1.000	1.647	7.1
GPU $(32, 8, 256, 16)$	1.000	1.618	7.2
GPU, NN	1.000	1.873	6.2
GPU, TI	0.516	2.055	5.7
GPU, TI2	0.588	1.770	6.6

Table 1: *Experimental results for uniformly distributed input. Timings are given in seconds.*

SIMPLEAPXBALL					
model	n	k	CPU	cuBLAS	GPU
Lucy	14.0M	15	0.595	0.256	0.061
Thai Statue	5.0M	584	8.215	2.969	0.929
Vase	4.6M	721	9.435	3.424	1.116
Asian Dragon	3.6M	730	7.416	2.727	0.885
Goblet	1.0M	830	2.354	1.047	0.386

FASTAPXBALL					
model	n	k	CPU	cuBLAS	GPU
Lucy	14.0M	5	0.198	0.120	0.020
Thai Statue	5.0M	8	0.113	0.060	0.013
Vase	4.6M	7	0.088	0.051	0.011
Asian Dragon	3.6M	6	0.061	0.037	0.007
Goblet	1.0M	10	0.028	0.017	0.005

Table 2: *Experimental results for polygon meshes in 3D. The Lucy, Thai Statue, and Asian Dragon models are provided by the Stanford Computer Graphics Laboratory. Timings are given in seconds.*

reductions per row is minimized. Furthermore, it enables the kernel-fusion optimization described in Section 2.1.

All the distance filtering techniques show successful reductions in distance computations under certain circumstances, leading to quite impressive speedups. In $d = 10$, Nielsen and Nock's method is the most effective, but loses its effectiveness rapidly as the dimension grows. Given the uniform distribution of these point sets, this is an expected result, as no clusters tend to be formed around the origin. Furthermore, since it becomes increasingly improbable that c^* occurs in the vicinity of the origin in the higher dimensions, it also becomes less likely that c_i does. The triangle inequality-based methods, on the other hand, show less pruning power in $d = 10$, but give better results in higher dimensions.

In the GPU case, it is clear that the overhead of the additional kernel invocations needed to realize the filtering procedure has a limiting effect on the achieved performance. In fact, the introduction of distance filtering can be seen to decrease performance in several cases, particularly in the higher dimensions, where fewer distance computations are skipped. In this regard, the CPU algorithms have the benefit that testing the filtering condition and updating the cached distances can be done in an integrated fashion during a single traversal of the point set. Nevertheless, several successful cases can be observed as well, indicating a potential of the presented approach.

Noteworthy here is also that the sequential implementations possess an additional opportunity to skip more distance computations compared to the parallel implementations: during the sequential scan for the farthest point, the lower bound on the largest distance is updated continuously as new candidate farthest points are encountered. Thus, the upper distance bounds are compared against a gradually increasing lower bound, whereas in the parallel distance filtering, the same lower bound is used throughout the whole pass. The effects of this optimization can be seen in column d.c. in Table 1 by comparing the figures of the CPU and GPU versions that use the same filtering method.

4.2. Polygon meshes

To evaluate the presented techniques also in low dimensions, and to include more realistic data sets in the experiments, we executed the algorithms with a selection of 3-dimensional polygon meshes as input. The results from this experiment are shown in Table 2. Listed for each case are the number of vertices n in the model, the total number of passes k, and the run time in seconds of three of the evaluated implementations. Included here are the non-filtering CPU versions, the GPU versions based on the column-major kernel in cuBLAS, and the non-filtering GPU versions using the kernel optimized for tall matrices. The parameters used for the latter were $t_y = 512$, $blockDim.y = 64$, and $w_y = 64$ in all runs.

The results from the GPU versions based on the tailor-made distance computation kernel are encouraging, with speedups in the range 5.8–9.8× on both algorithms. The cuBLAS-based versions, on the other hand, give somewhat disappointing speedups of at most 2.8× on SIMPLEAPXBALL and 1.9× on FASTAPXBALL. Again, this indicates a lack of support for tall matrices in the cuBLAS GEMV kernels.

Note that in the 3-dimensional case, less performance

benefits can be expected from using the discussed distance filtering techniques, in the GPU case as well as the CPU case. As it takes only 8 arithmetic operations to compute a squared distance, the relative savings from filtering such a computation is limited. On the above input examples, distance filtering gave occasional performance improvements of up to 2.1× in the CPU version of SIMPLEAPXBALL, and minor slowdowns to modest speedups in the CPU version of FASTAPXBALL.

5. Conclusions

Given the extensive applicability of minimum enclosing ball algorithms in both low and high dimensions, we expect performance studies and speed-up techniques, such as the ones presented in this paper, to be beneficial in several research communities. Clearly, the offloading of the repeated farthest point queries to massively parallel GPUs pays off with speedups up to 11×. Also, the presented algorithmic techniques for distance filtering give additional opportunities for savings in execution time both in the sequential and the parallel implementations.

The proposed distance filtering approaches can be harnessed also in applications using other distance measures than Euclidean distance, as long as the used distance function obeys the triangle inequality. Furthermore, despite that only points sets were considered as input here, it is straightforward to generalize the presented techniques to deal also with ball sets.

In the future, it would also be exciting to compare other possible acceleration techniques with the strategies presented here, such as considering alternative ways of parallelization, different pruning methods [KL], usage of k-farthest points queries in each algorithm pass, and hierarchical searching using multidimensional tree structures (see, e.g., [CPZ97]). This also includes evaluating combinations of the approaches, such as using pruning techniques to eliminate points permanently combined with distance filtering on the remaining points, as well as more aggressive and diverse algorithm parallelization on heterogeneous computing platforms with support for different compute targets (CPU, GPU, and FPGA). Hopefully, such studies could provide a basis for the design of even faster ball computation algorithms.

Acknowledgements

Both authors are supported by a research grant from the Swedish Foundation for Strategic Research (No. IIS11-0060).

References

[BC03] BÂDOIU M., CLARKSON K. L.: Smaller core-sets for balls. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms* (2003), pp. 801–802. 1, 2

[BEKS01] BRAUNMULLER B., ESTER M., KRIEGEL H.-P., SANDER J.: Multiple similarity queries: a basic DBMS operation for mining in metric databases. *IEEE Transactions on Knowledge and Data Engineering 13*, 1 (Jan 2001), 79–95. 4

[BH12] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for CUDA. In *GPU Computing Gems Jade Edition*, Hwu W.-m. W., (Ed.). Morgan Kaufmann Publishers Inc., 2012, pp. 359–371. 2

[BK73] BURKHARD W. A., KELLER R. M.: Some approaches to best-match file searching. *Communications of the ACM 16*, 4 (Apr. 1973), 230–236. 4

[BS98] BERMAN A., SHAPIRO L.: Selecting good keys for triangle-inequality-based pruning algorithms. In *IEEE International Workshop on Content-Based Access of Image and Video Database* (Jan 1998), pp. 12–19. 4

[CPZ97] CIACCIA P., PATELLA M., ZEZULA P.: M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases* (1997), Morgan Kaufmann Publishers Inc., pp. 426–435. 9

[DDG*] DONGARRA J., DONG T., GATES M., HAIDAR A., TOMOV S., YAMAZAKI I.: MAGMA: a new generation of linear algebra library for GPU and multicore architectures. 2

[DO12] DAVIDSON A., OWENS J.: Toward techniques for auto-tuning GPU algorithms. In *Applied Parallel and Scientific Computing*, vol. 7134 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 110–119. 6

[Gär99] GÄRTNER B.: Fast and robust smallest enclosing balls. In *Proceedings of the 7th Annual European Symposium on Algorithms* (1999), Springer-Verlag, pp. 325–338. 2

[HS03] HJALTASON G. R., SAMET H.: Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems 28*, 4 (Dec. 2003), 517–580. 4

[KL] KÄLLBERG L., LARSSON T.: Improved pruning of large data sets for the minimum enclosing ball problem. *Graphical Models (to appear)*. 2, 9

[KMY03] KUMAR P., MITCHELL J. S. B., YILDIRIM E. A.: Approximate minimum enclosing balls in high dimensions using core-sets. *Journal of Experimental Algorithmics 8* (2003). 1, 2

[LK13] LARSSON T., KÄLLBERG L.: Fast and robust approximation of smallest enclosing balls in arbitrary dimensions. *Computer Graphics Forum 32*, 5 (2013), 93–101. 1

[NN04] NIELSEN F., NOCK R.: Approximating smallest enclosing balls. In *Proceedings of International Conference on Computational Science and Its Applications (ICCSA)* (2004), vol. 3045 of *Lecture Notes in Computer Science*, Springer. 5

[PS85] PREPARATA F. P., SHAMOS M. I.: *Computational Geometry: An Introduction*. Springer-Verlag New York, Inc., 1985. 1

[Sør12] SØRENSEN H. H. B.: High-performance matrix-vector multiplication on the GPU. In *Euro-Par 2011: Parallel Processing Workshops*, vol. 7155 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 377–386. 6

[TKK07] TSANG I. W., KOCSOR A., KWOK J. T.: Simpler core vector machines with enclosing balls. In *Proceedings of the 24th International Conference on Machine Learning* (2007), ACM, pp. 911–918. 1

[Yıl08] YILDIRIM E. A.: Two algorithms for the minimum enclosing ball problem. *SIAM Journal on Optimization 19*, 3 (Nov. 2008), 1368–1391. 1, 2