1 The initial version

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, an immediate speed-up was noticed! It approximately doubled our naive multiplication. See figure 1. It should be noted that we had to add the *-msse4.2* flag to the Makefile in order to enable GCC to understand the SEE instructions. This was in addition to the *-O3* flag which was already on. We are going to discuss this further in Section 3.

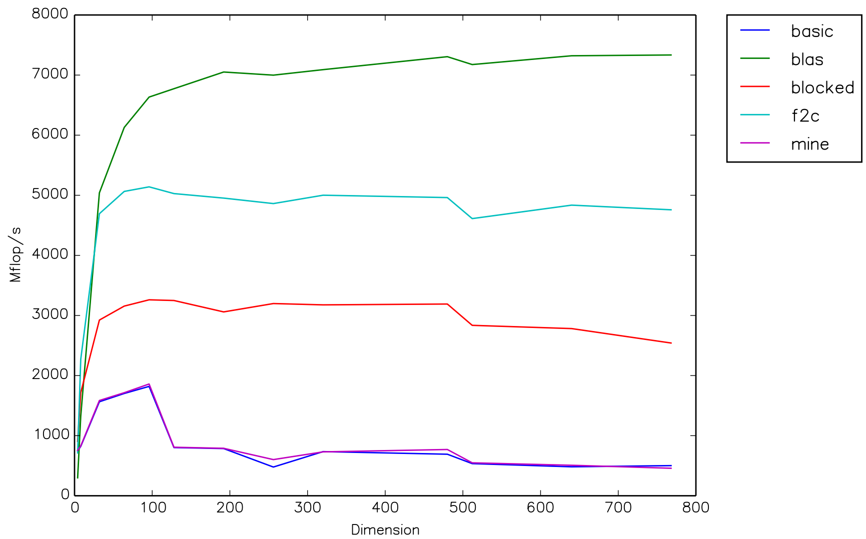

Figure 1: Our initial set-up using the SSE code provided. Note that the "blocked" line is actually our matrix multiply as it was a fairly early version of the code.

What the code does is fairly basic:

1. Intialize 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 intially, 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.

2 Loop Reordering and Copy Optimization

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

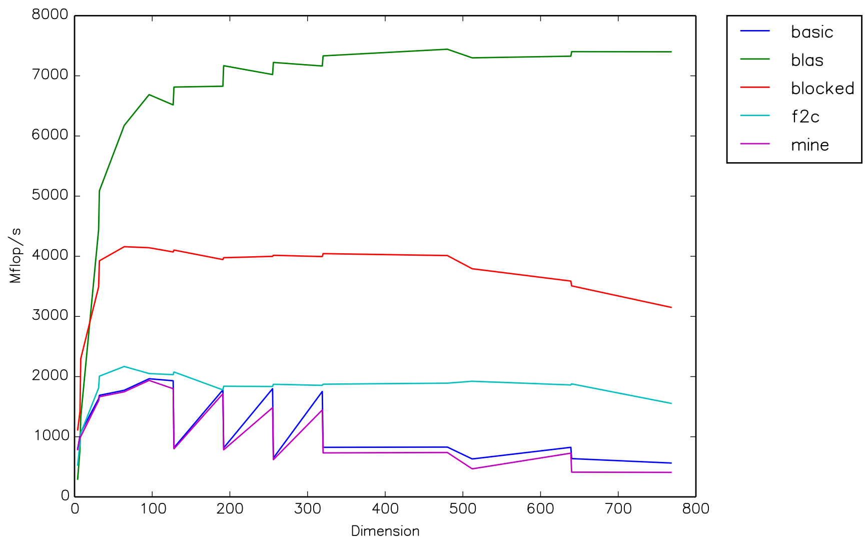

Figure 2: After adding the odd cases in and using GCC instead of icc(note the "blocked" line is ours once again).

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

```
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]);
  }
}
```

We noted that flipping bi and bj's loop order hurts the performance a lot. For example, at the far end of the spectrum of the sizes the difference is quite noticeable. 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 At increments via j and 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 two 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, the performance increases slightly.

Henceforth, we are going to call the version with all these small optimizations and the *O3* and *msse4.2* flags as the Basis configuration. The performance of this version can be seen in figure 3. The next step is to further optimize this version by working on compiler optimization flags.

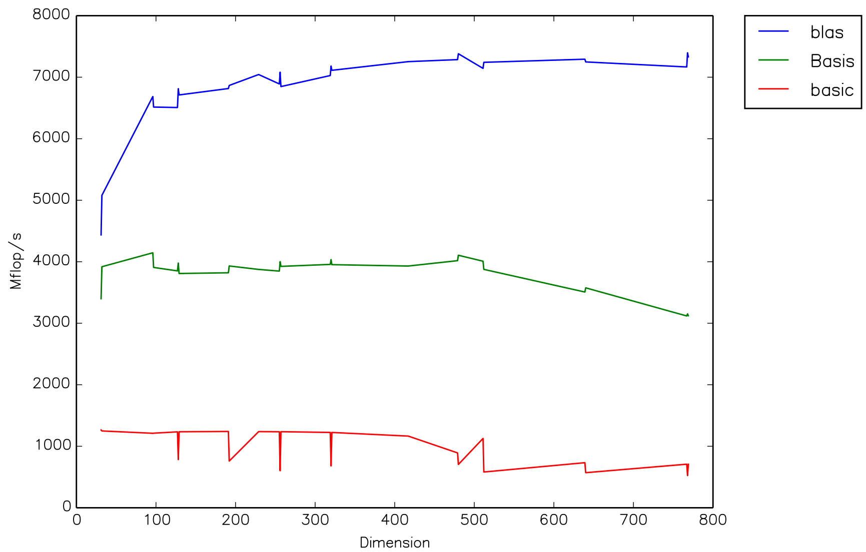

Figure 3: The Basis configuration before further compilation flag optimization.

3 Compiler Optimization Flags and Keywords

As mentioned in the previous sections, we start our analysis with the Basis configuration which already has the *O3* and *msse4.2* flags on. The *msse4.2* flag enables GCC to understand SSE instructions which is necessary as our kernel is an SSE code. As for the *O3* flag, we performed our analyses on compiler optimization flags for the three cases of *O1*, *O2* and *O3*. However, due to lack of time, we provide the results only for the latter case as it led to better performance.

3.1 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 each and all combinations of the options mentioned in Table 2.

Figure 4 shows a comparison between the performance of the Basis configuration and that of each loop optimization flag. *-ftree-loop-im* and *-ftree-loop-linear* do not improve the performance in our case. This was expected for *-ftree-loop-im* as our code does not have loop invariant code inside loops. Using the *-funroll-loops* results in a significant performance improvement because it performs automatic unrolling for some loops that we did not have a chance to unroll manually.

Figure 5 shows a comparison between the *-funroll-loops* flag and all combinations of the loop optimization flags. The combination of *-ftree-loop-im* and *-ftree-loop-linear* more or less has the same performance as that

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.

of the Basis configuration. All the other combinations improve the performance significantly and their performances are very close to each other. What is interesting is that *-funroll-loops* is part of all highly performed combinations and its performance by itself is more or less on par with those combinations. This probably means that most of the performance boost is thanks to *-funroll-loops* and other flags have marginal contributions.

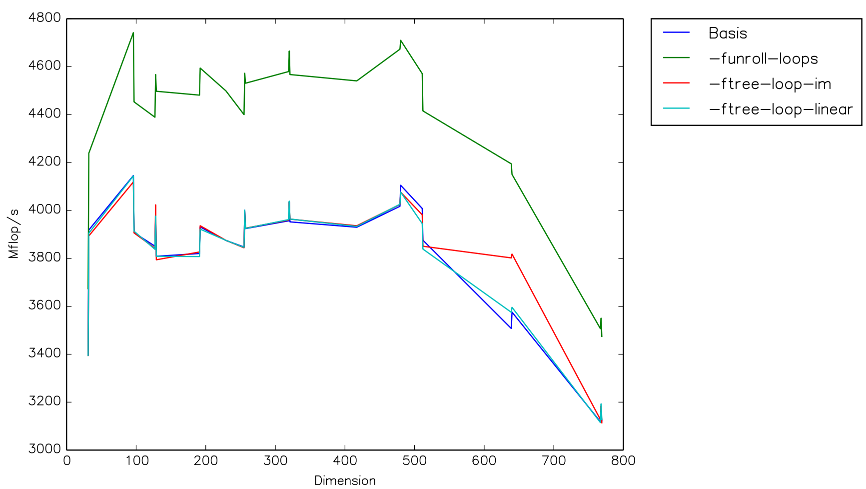

Figure 4: MFLOPS/s vs Matrix Size: Comparing the basis configuration with each of the loop optimization flags.

3.2 Restrict Keyword

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. Hence, we tried the restrict keyboard combined with the two loop unrolling options considered in the previous section, i.e. *-funroll-loops* and *-ftree-loops-linear*.

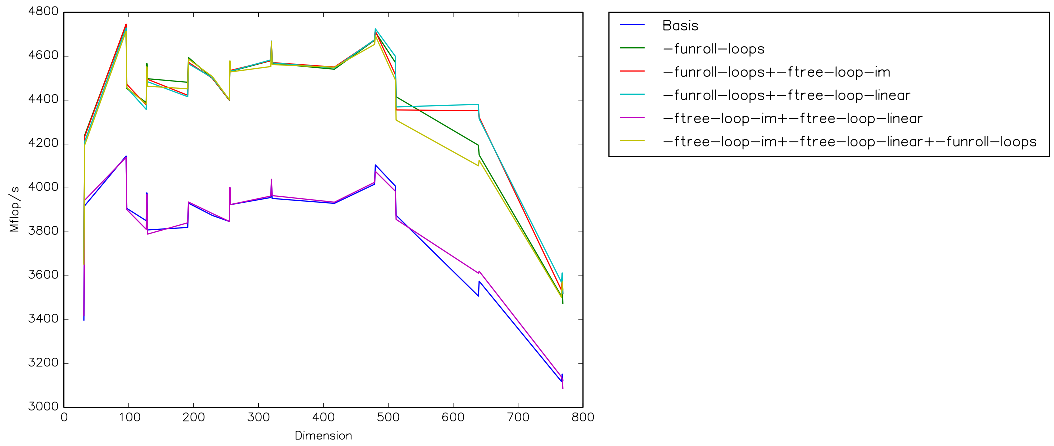

Figure 5: MFLOPS/s vs Matrix Size: Comparing different combinations of loop optimization flags with the Basis configuration and *-funroll-loops* by itself.

You can see the implementation in $dgemm_mine_restrict.c$.

The outcome can be Fig 6. It turns out that using this keyword does not improve the performance and at most reproduces that of *-funroll-loops* by itself. Hence, we will not consider this flag in our analysis for finding the optimal combination of optimization flags.

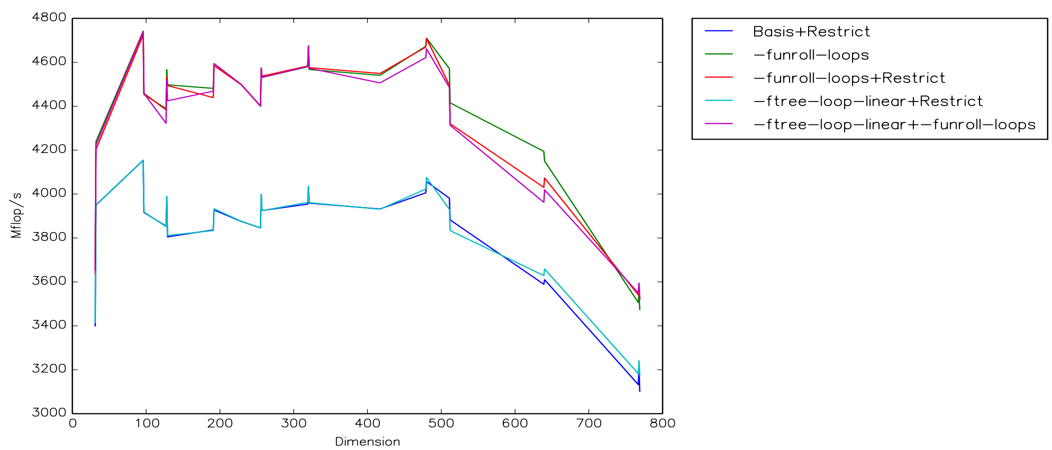

Figure 6: MFLOPS/s vs Matrix Size: Different combinations of loop unrolling flags with the restrict keyword.

3.3 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 7, this option does not affect the performance significantly and indeed it degrades the performance for some cell sizes. Hence, we will not consider this flag in our analysis for finding the optimal combination of optimization flags.

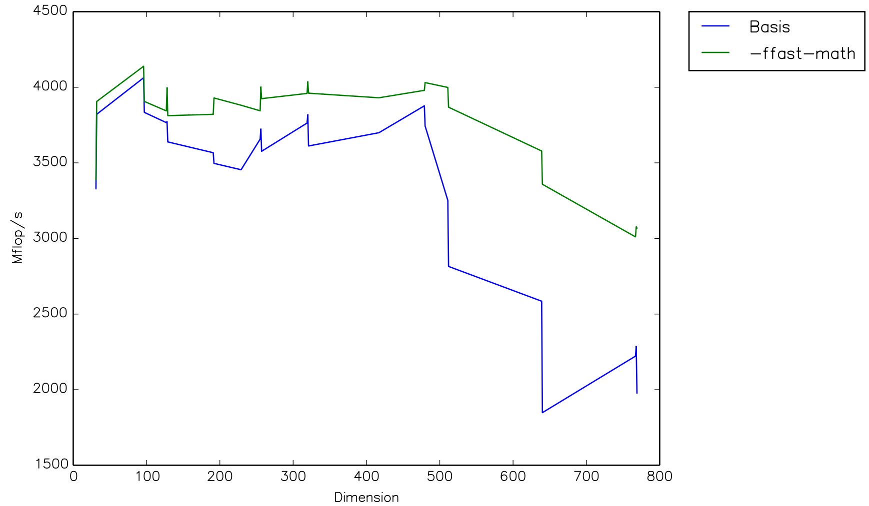

Figure 7: MFLOPS/s vs Matrix Size for *-ffast-math*.

3.4 Architecture Specific Flags

Different CPUs have different capabilities, support different instruction sets, and have different ways of executing code. *-march* and *-mtune* are the most common GCC flags for this purpose. On *x86* and x86-64 CPUs, which includes the Xeon E5504 cores on our instructional nodes, *-march* will generate code specifically for that CPU using all its available instruction sets and the correct ABI. Also, including *-march* implies *-mtune*. So, unless we are not aware of the processor family we are using, having *-march* must be enough.

We have GCC 4.8.2 on the cluster. The relevant *-march* options on this version that are relevant to our processor are described in Table 3. Our processor, Xeon E5504, has the Nehalem-EP microarchitecture. In fact, the first Nehalem-EP processor that came out was Core i7. There are core i7 processors that are based on this architecture and are similar to ours. That is why *corei7* seems to be the most relevant option. However, GCC is not that consistent about how it treats architecture specific flags. That is why we also considered the *core2* and *native* options. The reason for *core2* is that Nehalem-EP is a successor of the Core microstructure and we considered *native* because it lets the compiler to determine the architecture by itself.

Figure 8 shows the performance obtained by each case compared to the basis configuration.The three options do not significantly affect the performance, although *native* seems to perform better for larger sizes. This is probably because our kernel is already an SSE code and hence is not affected much by architecture specific flags.

Although the architecture specific flags do not seem to improve the performance by themselves, they might do so in combination with other flags as they might change the way the compiler use certain flags. Hence, we will reconsider these flags in our analysis for finding the optimal combination of optimization flags.

3.5 Combining Flags and Picking the Optimal Configuration

We start our analysis by considering all combinations of those flags that either improved the performance in our observations or could potentially do so by being combined with other flags. For the loop optimization flags, we chose the 4 combinations that involved *-funroll-loops* in them as they all improved the performance at a similar level, c.f. Fig 5. As for architecture specific flags, we only considered *-march = native* because, as shown in Fig 8, the three flags result in more or less the same performance. Therefore, the combinations we considered were the following:

Option	Description
Native	This selects the CPU to tune for at compilation time by determining the processor type of the compiling machine. Using -mtune=native will produce code optimized for the local machine under the constraints of the selected instruction set. Using -march=native will enable all instruction subsets supported by the local machine (hence the result might not run on different machines).
corei7	Intel Core i7 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2 instruction set support.
core2	Intel Core 2 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support.

Table 3: Loop unrolling options that we tried.

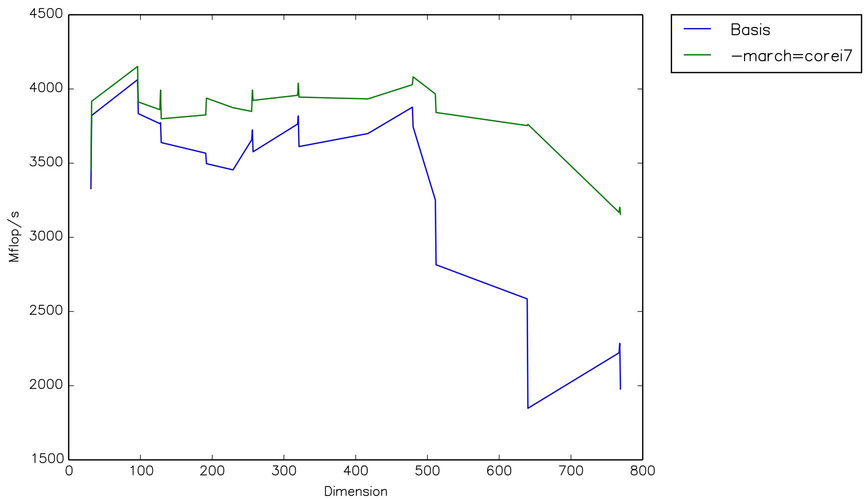

Figure 8: MFLOPS/s vs Matrix Size for the architecture specific flags.

1. OPTFLAGS = -O3 -msse4.2 -ftree-loop-linear -funroll-loops -ftree-loop-im -march=native
2. OPTFLAGS = -O3 -msse4.2 -ftree-loop-linear -funroll-loops -march=native
3. OPTFLAGS = -O3 -msse4.2 -ftree-loop-im -funroll-loops -march=native
4. OPTFLAGS = -O3 -msse4.2 -funroll-loops -march=native
5. OPTFLAGS = -O3 -msse4.2 -funroll-loops

Note that *O3* and *-msse4.2* constitute our Basis configuration which all the other include. The reason for *O3* is that our observations showed that, when used with the above flags, it results in better performance than that obtained by *O1* and *O2*. We used *-msse4.2* in our Basis configuration because it is the fastest SSE flag for our architecture.

Fig 9 shows the the outcome for all of the above combinations. The 3rd, 4th and 5th combinations clearly do a better job. Their outcomes are shown in Fig 10. It can be seen that the 3rd and 4th combinations perform slightly better and have more or less the same performance. The better performance of these combinations compared to the 5th one is thanks to the architecture specific flag *-march = native* because, as mentioned in the Loop Optimization section, the *-ftree-loops-im* flag has a significantly smaller contribution compared to *-funroll-loops*. Hence, we pick the 4th combination namely

- OPTFLAGS = -O3 -msse4.2 -funroll-loops -march=native

as our optimal configuration of compiler optimization flags. Fig 11 shows the performance of our code in comparison with those of Blas and the native code under the same configuration.

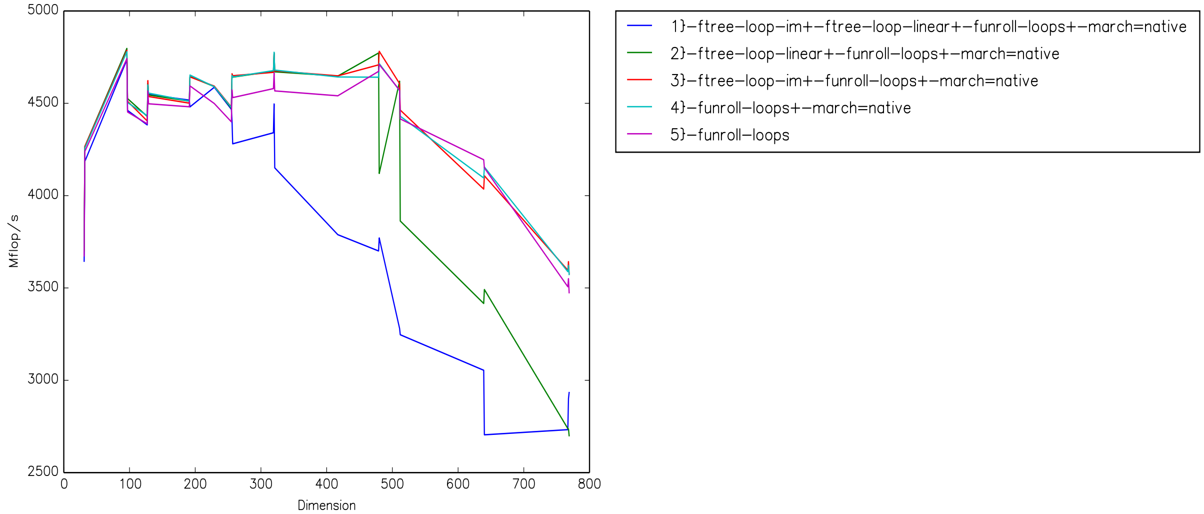

Figure 9: MFLOPS/s vs Matrix Size: combinations considered for finding the optimal configuration of the compiler optimization flags.

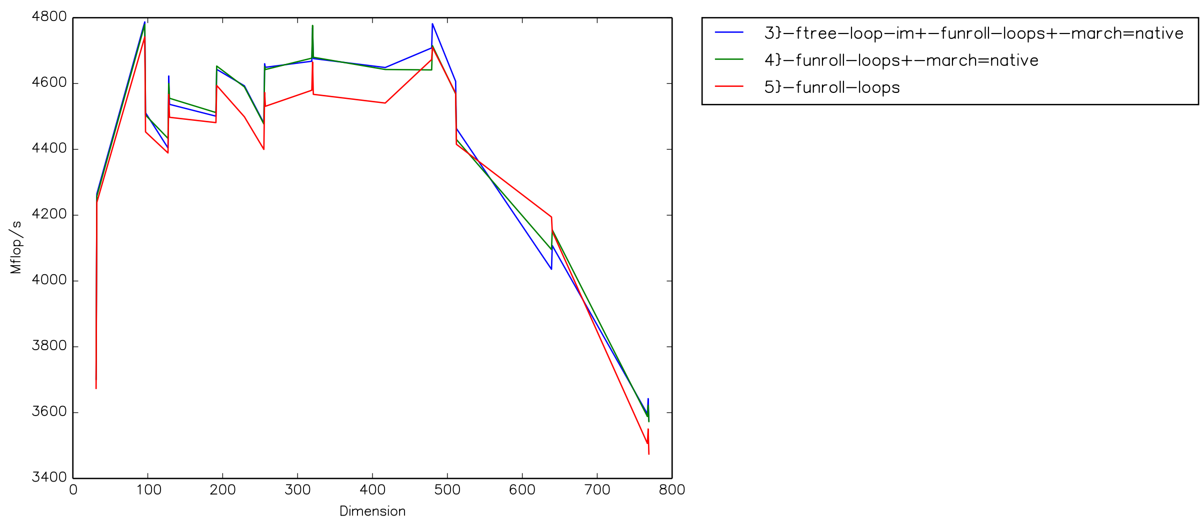

Figure 10: MFLOPS/s vs Matrix Size: the three combinations that performed better in our analysis on the compiler optimization flags.

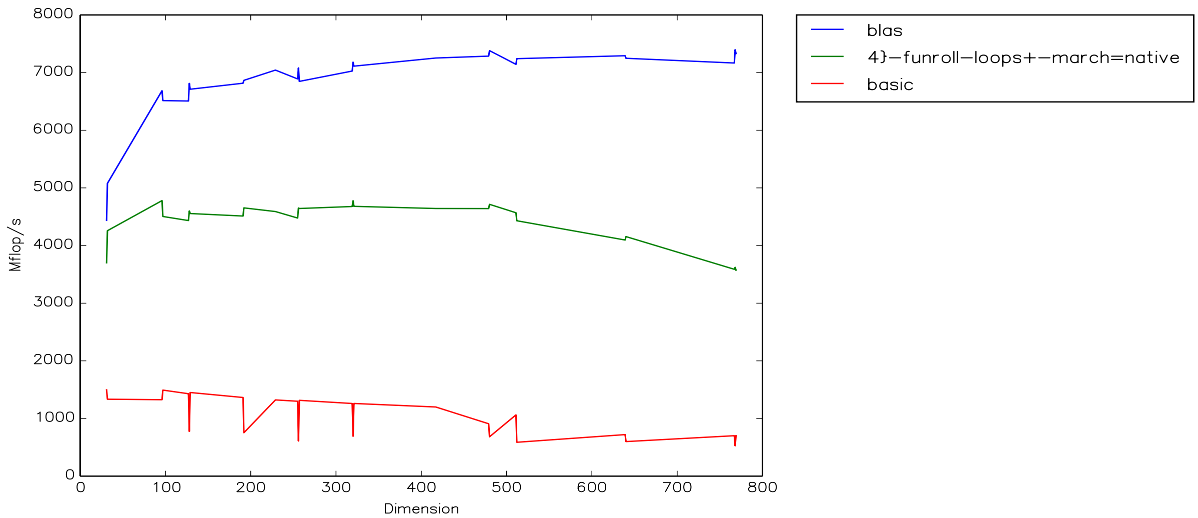

Figure 11: MFLOPS/s vs Matrix Size: the final optimized code vs Blas and the naive code.

4 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:

```
[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== LLi misses:     1,518
==18730== I1 miss rate:    0.00%
==18730== LLi 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== LLd misses:  7,256,578 (  5,016,816 rd +  2,239,762 wr)
==18730== D1 miss rate:     6.1% (       7.0%   +        0.2% )
==18730== LLd 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.

5 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 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

Name	Size	MFlops
Our (double rainbow)	8 by 8 by 8	5525.91
Bindel's 2P2 (gcc)	2 by 5000 by 2	4885.36
Our (double rainbow 4 x 4)	4 by 4 by 4	4178.78
Bindel's 2P2 (icc)	2 by 5000 by 2	3644.69
Naive (simple)	4 by 4 by 4	2964.03

Table 4: Kernel Timing Results (Note: Bindel's kernel seems to do slightly better for larger P values).

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 4). The use of icc compiler actually was better for our kernel, compared to Dr. Bindel's.

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. One thing to note though was that using Dr. Bindel's kernel, we achieved roughly 4 Gflops of performance, while the simplest case in ktimer allowed 4.8 Gflops. One can safely hypothesize that with an 8 by 8 kernel, we would get about 5 Gflops in performance for the best cases, and slightly lower for odd cases.