An Arbitrary N-dimensional Stencil Transformer in Cilk++*

Yuan Tang
yuantang@csail.mit.edu

Steven Bartel
sbartel@mit.edu

Dina Kachintseva
dinka@mit.edu

ABSTRACT

Stencil computation is derived directly from solving partial differential equations (PDEs). The conventional way of solving the stencil problem is using nested loops, which sweep over a spatial grid, updating each point at time $t+1$ by neighboring grid points at time t, $t-1$, ..., $t-k$. These kinds of loop algorithms, along with all their variants, inevitably suffer from consistent cache misses due to their memory access pattern. Matteo Frigo's paper [3, 5, 4] invented a cache oblivious algorithm which greatly reduces the cache miss ratio. In this paper, we explored all known optimizations for stencil computations, along with some of our own innovative approaches, and concluded with concrete performance graphs which approaches are effective and which ones are not.

Keywords

stencil computation, cache aware algorithm, cache oblivious algorithm, optimization

1. BACKGROUND

Partial differential equation (PDE) solvers constitute a large fraction of scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics. These applications are often implemented using iterative finite-difference techniques, which sweep over a spatial grid, performing nearest neighbor computations called *stencils* [6, 2]. Formally, in a stencil computation, [5] each point in an n-dimensional spatial grid at time t is updated as a function of neighboring grid points at time $t-1$, ..., $t-k$, —- thereby representing the coefficients of the PDE for that data element. The n-dimensional grid plus the time dimension span an $(n+1)$-dimensional spacetime. These operations are then used to build solvers ranging from simple Jacobi iterations to complex multi-grid and adaptive mesh refinement methods.

Stencil computations perform global sweeps through data structures that are typically much larger than the capacity of available data caches. As a result, stencil computations generally achieve a low fraction of theoretical peak performance, since data from main memory cannot be transferred fast enough to avoid stalling the computational units on modern micro-processors.

In class, we are introduced to a cache-oblivious algorithm invented by Matteo Frigo [5, 4] that by employing the divide-and-conquer strategy, and by conducting cuts in both spacial dimensions and time dimension, we are able to save a factor of $O(Z^{1/n})$ [1] cache misses compared to a naive algorithm in the serial version and incur only $O(n2/Z + n + \sqrt{Pn^{3+\epsilon}})$ cache misses with high probability in parallel version.

However, without advanced optimization and tuning techniques, the cache-oblivious algorithm will be less likely to outperform the cache-aware time-skewing algorithm [1]. In the serial version, the main bottleneck to higher performance is the excessive overhead incurred by too many recursive function calls. In the parallel version, the situation is exacerbated by the lack of parallelism.

2. GOALS

Based on the advanced technology of Cilk++, we are going to develop a generalized cache-oblivious Cilk++ stencil computation program which will accomplish the following goals:

1. optimize to achieve fastest possible runtime
2. generalize the computation to arbitrary n-dimensions
3. dealing with boundary condition
4. dealing with irregular shaped stencil computation

3. SUMMARY

In order to make our stencil transformer faster, we have explored the following approaches:

- coarsening strategy for the recursive cache oblivious algorithm, which boosts the performance by 16%
- cutting heuristics for spatial dimensions, which gains another 12% in performance

*We name it transformer because we are going to leverage the template meta-programming techniques to transform the input specification written in some meta-language to some internal DAG , and then transform it to final Cilk++ code after applying our optimizations on internal DAG.

[1] Z is the size of an Ideal cache

MIT course 6.884 Concepts in Multicore Programming

- zero padding technique to eliminate the branching overhead in boundaries. The performance gain from this technique is 25%
- SIMDizing, which provides a 10% improvement in performance.
- optimizing 'N', which generates N-specific code for any specified 'N' from the original template. The final code is 42% more efficient than the template.

By applying all these optimization to the cache oblivious stencil algorithms, our software achieves a 2.6 times speedup compared to the naive parallelization of nested loops.

In summary, our software has several key features, which make our software relatively unique in the field of stencil computation:

- Our software can compute arbitrary N-dimensional stencils, based on the user's specification.
- Our software can compute a stencil on an irregularly shaped grid, which means that the shape of the spatial dimensions is no longer restricted to being rectangular.
- We can combine both periodic boundary conditions and non-periodic boundary conditions into one stencil problem.
- We exploited SIMD plus loop unrolling techniques to further accelerate the execution of base case. And we plan to compare this manually optimized code with what could be potentially done by the compiler.
- Static elaboration. Although currently this technique has not shown significant performance advantages, it can be used as a medium to mingle different algorithms for stencil computation in one run, which might be of some theoretical interest.

4. TESTING PLATFORM

We test all of our algorithms and optimizations on a 8-core Cagnode machine with roughly 8 GB of memory. The processor is Intel Xeon X5460 @ 3.16GHz, with cache size of 6 MB per core. Unless otherwise noted, we run our simulations on all 8 cores.

Unless specified, each test contains 100 different simulations, ranging from small grid sizes and small time steps to large grid sizes and large timesteps. Each simulation is the average of three trials. We generate 3D plots in which the width of the dimensions is on the x-axis, the number of time steps is on the y axis, and the performance, measured in ms, is on the z-axis. On multi-dimensional stencil computations, all of the dimensions have the same width. A higher curve translates to higher runtime, or worse performance. Our base case kernel applies the n-dimensional heat diffusion equation.

5. OPTIMIZATIONS

5.1 Cutting Heuristics

The cache oblivious stencil algorithm, outlined in Matteo Frigo's paper [3, 5, 4], cuts space into as many trapezoids as possible. We explored the idea of cutting into a fixed number, r, of trapezoids on space cuts. Figure 1 demonstrates two types of cuts: one where space is cut into as many trapezoids as possible, and one where space is cut into a fixed number of trapezoids ($r = 2$).

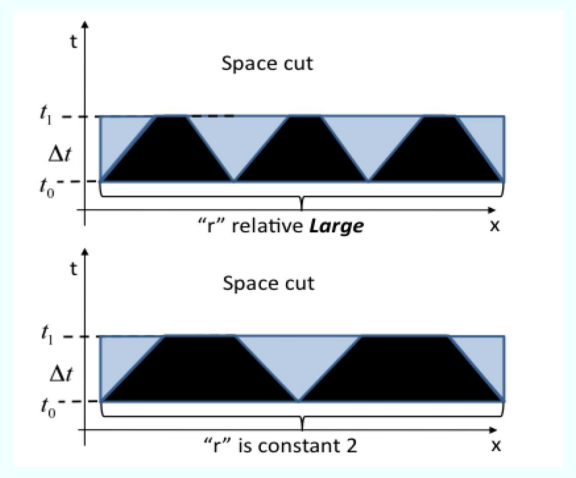

Figure 1: In the cache oblivious stencil algorithm, space can be cut into as many trapezoids as possible (top), or a fixed number of trapezoids (bottom).

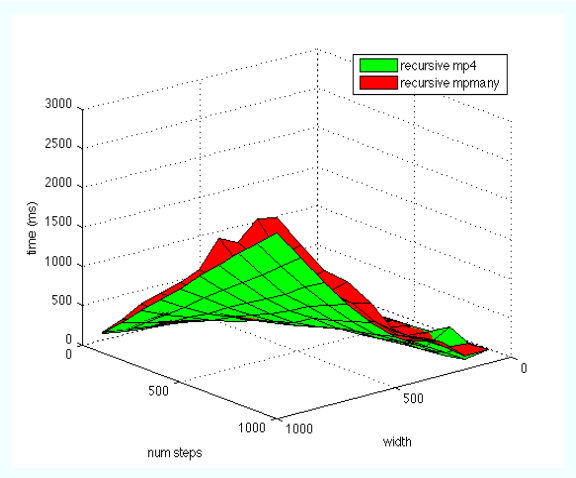

Figure 2: Performance of cutting into 4 trapezoids vs cutting into as many as possible.

We tested different numbers of cuts $r = 2, 4, 8, 16$ and as many as possible on grids of varying size and time steps. On average, $r = 4, 8$ (539 ms, 552 ms respectively) outperformed $r = 2, 16$ and as many as possible (615 ms, 622 ms, 613 ms respectively). Figure 2 shows how $r = 4$ compares to as many as possible. On the smallest grid (100×100), the speedup is 17.49 ms vs 18.36 ms (4.7%) and on the largest grid (1000×1000), the speedup is 539.12 ms vs 613.50 ms (12%). In nearly every case, $r = 4$ performs better than as many as possible. $r = 8$ is omitted because it is nearly identical to $r = 4$. $r = 2, 16$ are omitted because they are nearly identical to as many as possible.

5.2 Coarsening Strategy

In the paper [5, 4], the cache oblivious algorithm for sten-

cils looks something like the following:

```
Walk (...) {
      if (lt <= T_STOP) {
        base_case(...);
      } else if (lt > T_STOP) {
        if (x1-x0 > 4 * slope_x * lt) {
          /* cut into X dimension */
          ...
        }
      } else {
        /* cut into Time dimension */
        Walk(t0, t0+lt/2, ...);
        Walk(t0+lt/2, t1, ...);
      }
}
```

This implementation has several weaknesses. Firstly, the coarsening is conducted in the time dimension only. Secondly, it always cuts into space before any attempt to cut into the time dimension, which means that the space cube may be too small by the time the algorithm first cuts into time. So it loses the benefits of coarsening in space if there are any. Thirdly, the computation of the base case can only be triggered by a previous cut in time, which is severely serialized in execution. Even worse, we can imagine that for some extreme case in which we compute a stencil on a one billion by one billion grid and only one time step, the parallel version of the cache oblivious algorithm will regress to a serial loop.

If we understand the problem, the solution is also very straightforward: we step into the base case computation depending on the size of all space and time dimensions. Moreover, we must put in some guards to prevent small cuts into space dimensions. The program will look something like the following:

```
Walk(...) {
    if (lt <= T_STOP &&
      lx <= X_STOP && ly <= Y_STOP) {
      base_case(...);
    } else {
      if (lx > 4 * slope_x * lt && lx > X_STOP) {
        /* cut into X dimension */
        cilk_spawn Walk(black1);
        Walk(black2);
        cilk_sync;
        cilk_spawn Walk(gray1);
        cilk_spawn Walk(gray2);
        Walk(gray3);
        return;
      } else if (ly > 4 * slope_y * lt && ly > Y_STOP) {
        /* cut into Y dimension */
      } else {
        /* cut into Time dimension */
        Walk(t0, t0+lt/2, ...);
        Walk(t0+lt/2, t1, ...);
      }
    }
}
```

5.3 Overlap Time-Skewing

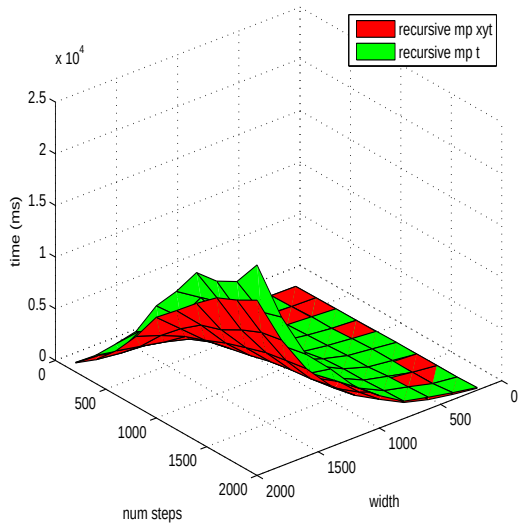

Figure 3: Speedup of coarsening on lt only versus coarsening on all space and time dimension

Both the parallel naive algorithm and the cache oblivious recursive algorithm suffer from communication overhead when two cores compute on adjacent trapezoids near the boundaries in parallel. The parallel naive algorithm also suffers from burdened parallelism because it spawns both grey and black trapezoids (see Figure 1). We hoped to address these issues by applying overlapped time-skewing to the parallel naive algorithm. In overlap time skewing, bases of trapezoids overlap, resulting in redundant computation as shown in Figure 4. In addition, these regions must be copied to avoid race conditions. However, the entire space cut can be computed in parallel because there is no longer any need for grey trapezoids.

We implemented overlap time-skewing by using a toggle array that spans the entire grid outside of each base case, and smaller toggle arrays per base case trapezoid. When stepping into the base case, values are read from the large toggle array into the base case trapezoid toggle array. For interme-

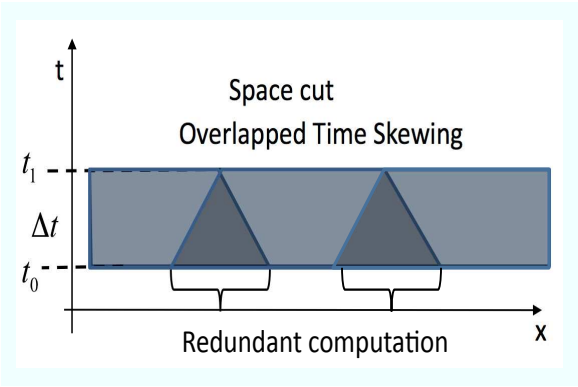

Figure 4: In overlap time skewing, redundant computation is performed to remove the grey trapezoids from Figure 1.

diate time steps, computation is done using the trapezoid toggle array and then the results are written back out to the large toggle array on the last base-case time step.

On the 2D heat equation kernel, overlap time-skewing performed better on small grids (smaller than 500x500), but worse on large grids (larger than 500x500). Presumably, cache misses have a larger effect on larger grids, because less of the grid fits into cache. Because the overlap time-skewing approach uses an additional toggle-array per trapezoid, it needs to load more data, which we guess hurts performance due to more cache misses.

5.4 Optimizing the Base Kernel for 2D

The base calculation for our 2D stencil computations uses the heat diffusion equation to update the heat of a given point in XY space at time $t+1$ using the heat of the surrounding points at time t. The original implementation of this heat calculation is shown below:

```
U(Q, t+1,x,y) = Q->CX * (U(Q, t,x+1,y)
       - 2.0 * U(Q, t,x,y) + U(Q, t,x-1,y))
            + Q->CY * (U(Q, t,x,y+1)
     - 2.0 * U(Q, t,x,y) + U(Q, t,x,y-1))
            + U(Q, t,x,y);
```

In order to speed up this calculation, we replaced the individual floating point operations for each dimension with SIMD instructions which perform these operations in parallel for the X and Y dimensions. Since an SIMD register can hold up to 128 bits, it can store a vector of two doubles, each containing the heat information for a given point in XY space. Thus, by creating four vectors:

```
1. vec1 = [ U(Q, t,x+1,y), U(Q, t,x,y+1) ]
2. twovec = [2.0 * U(Q, t,x,y), 2.0 * U(Q, t,x,y)]
3. vec2 =[U(Q, t,x-1,y), U(Q, t,x,y-1)]
4. cvec = [Q->CX, Q->CY]
```

six of the floating point operations shown in the code above can be executed using three SIMD operations, not including the load operation. The code for this is shown below:

```
asm volatile(
"movapd (%0), %%xmm0\n\t"
"addpd (%1), %%xmm0\n\t"
 "subpd (%2), %%xmm0\n\t"
 "mulpd (%3), %%xmm0\n\t"
 "movapd %%xmm0, (%0)"
 ::"r"(vec1),  "r"(vec2), "r"(twovec), "r"(cvec)
 );
```

The addpd, subbpd, mulpd instructions each perform two double operations - one for each of the space dimensions. Although additional space and time must be spent to instantiate the vector arrays for storing these double values, this is an acceptable cost if they are created only once per time step.

We compared performance of the SIMD instructions versus the original code on a series of 2D space heat grids, with sizes ranging from 100×100 to 1000×1000 and number of time steps ranging from 100 to 1000. The results of these tests are summarized in Table 1 and Figure 5. Based on these results, we observed that replacing the normal C++ code floating point operations with SIMD instructions created a speedup of 10% for the 2D code.

Table 1: Runtimes for 2D SIMD vs Original Base Kernel

	Original	SIMD
Range 100×100 to 1000×1000	41 ms to 6159 ms	37 ms to 5521 ms
Average	1378 ms	1237 ms

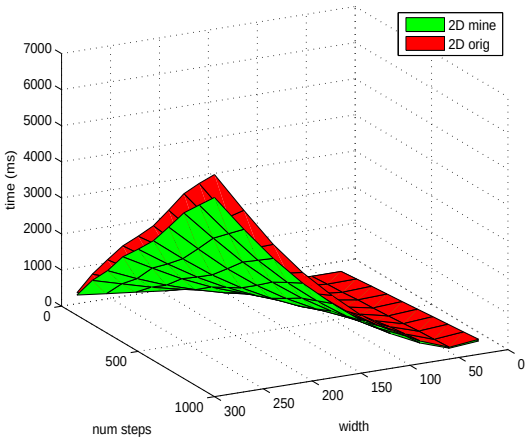

Figure 5: Runtimes for Stencil Computation Heat Diffusion Algorithm with Optimized vs Original 2D Kernel

6. GENERALIZE TO ARBITRARY N- DIMENSION

A fresh feature of our software is to provide the user the option to calculate stencils on arbitrary N-dimension grids. That is, we provide some template, and based on user's input we generate a stencil code of a specific number of dimensions from the template. Firstly, it is generated from the template, so there's no additional overhead incurred. Secondly, it provides the user with more options.

With this new feature, we have tried up to 8 dimensions, 10^8 grids can be computed, which consumes $2 * (size + 2 * slope)^N * 8$ bytes of memory. The only constraint is the amount of memory that system can allocate to a user level application.

Currently, we use a Python script to generate the final code out of this arbitrary N-dimensional template. In this code generation procedure, we have applied some optimizations, which will be detailed in the following section.

6.1 Optimizing the Base Case for N dimensions

The original implementation of the arbitrary N-dimensional case assumed that N would not be known until compile time, and thus used an *update_index* function to iterate over N-dimensional space for each time step. The code for this is shown below:

```
for (int t = t0; t < t1; t++) {
    while (!done) {
        kernel(Q, t, l_index);
        done = update_index(l_index,
                l_head_index, l_tail_index);
```

```
    }
...
}
```

To optimize this code, we took advantage of meta-programming techniques to expand the above while loop into a series of nested for loops, one for each space dimension being iterated over. We created a script that, given N, generates an implementation of the *base_case_kernel* method specific to the number of dimensions specified. Currently, this script is implemented in python; we would eventually replace it with the corresponding C++ metaprogramming. For the case when $N = 3$, our script would replace the above loop for iterating over N dimensional space with the following series of loops:

```
for (int t = t0; t < t1; t++) {
for(int x0= l_head_idx[0]; x0<l_tail_idx[0]; x0++){
for(int x1= l_head_idx[1]; x1<l_tail_idx[1]; x1++){
for(int x2= l_head_idx[2]; x2<l_tail_idx[2]; x2++){
....
```

In order to keep track of the point in N dimensional space being updated, our code uses an array of length N, called *l_idx*, to store the indexes of the current point. Within the *kernel* code to calculate the heat at a specific point, this array had to be incremented and decremented to perform each part of the heat calculation for each dimension. This code is shown below:

```
double tmp = 0;
l_idx[i]++;
tmp += U(Q, t, l_idx);
l_idx[i]--;
tmp += -2.0 * U(Q, t, l_idx);
l_idx[i]--;
tmp += U(Q, t, l_idx);
l_idx[i]++;
U(Q, t+1, l_idx) += Q->CX[i] * tmp;
```

Since the *l_idx* array has to be modified to perform each successive calculation, this implementation is difficult for the compiler to optimize. To change this, we had our N dimension generating script specify each array of indexes directly for each calculation done by the *kernel*. In addition, to optimize these calculations even further, we again replaced the C++ code floating point operations with SIMD instructions. In order to do this we had to unroll the N dimension loop so that it could calculate the heat update values for two dimensions at a time, instead of just one. Below is the heat update calculation code generated by our script for $N = 3$:

```
l_idx[0] = x0; l_idx[1] = x1; l_idx[2] = x2;
....
U(Q, t+1, l_idx) = U(Q, t, l_idx);
temp_index[0]=x0+1;temp_index[1]=x1;temp_index[2]=x2;
vec1[0] = U(Q, t, temp_index);
temp_index[0]=x0;temp_index[1]=x1+1;temp_index[2]=x2;
vec1[1] = U(Q, t, temp_index);
twovec[0] = 2*U(Q, t, l_idx);
twovec[1] = 2*U(Q, t, l_idx);
temp_index[0]=x0-1;temp_index[1]=x1;temp_index[2]=x2;
vec2[0] = U(Q, t, temp_index);
temp_index[0]=x0;temp_index[1]=x1-1;temp_index[2]=x2;
vec2[1] = U(Q, t, temp_index);
cvec[0] = Q->CX[0];
cvec[1] = Q->CX[1];
asm volatile(
 "movups (%0), %%xmm0\n\t"
"addps (%1), %%xmm0\n\t"
"subps (%2), %%xmm0\n\t"
"mulps (%3), %%xmm0\n\t"
"movups %%xmm0, (%0)"
::"r"(vec1),  "r"(vec2), "r"(twovec), "r"(cvec)
);
U(Q, t+1, l_idx) += vec1[0]+vec1[1];
```

In order to compare the performance of our optimized code against the original code, we performed tests on a series of 3D space heat grids, with sizes ranging from $50 \times 50 \times 50$ to $300 \times 300 \times 300$ and number of time steps ranging from 100 to 1000. We tested our optimized base case both for the loops and recursive implementations of our stencil algorithm. The results of these tests are summarized in Tables 2, 3, 4 and Figures 6, 7.

Table 2: Runtimes for 3D Script Generated Base Kernel vs Original Base Kernel for Loops Implementation

	Original	Optimized
$50 \times 50 \times 50$	129 ms	80 ms
$300 \times 300 \times 300$	193211 ms	189575 ms
Average	32669 ms	30763 ms

Table 3: Runtimes for 3D Script Generated Base Kernel vs Original Base Kernel for Recursive Implementation

	Original	Optimized
$50 \times 50 \times 50$	65 ms	47 ms
$300 \times 300 \times 300$	202124 ms	115489 ms
Average	34128 ms	19898 ms

Table 4: Speedup for Different Size 3D Grids for Optimized Base Kernel: Loops vs Recursive Implementation

	Loops	Recursive
$100 \times 100 \times 100$	30%	42%
$300 \times 300 \times 300$	–	42%

Based on these results, we observed that replacing the original N dimensional code with our script-generated code created a total speedup of 30% on small ($width \leq 100$) 3D space grids for the loops implementation, and 42% on *all* 3D space grids for the recursive implementation. For the loops implementation, as the size of the 3D space increases, the optimized implementation no longer performs better than the original implementation. Presumably, this is due to the fact that as the size of the 3D space gets larger, the dominating performance degrading factor becomes the high number of cache misses.

To determine the portion of the 42% speedup for the recursive implementation that is caused by the loop rewriting vs the portion caused by the SIMD instructions, we tested the algorithm optimized just with the SIMD instructions,

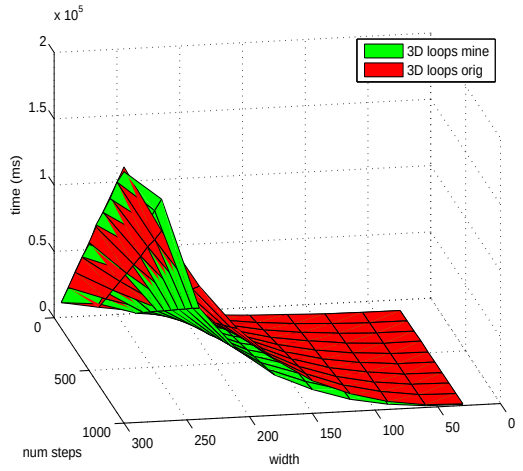

Figure 6: Runtimes for Stencil Computation Heat Diffusion Loops Algorithm with Optimized vs Original 3D Kernel

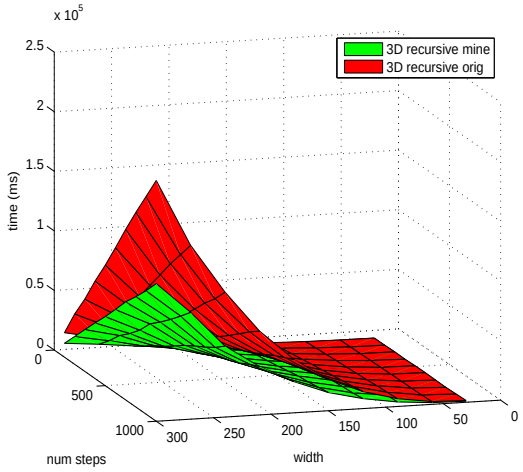

Figure 7: Runtimes for Stencil Computation Heat Diffusion Recursive Algorithm with Optimized vs Original 3D Kernel

```
if (x == 0 || x == Q->X-1 ||
    y == 0 || y == Q->Y-1) {
     U(Q, t+1,x,y) = IDENTITY;
}
else
     U(Q, t+1,x,y) =
     Q->CX * (U(Q, t,x+1,y)
        - 2.0 * U(Q, t,x,y) + U(Q, t,x-1,y))
     + Q->CY * (U(Q, t,x,y+1)
        - 2.0 * U(Q, t,x,y) + U(Q, t,x,y-1))
     + U(Q, t,x,y);
```

Figure 8: Original kernel yields poor performance.

but without the code generated by the N loop expansion script. Again, we ran a series of tests on this algorithm with 3D grid sizes ranging from $50 \times 50 \times 50$ to $300 \times 300 \times 300$ and number of time steps ranging from 100 to 1000. The results we obtained are summarized in Table 5.

Table 5: Runtimes for Fully Optimized vs SIMD Base Kernel vs Original Base Kernel for Recursive Implementation

	Original	SIMD	Optimized
$50 \times 50 \times 50$	65 ms	61 ms	47 ms
$300 \times 300 \times 300$	202124 ms	174695 ms	115489 ms
Average	34128 ms	29648 ms	19898 ms

These results show that without the N dimension code generating script, the algorithm achieves a 13% speedup with just the SIMD optimization. This indicates that a larger portion of the speedup observed in the fully optimized version is due to the metaprogramming that generates the N dimensional base kernel, and a smaller portion to the SIMD optimization.

7. BOUNDARY CONDITIONS

7.1 Boundary Condition Optimizations

To improve the performance of the kernel, we tried two approaches to optimizing boundary calculations. The original kernel shown in Figure 8 yields poor performance because each time it calculates a space point, it must do several calculations to determine if the point is on a boundary.

The first optimization we tried pre-computes the boundary conditions for each space point, storing the results in a bitmap. The bitmap stores one bit per space coordinate, where the bit is set to 0 if it is on the boundary and 1 otherwise. Extending the kernel to include the boundary bitmap is shown in Figure 9. This optimization saves computation on each call to the kernel.

Next, we tried applying the zero padding technique to further improve performance. For zero padding, the n-dimensional array is padded with extra space. Figure 10 shows how zero padding of width 1 can be applied to the 2-dimensional heat equation. The kernel is called from x_0 to x_1 in the x dimension and y_0 to y_1 in the y dimension. Since the kernel is never called on the padding, these space points remain fixed and the conditional in Figure 8 can now be removed entirely.

```
U(Q, t+1,x,y) = (B(x,y) ?
Q->CX * (U(Q, t,x+1,y)
   - 2.0 * U(Q, t,x,y) + U(Q, t,x-1,y))
+ Q->CY * (U(Q, t,x,y+1)
   - 2.0 * U(Q, t,x,y) + U(Q, t,x,y-1))
+ U(Q, t,x,y) :
IDENTITY;
```

Figure 9: Extending the kernel to support bitmap boundaries is simple - if the bitmap of the space coordinate is 1, the stencil is computed at that coordinate, otherwise it isn't.

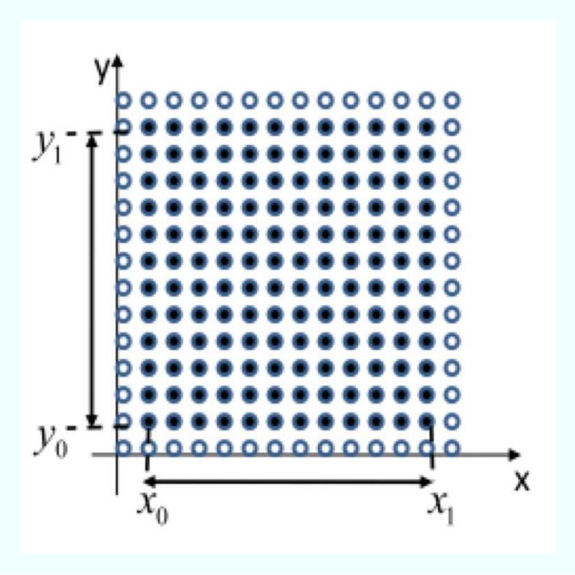

Figure 10: When Zero Padding is applied to the stencil computation, the kernel is called on points within the range x_0 to x_1 and y_0 to y_1.

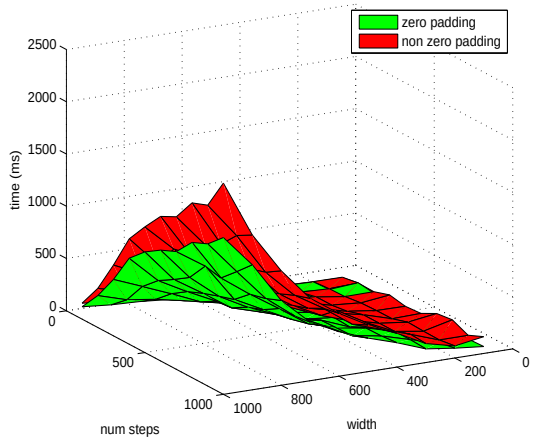

Figure 11: Speedup of zero padding on stencil computations.

Both the bitmap approach and zero padding improve performance. On average, the bitmap yielded gains of 17%, and the zero padding yielded gains of 25%. Figure 11 shows the speedup of applying the zero padding to grids of different sizes and time steps. On the smallest grid (100×100), the speedup is 6 ms vs 10 ms, and on the largest grid (1000×1000), the speedup is 315 ms vs 418 ms (25%).

7.2 Periodic versus non periodic boundary conditions

In non-periodic boundary stencils, we can utilize the zero-padding technique to greatly reduce the overhead accessing and updating of boundary element. In periodic boundary stencils, we adopt a strategy of divide and conquer. We separate the region into internal region versus boundary region and call different kernels for them. So the basic idea is to restrict the use of modulo operation for boundary regions only. Based on this region partition, we can further remove all the modulo operation by checking all elements in boundary region to see whether they are exactly on the boundary to set up their indexes directly by some conditional instructions, such as in Figure 12.

By adopting these optimizations on periodic boundaries, we save roughly 60% in performance.

Another trick in periodic boundary stencil is that we have to merge the beginning trapezoid and end trapezoid of every dimension into a big trapezoid to avoid asynchronous updates to the wrap-up elements.

8. STENCILS ON IRREGULARLY SHAPED GRIDS

Stencil computation may not always conform to a standard n-dimensional grid. It is conceivable that stencil sim-

```
idx = (idx == 0) ? size - 1 : ++idx;
idx = (idx == size-1) ? 0 : --idx;
```

Figure 12: conditional instructions to remove all modulo operation in boundary region

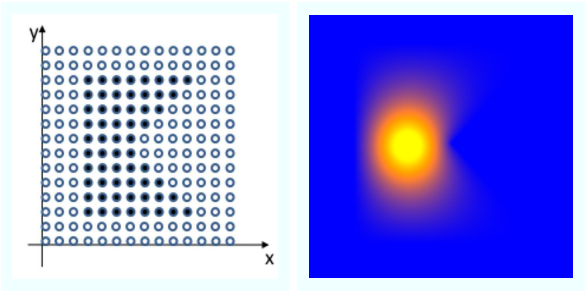

Figure 13: Bitmap for irregular boundary polygon defined by five points (left). Rendering of 2D heat equation on irregular polygon boundary (right).

ulations would require irregular boundaries. For example, when calculating diffusion of particles in a cell, an irregular boundary that matches a cell membrane would yield more realistic results than a standard grid. For this reason, our system has rudimentary support for irregular boundaries.

To support irregular boundaries, we allow users to define a bitmap, with one bit per space point. The bit for a space point is set to 1 if the point falls within the irregular boundary and 0 otherwise. The extension to the kernel to support irregular shape boundaries is the same as the bitmap boundary condition code shown in Figure 9.

Rather than set each bit manually, the user can call one of our helper functions: *polyBoundary* or *circBoundary*. *polyBoundary* allows the user to specify a polygon region by defining an array of vertices, and *circBoundary* allows the user to specify a spherical region by specifying a center and a radius. These functions pre-compute the bitmap by performing hit tests for each space point on the grid. Figure 13 shows what the bitmap (left) might look like for a sample polygon defined by 5 points. Points inside the boundary are solid; points outside are not. Also shown in Figure 13 is a time slice of a 2-dimensional heat equation simulation on a 5-point irregular boundary polygon.

Performance gains are also obtained by computing over irregular shaped domains, as compared to the rigid n-dimensional grid because only a subset of the space coordinates need to be computed at each time step. Figure 14 shows the speedup of applying the irregular shape from in Figure 13 to grids of different sizes and time steps. As the grids are scaled up, the irregular shape is also scaled up, relative to the grid size. The performance graph for irregular boundary conditions and regular boundary conditions are shown in green and red respectively. The irregular boundary condition wins in nearly every case. The average speedup over all of the trials is 234.6 ms vs 315.0 ms (26%). In the smallest grid (100×100), both are the same at 6 ms. In the largest grid (1000×1000), the speedup is 1026 ms vs 1500 ms (32%).

9. CONCLUSION

In this paper, we have presented many optimizations to improve the performance of stencil computations. In the 2-dimensional recursive algorithm, coarsening on space and time, we obtain a 16% performance gain compared to just coarsening on time. Also, by cutting a fixed number of times ($r = 4, 8$), we see performance gains of 12%. By applying zero padding to the 2-dimensional recursive algorithm, we

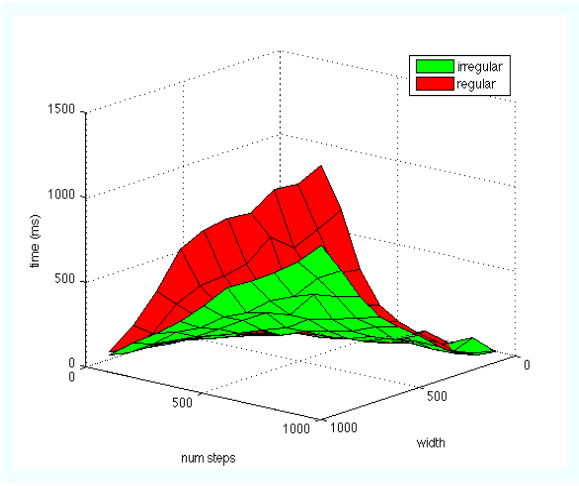

Figure 14: Speedup of irregular boundary conditions vs normal boundary conditions.

obtain performance gains of 25%. Adding SIMD instructions to the kernel yields 10% on the 2-d heat equation. Meta-programming the N-dimensional code with loop unrolling and SIMD instructions yields performance gains of 42%. Even without SIMD instructions, our recursive cache oblivious algorithm performs 2.6 times better than a trivial parallel loops implementation when we apply our best optimizations (see Figure 15).

In addition, we present many features that make our stencil transformer unique, such as support for arbitrary N-dimensions, irregular grids, and periodic vs non-periodic boundary conditions. In the future, we would like to further improve performance by refining the base case computation. For example, we may investigate expression templates, such as Blitz++, freepooma, etc. We would like to further investigate real world applications of stencil computation on irregular shaped grids to gain insights on how to optimize irregular boundary conditions. In addition, we need to apply our optimizations other stencils, such as Lattice Boltzmann.

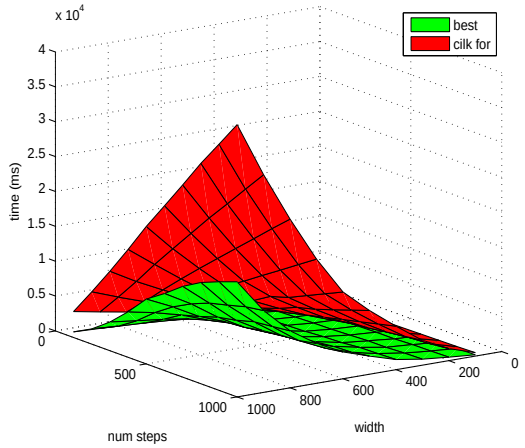

Figure 15: Speedup of our optimized recursive algorithm compared to trivial parallel loops.

Finally, we would like to define a user specification for how users will interact with our stencil transformer.

10. REFERENCES

[1] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[2] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[3] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

[4] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 271–280, New York, NY, USA, 2006. ACM.

[5] Matteo Frigo and Volker Strumpen. The memory behavior of cache oblivious stencil computations. *J. Supercomput.*, 39(2):93–112, 2007.

[6] Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM.