

Our first step to optimizing the program was to jump directly into SSE by using the example script already written in Bitbucket. The hope was that the SSE code was a fast enough kernel that we can build a good multiplier around it; indeed, it an immediate speed-up was noticed! It approximately doubled our naive multiplication. See figure 1.

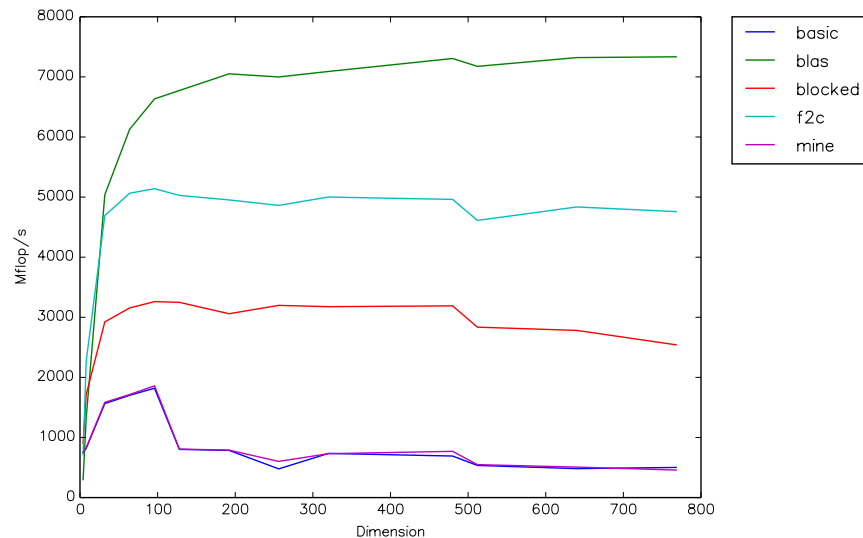


Figure 1: Our initial set-up using the SSE code provided. Note that the “blocked” line is actually our matrix multiply as it was a fairly early version of the code.

What the code does is fairly basic:

1. Initialize arrays At, Bt.
2. Takes in matrix A, B and copies them into At, Bt such that we have memory layout of 2 by P blocks in A and P by 2 blocks in B.
3. Loop over the blocks in A and B and multiply, adding the 2 by 2 matrix from the output into the C matrix.

We originally compiled this SSE code via the Intel compiler, but weirdly we found that compiling it with the GCC compiler results in a roughly 33% speedup over the Intel compiler! We’re actually not sure why, but various flags of the ICC doesn’t seem to coax the same performance out of the code.

This was all tested on even sizes initially, as there was no need to pad the matrices with anything. The odd cases are handled in the natural way, which was to pad a row and column and take the slight hit to the performance. When implemented, our performance was still pretty decent. See figure 2.

The next step to optimizing this was to play with the flags and use tiny code tricks.

## Tricks

One trick that we played with was to do some loop reordering. In our main block, we have the following

```
for (bi = 0; bi < n_blocks; ++bi) {
    const int i = bi * BLOCK_SIZE;
    for (bj = 0; bj < n_blocks; ++bj) {
        const int j = bj * BLOCK_SIZE;
        // trans is our ``transitional`` matrix which contains both At and Bt
        kdgemm2P2(n, tempC, &trans[bi*n*2], &trans[temp + bj*n*2]);
    }
}
```

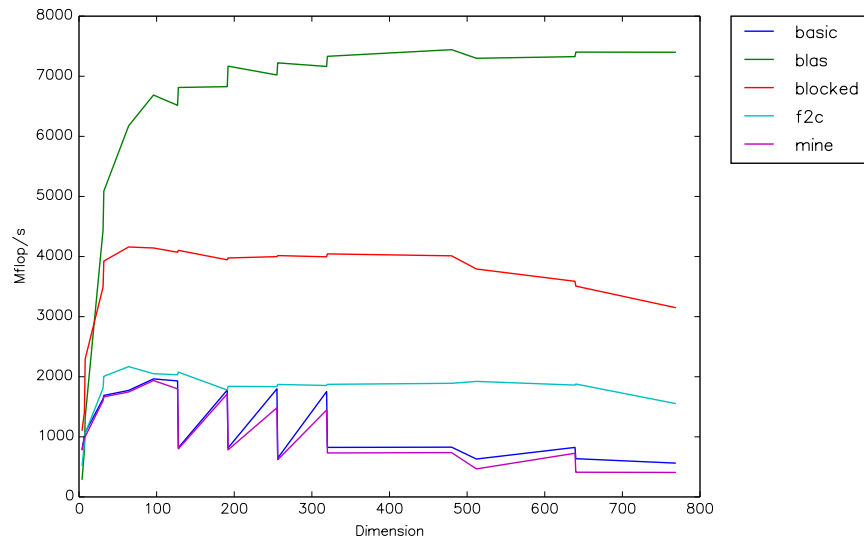


Figure 2: After adding the odd cases in and using GCC instead of icc(note the “blocked” line is ours once again).

Size	MFlops (flipped)	MFlops (as shown)
767	2660.82	3516.27
768	2348.95	3586.11
769	2229.79	3503.37

Table 1: Table of loop ordering

```

    }
}

```

We noted that flipping  $b_i$  and  $b_j$ 's loop order hurts the performance a lot. For example, at the far end of the spectrum of the sizes the different is quite noticable. See table 1, so we determined that Loop reordering was also applied to the copy commands:

```

if (odd == 0) {
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < m; i++) {
            At[i*n*2 + 2*j] = A[j*n + i*2];
            At[i*n*2 + 2*j + 1] = A[j*n + i*2 + 1];
        }
    }
}

```

while it wasn't obvious which order should be better as  $A_t$  increments via  $j$  via  $A$  increments via  $i$ , it turns out  $j, i$  ordering is *slightly* faster.

Another small speed increase laid with the serial performance of the copying. Initially, we actually declared to different arrays, one for  $A$  and one for  $B$ . We find that if we just create one giant array of size  $2n^2$ , and copy  $A, B$  into that giant array, performance increase slightly.

With all these small optimization and the flags (see section below), we achieve performance shown at figure 3.

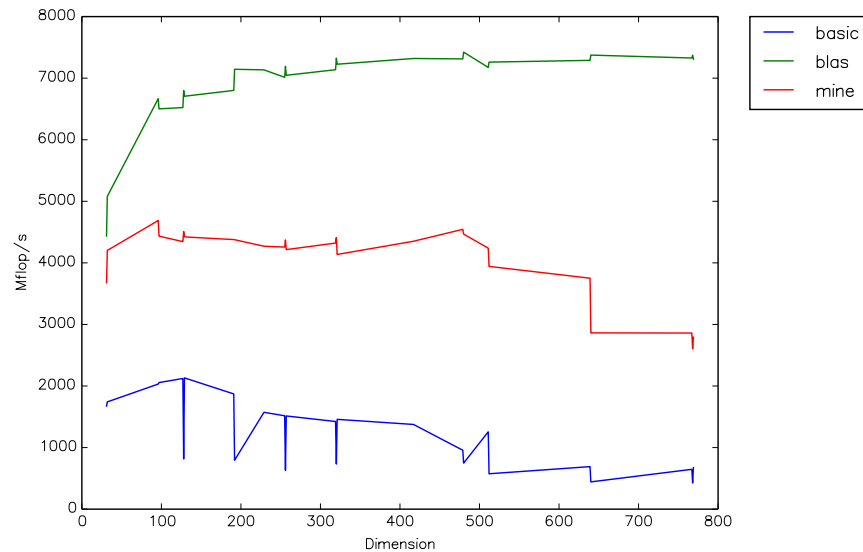


Figure 3: Final speedup

## Language Keywords

### Restrict

The **restrict** keyword is a type qualifier for pointers and is a part of C99 standard. It informs the compiler that the object designated by the pointer cannot be accessed by any other pointer. In other words, the same memory location cannot be pointed to by different pointers. This elimination of memory aliasing helps the compiler to do a better job at loop unrolling.

Since the pointers designating our matrices are non-aliasing, we can declare them as restricted pointers so the compiler can exploit this for automatic unrolling. Fig ??

only the pointer or a variable directly derived from it can access the memory address that it point to throughout its lifetime.

only the pointer or a variable directly derived from it can be

the object designated by the point can only be accessed by only the pointer or a value directly derived from it can be used to access the object designated by the pointer can only be accessed

only the pointer or a value directly derived from it will be used to access the associated memory address.

## Compiler Optimization Flags and Keywords

### Loop Optimizations

Since our manual loop unrolling analysis did not significantly improve the performance, we tried the automatic loop unrolling options offered by GCC. We tried the options mentioned in Table 2. Figure 4 shows the performance for the three cases. All the options improve the performance as they unroll loops that we had not done manually. *-funroll-loops* does a better job compared to the other two. Hence, we will consider it in combination with other flags in the next sections.

Option	Description
-ftree-loop-linear	Applies some linear transformations on loops (reversal, interchange, scaling, and skewing) that may improve cache utilization and simplify the loops to allow other optimizations to take place. In particular, loop interchange may change loop nesting for loops that walk through matrices.
-funroll-loops	One of the better known loop transformations. As the name indicates, if the compiler can determine how many iterations will the loop execute, and that number is some small constant $N$ , it emits $N$ copies of the loop body. This may produce faster code at the expense of increased size.
-ftree-loop-im	Removes loop invariant code from inside loops.

Table 2: Loop unrolling options that we tried.

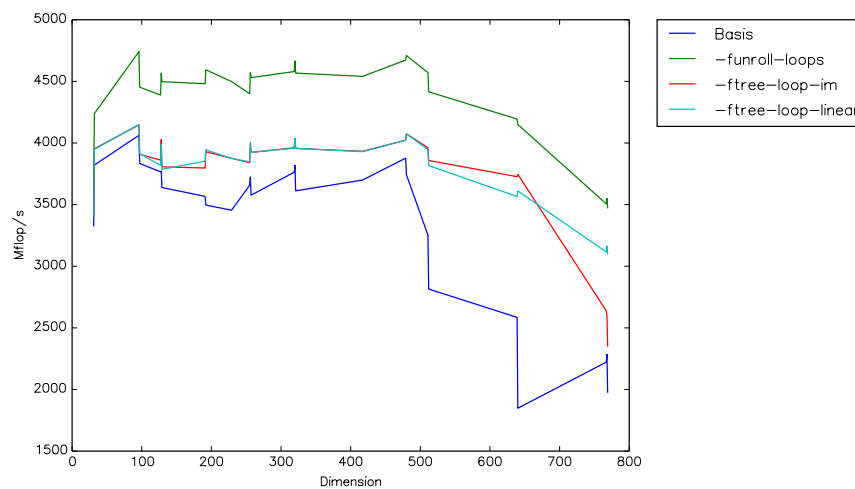


Figure 4: MFLOPS/s vs Matrix Size for loop optimization flags.

## Fast Floating Point Operations

The *-ffast-math* option includes most of GCC flags for fast floating point arithmetic optimization. This option is not always recommended though because it breaks strict IEEE compliance on floating-point operations which might result in accumulation of computational errors of unpredictable nature. Since the accuracy of our matrix multiplication routine is verified within the *matmul.c* function, we decided to give *-ffast-math* a try. As can be seen in Fig 5, this option significantly increases the performance, specially for large matrix sizes. Hence, we are going to consider it in combination with other flags in our final choice of flags.

## Memory Usage

One thing we tried was to break up the  $P$  by 2 and 2 by  $P$  matrices into smaller block. Theoretically, since the size of a double is 8 bytes, we need to fit  $2(2P) * 8 + 4 = 32P + 4$  bytes into the cache. While the level 1 cache of the educational nodes are quite low, it seems to not matter from the plots (our maximum size that we can fit in the L1 cache is  $P = 128$ ). If we look at the level 2 cache, our array size can go up to 1024 theoretically, which (theoretically) shouldn't impede us.

If we look at the cachegrind output for our program, we see the following:

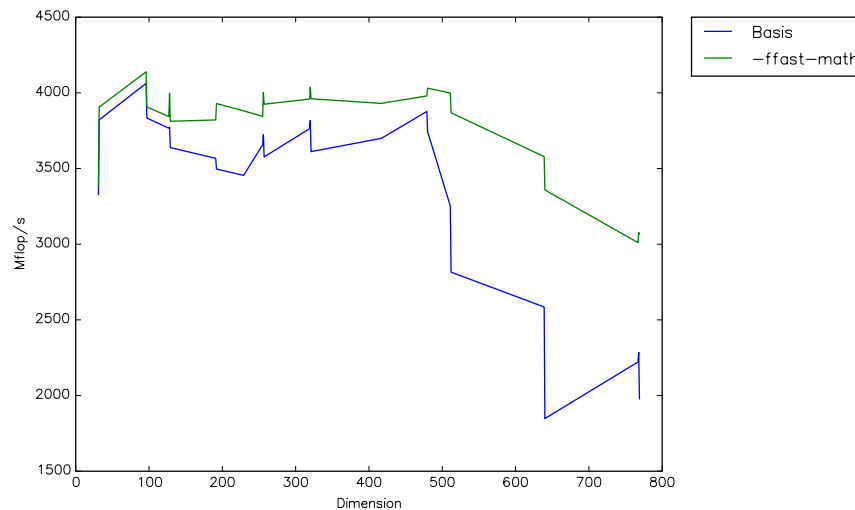


Figure 5: MFLOPS/s vs Matrix Size for *-ffast-math*.

```
[sj423@en-cluster02 matmul]$ valgrind --tool=cachegrind ./matmul-blocked
==18730== Cachegrind, a cache and branch-prediction profiler
==18730== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==18730== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==18730== Command: ./matmul-blocked
==18730==
--18730-- warning: L3 cache found, using its data for the LL simulation.
--18730-- warning: pretending that LL cache has associativity 24 instead of actual 16
Compiler: gcc
Options: -O3 -funroll-loops -msse4.2 -ffast-math -mtune=generic -march=corei7
Description: Simple blocked dgemm.

==18730==
==18730== I refs: 62,147,347,847
==18730== I1 misses: 1,539
==18730== L1i misses: 1,518
==18730== I1 miss rate: 0.00%
==18730== L1i miss rate: 0.00%
==18730==
==18730== D refs: 32,109,636,396 (27,962,135,435 rd + 4,147,500,961 wr)
==18730== D1 misses: 1,976,702,658 ( 1,966,158,637 rd + 10,544,021 wr)
==18730== L1d misses: 7,256,578 ( 5,016,816 rd + 2,239,762 wr)
==18730== D1 miss rate: 6.1% ( 7.0% + 0.2% )
==18730== L1d miss rate: 0.0% ( 0.0% + 0.0% )
==18730==
==18730== LL refs: 1,976,704,197 ( 1,966,160,176 rd + 10,544,021 wr)
==18730== LL misses: 7,258,096 ( 5,018,334 rd + 2,239,762 wr)
==18730== LL miss rate: 0.0% ( 0.0% + 0.0% )
```

which actually mimics the performance of the BLAS cache stats:

```
[sj423@en-cluster02 matmul]$ valgrind --tool=cachegrind ./matmul-blas
==1449== Cachegrind, a cache and branch-prediction profiler
```

```

==1449== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==1449== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==1449== Command: ./matmul-blas
==1449==
--1449-- warning: L3 cache found, using its data for the LL simulation.
--1449-- warning: pretending that LL cache has associativity 24 instead of actual 16
Compiler: gcc
Options: -O3 -funroll-loops -msse4.2 -ffast-math -mtune=generic -march=corei7
Description: System CBLAS dgemm.

==1449==
==1449== I refs:  59,157,389,231
==1449== I1 misses:      2,399
==1449== LLi misses:    2,297
==1449== I1 miss rate:   0.00%
==1449== LLi miss rate:  0.00%
==1449==
==1449== D refs:  30,847,547,276 (26,700,497,225 rd + 4,147,050,051 wr)
==1449== D1 misses: 1,670,642,244 ( 1,665,753,557 rd + 4,888,687 wr)
==1449== LLd misses: 3,023,479 ( 2,584,556 rd + 438,923 wr)
==1449== D1 miss rate:  5.4% (      6.2% +      0.1% )
==1449== LLd miss rate:  0.0% (      0.0% +      0.0% )
==1449==
==1449== LL refs:  1,670,644,643 ( 1,665,755,956 rd + 4,888,687 wr)
==1449== LL misses:  3,025,776 ( 2,586,853 rd + 438,923 wr)
==1449== LL miss rate:  0.0% (      0.0% +      0.0% )

```

Note that it's a similar miss rate in terms of percentages, but our algorithm had to use much more read/writes than the BLAS algorithm, which contributes to inefficiencies.

It's quite interesting to see that such small differences in the miss rates will result in such a large cut in performance from the BLAS package.

## New Kernel

While benchmarking matrix multiplication based on Bindel's kernel we also attempted to write a faster one, focusing on multiplying 8x8 matrices (saved in mm\_kernel/kdgemm\_doublerainbow.c). The approach was fairly straightforward: starting with a naive implementation we unrolled the 2 inner loops and replaced adjacent operations with their equivalent SSE instructions. Afterwards it became clear that replacing A and C with their transposes would make all memory accesses to each matrix sequential, improving cache use. This put the kernel on par with Bindel's. Several other optimizations were tried to no benefit. Unlike Bindel's

	Name	Size	MFlops
	Our (double rainbow)	8 by 8	5525.91
	Naive (simple)	4 by 4	2964.03

Table 3: Kernel Timing Results

kernel the basic 8x8 SSE is faster with the Intel compiler than gcc. After reading some documentation on icc we found a flag called 'fast' that makes architecture-specific optimizations and added the equivalent flags for the cluster nodes, but this didn't help. There was no gain from adding more temporary variables to eliminate false dependencies past the 8 in the final version. We also tried reducing the matrix size to 4x4 which actually resulted in a performance decrease.

We eventually did luck into another performance boost. Using Bindel's kernel we had experimented with the `-funroll-loops` flag and assumed it would do a near optimal job, and thought this would be especially true on a small matrix with size known at compile time. This turned out to be wrong. After adding `'#pragma unroll(8)'` above the main loop on a whim the kernel jumped to 5500Mflops (see table 3).

The new kernel was not ready in time to integrate with the main dgemm routine. It is available only in `mm_kernel/kdgemm_doublerainbow.c`. It is likely some performance would be lost making it work with larger matrices.