Proceedings of the 23rd UK Conference of the
Association for Computational Mechanics in Engineering
8 - 10 April 2015, Swansea University, Swansea

Calculating the Minimum Distance Between Two Triangles on SIMD Hardware

***Konstantinos Krestenitis[1], Tomasz Koziara[1]**

[1]School of Engineering, University of Durham, Durham, DH1 3LE

*konstantinos.krestenitis@durham.ac.uk

ABSTRACT

This paper presents an investigation into calculating the minimum distance between two 3D triangles on SIMD hardware. We formulate the problem as constrained minimization and apply a quasi-Newton method to solve its multiple instances simultaneously on separate SIMD vector lanes. We test Intel's open-source ISPC compiler [0], which provides an explicit programming model for writting vectorized code for modern CPUs and GPUs. In the current communication we share our initial experience of producing an optimized ISPC implementation and show some preliminary numerical results.

Key Words: *Triangles, ISPC, SIMD, Minimum Distance*

1. Introduction

We try to find the minimum distance between two 3-dimensional triangles exploiting the Single Instruction Multiple Data (SIMD) parallelism of modern CPUs. The inspiration for this work comes from a paper by Kris Hauser [0], where a robust contact point generation method is proposed in the context of rigid body simulations on unstructured meshes. To avoid ill conditioned contact points that produce jitter and divergence, he introduce a virtual margin around the bodies and sought for pairwise distance between triangles of so expanded meshes. Our optimized subroutine can not only be used to speed up a framework like this, but also, it demonstrates a more general approach to optimizing computational primitives that may seem difficult to improve upon at first. As explicit SIMD-enabled programming models become more available now, it becomes more imperative to exploit this level of hardware parallelism. Consequently, we share our experience in the hope that it can be useful to others.

2. Finding The Shortest Distance Between Two Triangles

Our baseline method is the brute force one. Let v_{1i}, e_{1i}, v_{2i}, e_{2i} be the vertices and edges of triangles T_1 and T_2. The brute force approach computes all fifteen distances v_{1i}-to-T_2, v_{2i}-to-T_1, e_{1i}-to-e_{2i}, and then selects the smallest. This approach is quite simple to implement. Optimization of this method can at most rely on reusing parts of the computation, but it is not possible to avoid a relatively complex branching and therefore it is hard to optimize this approach for SIMD hardware.

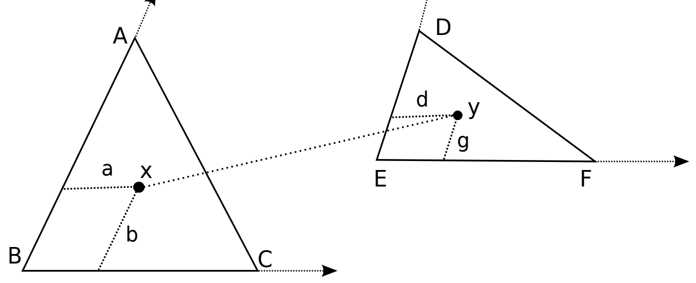

Figure 1: Barymetric Coordinates of a Triangle

Our optimized methods rely on an optimization based reformulation of the minimum distance problem. Triangles can be parameterized using barycentric coordinates. Let x and y be two points, belonging respectively to triangle T_1 and T_2. Assuming that points A, B, C are vertices of T_1 and that points D, E, F are vertices of T_2, x and y can be defined using the following equations: $T_1 : \overrightarrow{x}(a,b) = \overrightarrow{A} + (\overrightarrow{B} - \overrightarrow{A}) \cdot a + (\overrightarrow{C} - \overrightarrow{A}) \cdot b$, $T_2 : \overrightarrow{y}(g,d) = \overrightarrow{D} + (\overrightarrow{E} - \overrightarrow{D}) \cdot g + (\overrightarrow{F} - \overrightarrow{D}) \cdot d$ To find the minimum distance between T_1 and T_2 we minimize $f(a,b,c,d) = \left\| \overrightarrow{x}(a,b) - \overrightarrow{y}(c,d) \right\|^2$. What has to be noted is that for x and y to stay within the triangles, the four parameters of the function f have to comply with the six inequality constraints: find $\min f(a,b,g,d)$ such that $\{a \geq 0, b \geq 0, a + b \leq 1, d \geq 0, g \geq 0, g + d \leq 1\}$. This forms the basis of the optimization problem to be solved.

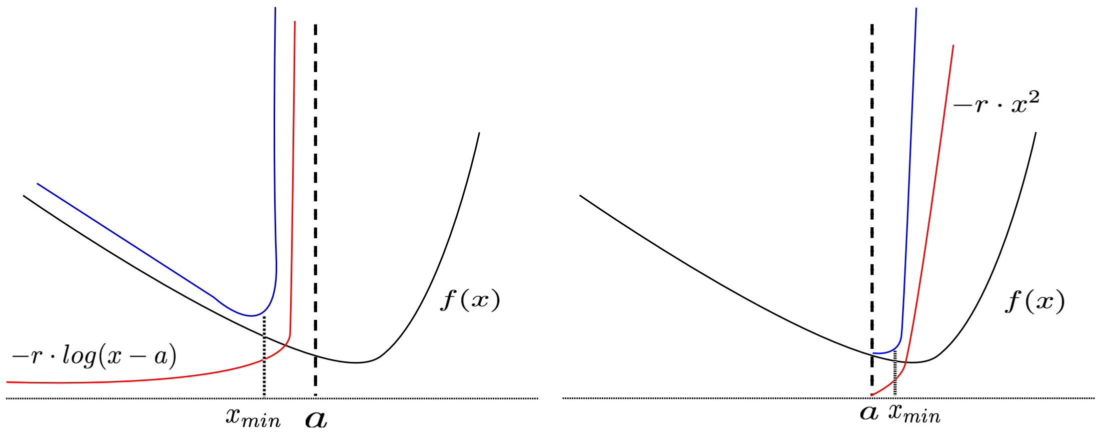

Figure 2: Illustration of: barrier method (left), penalty method (right)

In our exploration of suitable optimization approaches we begun by testing the barrier method. The barrier method exploits a logarithmic function in order to incorporate the constraints into an extended objective function: $B(x) = f(x) + r \sum_{i=1...6} -log(c(x_i))$, cf. Figure 2, left. The barrier method is characterized by its requirement to arrive at the solution from within the feasible region ($B(x)$ is not defined outside of the feasible region). In practice the barrier method has computational difficulties due to the fact that as the boundary of the feasible region is approached the log barrier function becomes infinite. An explicit check and re-positioning has to be made to guarantee that the search remains within the feasible region.

Our second method of choice, the penalty method, is on the other hand slightly different. This approach adds a penalty term to the objective function to penalize the solution when outside of the feasible region: $P(x) = f(x) + r \sum_{i=1...6} max(c(x_i))^2$, cf. Figure 2, right. Unlike the barrier method, the penalty method doesn't require us to check whether the solution remains within the feasible region at each step. One aspect that needs extra care however, is the invertiblity of the Hessian $\nabla\nabla P$. Unlike the barrier method, where the log terms are nonzero inside of the feasible region and help to regularize $\nabla\nabla B$, the penalty approach exposes the fact that f has multiple minima and $\nabla\nabla f$ is singular. Consequently, $\nabla\nabla P$ is also singular inside of the feasible region. To overcome that we resort ourselves to a quasi-Newton approach, where the Hessian is approximated by a perturbed operator $\nabla\nabla P + \epsilon I$, where I is an identity matrix and ϵ is suitably small. This regularization, also akin to pseudo-transient continuation [0], allows as to avoid line search at the same time. This renders so formulated penalty approach specifically attractive. In the remaining part of the current paper we only focus on the penalty approach.

3. Implementation

For the prototype implementation, included in Algorithm , we use MATLAB. The non-optimized code follows a simple sequential pattern. It starts by initializing the edge vectors, the hessian of f, `hf` and the solution in lines 1-6. It then involves a loop section for the Newton steps, where `X`, `Y` are the evaluation of the points on the triangles T_1, T_2 respectively. In line 6 `gf` is the gradient of the objective function. The constraints `h` are specified in vector form and their derivatives in a six by four matrix `dh`. Due to the penalty function $p(x) = r \sum_{i=1...6} max(c(x_i))^2$ we have to evaluate which constraints are active at each Newton step; we introduce a mask array for each of the six constraints and we use it to form a new matrix `dmax` that contains the active derivative constraints in line 11. The gradient `gra` and hessian `hes` is created to solve the Newton step `dx` by using the left slash operator. Finally the solution error is calculated on real geometry in line 21 and the termination condition is checked based on the tolerance supplied by the user.

Algorithm 1 MATLAB version of the TTD test.

```
   function x = ttd1(A, B, C, D, E, F, rho, tol)
1  BA = B-A; CA = C-A; ED = E-D; FD = F-D;
2  hf = [2*BA*BA', 2*CA*BA',-2*ED*BA',-2*FD*BA';
3        2*BA*CA', 2*CA*CA',-2*ED*CA',-2*FD*CA';
4       -2*BA*ED',-2*CA*ED', 2*ED*ED', 2*FD*ED';
5       -2*BA*FD',-2*CA*FD', 2*ED*FD', 2*FD*FD'];
6  x = [0.33; 0.33; 0.33; 0.33];
3  for i=1:99
4    X = A+BA*x(1) + CA*x(2);
5    Y = D+ED*x(3) + FD*x(4);
6    gf = [2*(X-Y)*BA'; 2*(X-Y)*CA'; -2*(X-Y)*ED'; -2*(X-Y)*FD'];
7    h = [-x(1); -x(2); x(1)+x(2)-1; -x(3); -x(4); x(3)+x(4)-1];
8    dh = [-1, 0, 1, 0, 0, 0; 0, -1, 1, 0, 0, 0;
9           0, 0, 0, -1, 0, 1; 0, 0, 0, 0, -1, 1];
10   mask = h' >= 0;
11   dmax = dh.* [mask; mask; mask; mask];
12   gra = gf + rho * dmax * max(0,h(:));
17   hes = hf + rho*dmax*dmax' + eye(4,4)/rho^2;
18   dx = hes\gra;
19   DX = BA*dx(1) + CA*dx(2);
20   DY = ED*dx(3) + FD*dx(4);
21   error = sqrt(DX*DX'+DY*DY');
22   if error < tol, break; end
23   x = x - dx;
24 end
25 end
```

The sequential MATLAB prototype forms the basis for the implementation of the optimized version of the algorithm. The optimized code uses the Intel SPMD Program Compiler (ISPC) [0]. ISPC is a C language extension for SIMD high performance CPU programming. It's designed to deliver very high performance on CPUs thanks to effective use of both multiple processor cores and SIMD vector units.

SIMD vector units allow to execute in parallel certain algebraic operations on data items with suitable memory alignment. Since our Newton method, cf. Algorithm , is relatively light, the key insight to optimizing it is in running multiple Newton methods (form multiple triangle pairs) in parallel on individual SIMD vector lanes. On each vector lane, we try to balance out the amount of computation and the amount of data movement (i.e. reads and writes) by suitably restructuring the algebra and the use of temporary variables. Hence, even though ISPC provides a suitable programming model, in order to take full advantage of the architecture of modern CPUs, it is still vital to hand craft the ISPC source code. We share the experience of optimizing Algorithm below.

The optimized C version exploits matrix symmetries to reduce repetitions of calculations for matrix elements, e.g. `hf` in the above code is using symmetry for that reason. For the same reason `X-Y` is only calculated once and stored into a variable `XY` so that its value is accessed instead of being repetitively calculated. Inside of the Newton loop the optimized algorithm is totally different from the prototype. The derivatives of the constraints are stored in an array instead of the sparse matrix, so only the non-zeroes are used. Operator $max(0,h(:))$ is calculated without the use of the std library function $max()$ which is too generic for our code. We replace it with if statements and directly assign values to an array of active derivatives of constraints `dmax`, avoiding the masking operations in lines 10-11. The gradient `gra` is calculated so that any redundant operations are removed. The same techniques are used for the hessian matrix `hes`. The point here is to end up with as few assignments as possible. The most significant aspect of the optimized algorithm is the linear solution in line 18. Unlike MATLAB, which exploits a separate linear solver, in our implementation individual operations of a 4x4 Gaussian elimination have been merged with the rest of the algorithm in a monolithic manner. This means that Gauss elimination, calculating of the gradient and calculation of the hessian are all fused together, still minimizing the number of necessary assignments to temporary variables. Operations like division are also limited because of their computational latency, operations like addition, subtraction and multiplication are preferred. Memory allocation is never done dynamically.

4. Performance

All methods were implemented in C in both sequential and SIMD code. The performance comparison comprises calculating the minimum distance between a pair of triangles replicated one thousand times and the average run-time is taken for both penalty and brute force approaches. As it can be observed

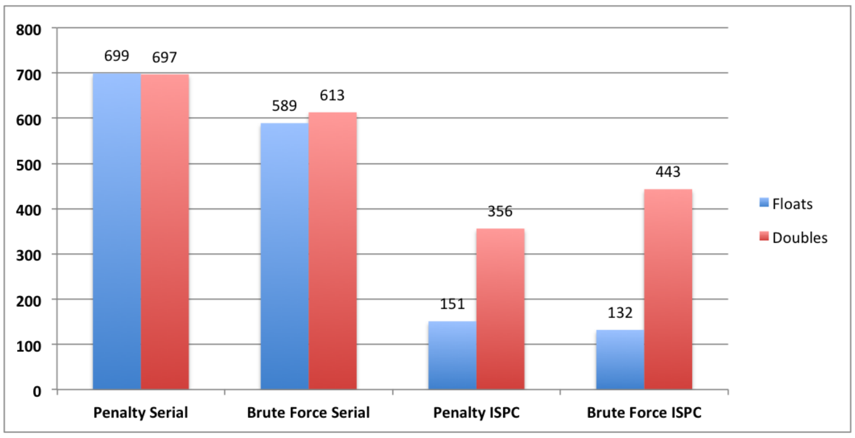

Figure 3: Illustration of run time comparison of the methods

in Figure from the comparison of the serial codes for double precision, the penalty is slightly slower than the brute force approach. But when we compare the ISPC version of both methods, there is a gain in performance for penalty approach because it can exploit SIMD vectors more efficiently. The ISPC version of the penalty approach has nearly doubled or quadripled its performance compared to its serial code for double and single precision versions respectively. Brute force method on the other hand behaves less consistently: its double precision ISPC version is outperformed by the penalty approach, while its single precision ISPC version outperforms the penalty method. We know that an ideal speedup for SIMD code is 8x in single and 4x in double precision, using the AVX instruction set on an Intel 2.3GHz i7 CPU that is used here. We think that achieving roughly half of this peak performance could be due to the throughput limit of the memory bus (1600 MHz DDR3 RAM), yet further checking is needed.

5. Conclusions

Tackling multiple small nonlinear problems is not uncommon in computational engineering. Our intention behind this study was to investigate whether such problems can still be optimized using SIMD hardware and ISPC. Our optimized penalty implementation shows that it is indeed possible to computationally gain from SIMD implementation and even outperform the brute force method (double precision). Further performance testing will be conducted for random sets of triangles. Furthermore a thorough performance profiling will be done in the hope of resolving the current computational bottleneck (i.e. the 2x and 4x speedup achieved versus the 4x and 8x speedups expected, based on SIMD vector width for AVX instruction sets used here).

References

[1] Kris Hauser. Robust Contact Generation for Robot Simulation with Unstructured Meshes. *Intl. Symposium on Robotics Research*, 2013.

[2] Matt Pharr, William R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. Intel Corporation. *Innovative Parallel Computing (InPar)*, 1 - 13. 2012.

[3] Todd S. Coffey, C. T. Kelley, David E. Keyes. Pseudo-transient continuation and differential-algebraic equations. *SIAM J. Sci. Comp*, Vol. 25, 553–569. 2013.