CS 5220: Homework 1 Writeup

Phillip Tischler (pmt43) and Joseph Vokt (jpv52)

02/16/2013

1 Description of Optimizations

General Approach. Our team's approach was to separately code and optimize a high performance kernel and matrix multiply subroutine. The kernel was implemented using SSE instructions, and a python script was used to tune the compile-time parameters for the kernel. Many approaches were implemented to achieve high performance matrix-multiply, but only one achieve performance better than that of simple blocked matrix multiply. The team attempted to implement the matrix-multiply as shown in the Goto paper [4]. This was then tuned using a python script similar to that used with the kernel. Finally, the team tested both compilers (GCC and Intel), experimented with various compiler flags, used profiling to identify hotspots, and compiled using profile guided optimization.

SSE Instructions & Kernel. The SSE code given by the professor was integrated into the matrix multiply kernel that is located in `kdgemm`. [3] The kernel performs a $[2x2] = [2xP][Px2]$ matrix multiply for a compile time parameter P. The value of P was selected using an auto-tuning process. This kernel was tested with the `ktimer` program and consistently achieved $5500Mflops$. Modifications were made to the copying of matrices A, B, and C to handle padding with zeros for differently sized inputs, and to handle re-ordering of elements in C for the SSE kernel. Additionally, the team experimented with different unrolling of the main SSE loop and found that unrolling by 8 achieved the highest performance.

Auto-Tuning Kernel. The kernel matrix performance is dependent on the compile time parameter P. The team modified the Makefile script to be able to define macros in the `kdgemm` code, and then used a python script to execute the build process with various values of P. This python script also executed the `ktimer` code and collected the output into a single file.

Iteratively Blocked Matrix Multiply with Single Copy Optimization. The first approach the team attempted for matrix multiply was an interative blocking approach. The concept was to perform blocking matrix multiply, and to process a set of blocks you would perform blocking matrix multiply on the blocks with a smaller block size. The idea was to perform the blocking at a size that optimized use of the different levels of cache. The team implemented a single copy optimization into an "L3" block, and then had a series of nested for loops which processed the blocks in sizes of "L2", "L1", and then the kernel. The final step was to copy from the "L1" blocks to the kernel memory and execute the SSE kernel. The team only achieved about $2000Mflops$ with this approach and subsequently abandoned it due to lack of performance. The team believes this approach was not as efficient because there was only one copy optimization, and the extreme size of the code base made achieving correctness and subsequently tuning very difficult. We then attempted a recursive strategy to get multi-level copy optimization and to drastically simplify the code base.

Recursively Blocked Matrix Multiply with Multi-Copy Optimization. The second approach the team attempted was to perform the same iterative blocking as before, but instead to restructure the code to downsize the code, make it easier to implement and optimize, and to perform multi-level copy optimizations. The team also abandoned this approach because performance dropped to about $1000Mflops$. The team believes this approach was not effective because the copy optimizations were very slow and did not actually work to improve cache use. After reading the Goto paper, the team believes that the extra copies were actually worse because it did not help align the memory segments and increase cache utilization. The team then attempted to adopt the approach described in the Goto paper.

Compilers and Compiler Flags. The team tested the two previous matrix-multiply kernels against both compilers and a series of compiler flags. This included different levels of optimization

(`O3, Ofast`, etc.), activating machine dependent optimizations (`march=native, -msse4, etc.`), and loop unrolling. The team also experimented with manually defining loop unrolling with pragma statements. The team found that the Intel compiler provided improvement over GCC, and the flags `O3, march=native, msse4,` and `ipo` had performance benefits.

Code Profiling to Identify Improvement Opportunities and Hotspots. The team use profiling to identify performance hot-spots and opportunities for improvement. To do this the team added the `pg` flag, compiled the code, executed the code, and then used the `gprof` command to view which functions and code execution branches dominated the matrix multiply computation. This was helpful in guiding the work of optimization.

Profile Guided Compilation. The team used profile guided compilation to optimize the way in which the compiler optimized the code. The team added the `prof-gen` flag to compile the program with profiling. The team then executed the code multiple times on the target platform to generate `dyn` files which contained runtime performance information. The `prof-gen` flag was then changed to `prof-use`, which then caused the compiler to use the profiles to guide the compiler's optimization. The team noticed a 2% improvement in kernel runtime and up to 10% improvement in matrix-multiply, depending on the approach used.

Goto Paper Algorithm and Implementation. The team decided that the previous implementations of matrix multiply were insufficient as they achieved only $1000 - 2000 Mflops$ with a kernel that can achieve $5500 Mflops$. To improve this, the team used the Goto paper which had implementation details for basic OpenBLAS subroutines. [4] The code was first implemented in Matlab to demonstrate correctness, and it was then ported to C. Figure 8 on page 11 of the paper shows the basic iterative blocking algorithm which was used. At a high level, this algorithm partitions A into block columns and B into block rows, each of which are partitioned into block rows and block columns respectively, until blocks which fit into the SSE instruction kernel are obtained. The paper suggested using a level 2 block size which fits into the L2 cache, and a level 3 block size which fits into the L1 cache, and we utilized this as a basis for our auto-tuning.

Auto-Tuning Matrix Multiply. The final optimization used was a python script to auto-tune the matrix multiply. This script was similar to that of the kernel optimization script in that it varied compile time constants, executed the code, and compared the runtimes. This script also executed the evaluations in a distributed fashion, using the `csub` command, so that more constants could be evaluated. The script then combined the outputs into a file sorted by performance.

2 Results & Any Odd Behaviors

Kernel Performance. Table 1 shows the kernel performance with different values of compile time parameter P. At $P = 240$, the team achieved a sustained performance of about $5600 Mflops$. The performance continued to increase until 240 because more of the L1 cache was being utilized and data was being accessed more sequentially. However, with values much larger than 240 the performance decreased, likely due to cache misses.

Matrix Multiply Performance. Table 2 shows the matrix multiply performance for different variables. The Goto paper had compile time parameters A and C, which were varied to achieve the highest performance. Figure 1 shows the comparison to the basic to our approach. In the lower matrix sizes our approach is slower, but for larger matrix sizes our approach is faster. This is likely due to the copy optimizations having greater affect. The performance of the matrix-multiply did not reach that of the kernel for many reasons. First, the `ktimer` code does manipulate memory with copy operations, but rather executes the kernel on the optimal amount of aligned memory that fits in the cache. To achieve equivalent performance the team would have to fully mask the overhead of the copy operations by fully utilizing the cache and prefetching the data optimally before it is needed.

P	**Mflops**	**Error**
16	4900.21	9e-16
32	4739.07	4e-15
48	4948.6	4e-15
64	5423.92	4e-15
80	5469.74	1e-14
96	5501.38	2e-14
112	5523.93	2e-14
128	5541.51	2e-14
144	5553.95	2e-14
160	5565.94	2e-14
176	5574.84	2e-14
192	5581.05	5e-14
208	5588.28	3e-14
224	5593.45	3e-14
240	5597.83	4e-14

Table 1: Kernel performance as reported by `ktimer`. The team varied the compile time parameter P, and achieved sustained performance with $P = 240$ of about $5600 Mflops$. The profiling was executed on the instructional nodes, instead of the head node, to reduce variability.

A	**C**	**M**	**Mflops**
256	48	767	1863.58
256	48	768	1859.85
256	48	511	1853.95
256	48	512	1845.52
256	48	256	1844.23
256	48	255	1834.73
256	48	480	1766.4
256	48	479	1756.39
256	48	229	1689.28
256	40	768	1635.09
256	40	512	1633.76
240	48	480	1633.55
256	48	767	1633.08
256	40	767	1632.34
256	48	256	1631.53
256	48	512	1631.31
256	40	511	1629.49
256	40	256	1627.04
256	48	511	1626.45
240	48	479	1624.31
192	40	768	1621.96
256	64	256	1618.18
192	40	767	1616.12
192	40	192	1615.72

Table 2: Matrix multiply performance for various values of A, C, and M. For certain values the team was able to achieve a peak of $1860 Mfops$ on the instructional nodes and $2600 Mflops$ on the head node.

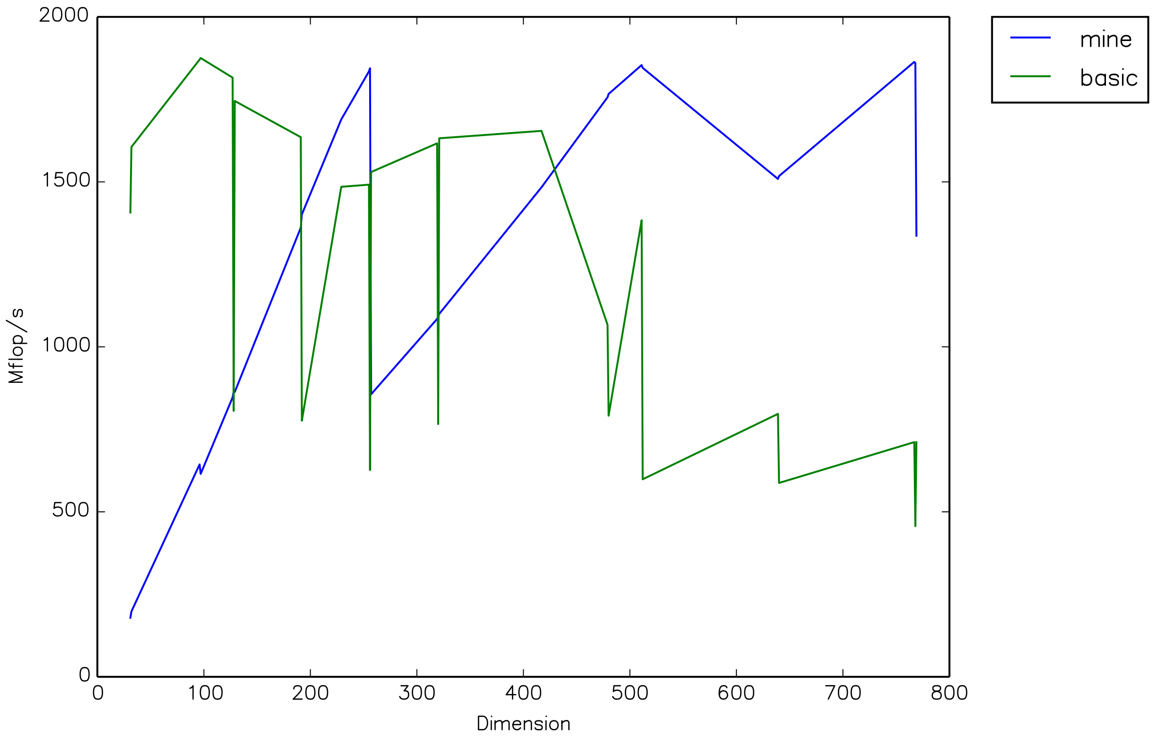

Figure 1: Performance comparison of our matrix multiply implementation vs. that of Basic.

References

[1] CS 5220 Homework 1 Webpage. `https:\bitbucket.org/dbindel/cs5220-s14/wiki/HW1`

[2] CS 5220 Serial Tuning Guide. `http:\www.cs.cornell.edu/~bindel/class/cs5220-f11/notes/serial-tuning.pdf`

[3] CS 5220 SSE Programming Webpage. `https:\bitbucket.org/dbindel/cs5220-s14/wiki/sse`

[4] Anatomy of High-Performance Matrix Multiplication by Kazushige Goto and Robert Van De Geijn. `http:\www.cs.utexas.edu/users/pingali/CS378/2008sp/papers/gotoPaper.pdf`