Compilers, Hands-Off My Hands-On Optimizations

Richard Veras Doru Thom Popovici Tze Meng Low
Franz Franchetti
Department of Electrical and Computer Engineering
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15215
{rveras, dpopovic, lowt, franzf}@cmu.edu

ABSTRACT

Achieving high performance for compute bounded numerical kernels typically requires an expert to hand select an appropriate set of Single-instruction multiple-data (SIMD) instructions, then statically scheduling them in order to hide their latency while avoiding register spilling in the process. Unfortunately, this level of control over the code forces the expert to trade programming abstraction for performance which is why many performance critical kernels are written in assembly language. An alternative is to either resort to auto-vectorization (see Figure 1) or to use intrinsic functions, both features offered by compilers. However, in both scenarios the expert loses control over which instructions are selected, which optimizations are applied to the code and moreover how the instructions are scheduled for a target architecture. Ideally, the expert would need assembly-like control over their SIMD instructions beyond what intrinsics provide while maintaining a C-level abstraction for the non-performance critical parts.

In this paper, we bridge the gap between performance and abstraction for SIMD instructions through the use of custom macro intrinsics that provide the programmer control over the instruction selection, and scheduling, while leveraging the compiler to manage the registers. This provides the best of both assembly and vector intrinsics programming so that a programmer can obtain high performance implementations within the C programming language.

Keywords

SIMD, Performance Engineering

Acknowledgment

This work was sponsored by the DARPA PERFECT program under agreement HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government. No official endorsement should be inferred.

WPMVP '16, March 13 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2870650.2870654

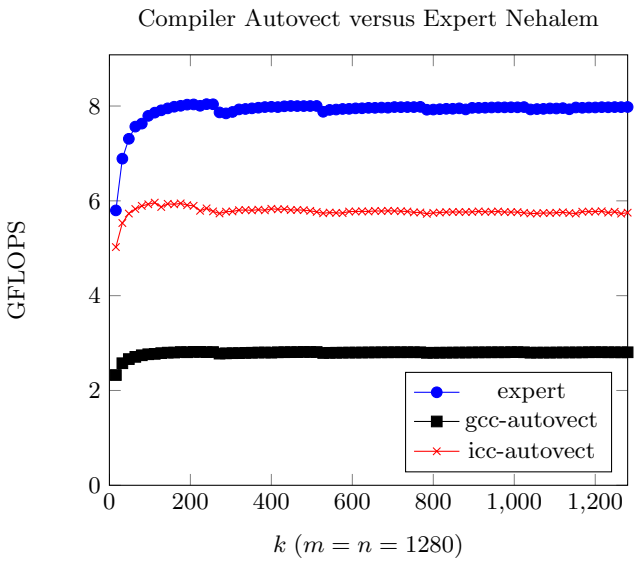

Figure 1: In this plot we compare the performance of an expert implementation of matrix-multiply against two compiler auto-vectorized implementations. The peak performance of the machine is 9.08 GFLOPS and the expert implementation reaches near that performance, while the two compiler vectorized implementations fall well below that.

1. INTRODUCTION

Implementing high performance mathematical kernels such as the matrix-matrix multiplication kernel is an extremely difficult task because it requires the precise orchestration of CPU resources via carefully scheduled instructions. This typically requires using single instruction multiple data (SIMD) units on modern out-of-order processors, such as Streaming SIMD Instructions (SSE) and Advanced Vector Instructions (AVX) [6] on Intel processors, AltiVec [2] on PowerPC processors, or NEON [1] on ARM processors.

For key kernels, this task is often undertaken by expert programmers who are knowledgeable about the application, available SIMD instruction set, and hardware architecture. This is because the programmer must manually select the appropriate instruction for their implementation, then they must schedule the instructions and finally orchestrate the movement of data to efficiently use the SIMD units. There-

fore, many high performance libraries rely on assembly coded kernels, mainly because of the high level of control offered by such languages. For this reason, library instantiations of the Basic Linear Algebra Subprograms (BLAS) [8, 4, 3] such as GotoBLAS [5] (Now OpenBLAS [9]), BLIS [10] and ATLAS [11] make use of routines written in assembly to implement operations like vector additions, scalar multiplications, dot products and matrix-matrix multiplications. In this work we address this issue, and provide a bridge between the low level control that an expert needs and the high level benefits of a compiled language. We do this through the use of custom intrinsic like macros.

Implementing a kernel directly in assembly requires a substantial amount of human effort and is typically reserved for performance critical code. Thus, to ease the programmer's burden, vector intrinsics are often used as a replacement for the low level assembly language. The compiler maps these vector intrinsics to the specific vector assembly code. Figure 2 shows the mapping between different vector intrinsics and assembly instructions. During the translation of the application to machine code, the compiler applies various optimizations to increase performance. In this process the compiler may move and schedule instructions and assign named registers according to general purpose heuristics. Thus, the programmer loses the control over the instruction schedule of the application.

```
vmulpd %ymm8, %ymm5, %ymm9 //assembly
t9 = _mm256_mul_pd(t8, t5); //intrinsic

vaddpd %ymm9, %ymm1, %ymm1 //assembly
t1 = _mm256_add_pd(t9, t1); //intrinsic

vperm2f128 $1, %ymm5, %ymm5, %ymm6 //assembly
t6 = _mm256_permute2f128_pd(t5, t5, 1); //intrinsic
```

Figure 2: The vector assembly instructions can be replaced with the vector intrinsics offered by vendors. The named registers are replaced with variables names. The compiler performs the translation between the intrinsics and assembly along with the mapping between the variable names and the named registers.

A programmer wants the best of both worlds. On one hand, he or she wants full control over the instructions and the scheduling mechanisms while on the other hand he or she desires programmability. The programmer simply wants to write the kernel in a high level language such as C, but somehow inhibit the compiler from moving and scheduling instructions, while still using it for operations such as register coloring or other optimizations for the glue code around the kernels. In other words, certain parts of the application or the library should not be modified by the compiler.

In this paper, we propose a mechanism for preserving instruction order that is as transparent to the programmer as existing compiler intrinsics. We use the inline assembly compatible with the `gcc` compiler to have control over application, however we embed our construct within parameterized C macros to hide the low-level details. Moreover we use the volatile construct to notify the compiler not to touch the instructions and preserve their statically scheduled order. Figure 7 shows an example of a vector addition

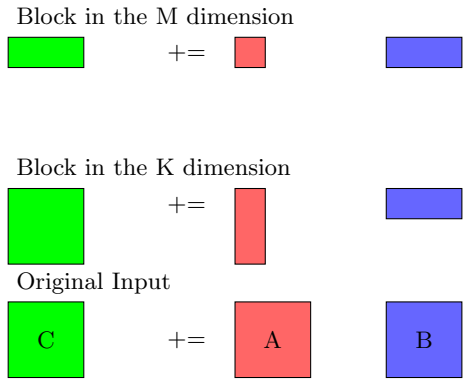

Figure 3: GotoBLAS approach layers loops around an extremely tuned assembly coded kernel. It first blocks the original input in the K dimension, then it blocks in the M dimension after that the blocks are fed to the tuned kernel.

described using our parametrized C macro. We use these macro instructions within matrix multiplication kernels and show that static scheduling of the kernels outperform the same kernels written with the normal vector intrinsics compiled with the same compiler.

Contributions.

Our work contributes the following:

- **Parametrized C macros.** We introduce vector macros that provide the same ease of programming afforded by traditional vector intrinsics, while providing the programmer a level of control of instruction selection and schedule that one would expect from programming in assembly.
- **Demonstration of flexibility.** We demonstrate the flexibility of these customized instructions through the implementation of high-performance matrix-matrix multiply kernels. We used generated and optimized code to show that when the customized instructions are used the performance is increased in comparison to generated code with of using compiler vector intrinsics.

2. BACKGROUND

As our running example, we focus our attention to matrix-matrix multiplication. When coded and tuned by an expert, this operation can achieve near the peak machine performance due to its $O(N^3)$ floating point operations to its $O(N^2)$ memory operations. Furthermore, there is great interest in achieving peak performance for every new architecture because the basic linear algebra subroutines (BLAS), specifically the level-3 BLAS [3], casts the bulk of its computation in terms of matrix-multiplication. Therefore, any improvements in the performance of matrix-multiplication translates to improvements in the rest of the level-3 BLAS and the numerical and scientific libraries that build upon it.

Without loss of generality, we further our focus on the variant of matrix multiplication of the form.

$$C = AB + C \tag{1}$$

```
// C = AB + C
for(int i = 0; i < m; ++i) {
  for(int j = 0; j < n; ++j) {
    for(int p = 0; l < k; ++k) {
      C[i][j] += A[i][p] * B[p][j];
    }
  }
}
```

Figure 4: The simplest implementation for the matrix-matrix multiplication algorithm. The inner-most loop computes the inner product between the rows of matrix A and the columns of matrix B. The outer two loops iterate through the rows and columns of the two matrices.

where the matrix $C \in \mathbb{R}^{m \times n}$. The simplest implementation of an algorithm for computing the matrix multiplication can be done with three nested loops, similar to the example written in C in Figure 4. The inner-most loop (k index) performs the inner product between row i of matrix A and column j of matrix B. The outer two loops (i and j index) iterate through the rows and columns of the two matrices. Although the implementation is fairly simple, the attainable performance of this approach is low. Thus, over the past 20 years there has been substantial efforts in the advancement of high performance implementation of matrix multiplication.

Layering for Performance.

GotoBLAS [5] (now maintained as OpenBLAS [9]) identified that a high performance implementation of matrix-matrix multiplication can be achieved by layering loops around an extremely tuned assembly implemented kernel. Figure 5 shows a simple implementation of a 4 by 4 kernel for matrix-matrix multiplication. The exposed matrix-matrix kernel

```
vmovaps       (%rcx), %ymm8
vmovaps       (%rdx), %ymm4
vmulpd         %ymm8, %ymm4, %ymm9
vaddpd         %ymm9, %ymm0, %ymm0
vshufpd    $5, %ymm4, %ymm4, %ymm5
vmulpd         %ymm8, %ymm5, %ymm9
vaddpd         %ymm9, %ymm1, %ymm1
vperm2f128 $1, %ymm5, %ymm5, %ymm6
vmulpd         %ymm8, %ymm6, %ymm9
vaddpd         %ymm9, %ymm2, %ymm2
vshufpd    $5, %ymm6, %ymm6, %ymm7
vmulpd         %ymm8, %ymm7, %ymm9
vaddpd         %ymm9, %ymm3, %ymm3
```

Figure 5: An example of a matrix-matrix multiplication of size $m = 4$, $n = 4$, and $k = 1$ using vector assembly instructions. Assembly coding with vector instructions provides control to the programmer, but requires the programmer to select instructions, manage registers, and schedule instructions manually.

operates on a large block L2 cache resident block of A. Smaller portions of B and C are blocked for the L1 cache and must be carefully implemented such that the kernel computes floating point operations at the rate that the elements are streamed from cache. In order to achieve this an expert needs to perform the following optimizations:

- An appropriate mix of SIMD instructions need to be selected such that the processor can sustain their rate of execution.
- These instructions must be statically scheduled such that their latencies are hidden by overlapping instructions.

Additionally, the following low level optimizations are necessary in order to prevent the fetch and decode stage of the processor from becoming a bottleneck.

- In some cases, we generate instructions that are meant to operate on single-precision data instead of instructions that operate on double-precision data. An example of this is the use of the `vmovaps` instruction to load `reg_a`, instead of `vmovapd`. This is because both instructions perform the identical operation but the single-precision instruction can be encoded in fewer bytes.
- For memory operations, address offsets that are beyond the range of -128 to 127 bytes require additional bytes to encode. Therefore, we restrict address offsets to fit in this range by subtracting 128 bytes from the base pointers into A and B.

The optimizations require that the expert has a certain level of control over their code. In the following sections we will discuss the various alternatives that an expert can use to implement the kernels.

Intrinsic functions.

Intrinsic functions are instructions offered by different programming languages which are handled differently by the compiler. The compiler replaces the intrinsic function with an instruction or with a sequence of instructions (i.e. assembly instructions) similar to an inline function. However, the compiler has some extra knowledge regarding the intrinsic function. This extra knowledge is helpful because it guides the compiler to better optimize the application. These optimizations are applied only when the user explicitly specifies it to the compiler through a flag, i.e. `-mavx` for vector intrinsics using the Advanced Vector Extension (AVX) SIMD instruction set, or `-fopenmp` for parallel intrinsics. In this paper we focus on the vector intrinsic functions that may help with the performance improvement on architectures that offer vector units.

Vector intriniscs are intrinsic functions that specify to the compiler that the instructions are vector assembly instructions and that the data on which the instructions operate are stored in vector registers, as opposed to scalar registers. The difference between a vector register and a scalar register is that the vector register can store more data. For example, a single AVX 256-bit width register can store up to four double precision floating point number, whereas a scalar 64-bit register that can hold exactly one double precision floating point number.

3. CUSTOM INTRINSICS

```
__m256d y8 = (__m256d)_mm256_load_ps(&A[0]);
__m256d y4 = (__m256d)_mm256_load_ps(&B[0]);

__m256d tmp = _mm256_mul_pd(y8, y4);
y0 = _mm256_add_pd(tmp, y0);

__m256d y5 = _mm256_shuffle_pd(y4, y4, 5);
tmp = _mm256_mul_pd(y8, y5);
y1 = _mm256_add_pd(tmp, y1);

__m256d y6 = _mm256_permute2f128_pd(y5, y5, 1);
tmp = _mm256_mul_pd(y8, y6);
y2 = _mm256_add_pd(tmp, y2);

__m256d y7 = _mm256_shuffle_pd(y6, y6, 5);
tmp = _mm256_mul_pd(y8, y7);
y3 = _mm256_add_pd(tmp, y3);
```

Figure 6: The example in Figure 5 implemented with the help of vector intrinsics. The variables are declared as `__m256d` which specifies to the compiler that the register is going to contain 4 double precision doubles.

3.1 Custom instructions

We introduce custom intrinsics that provide the control offered by assembly instructions, while also allowing us to leverage the assistance offered by the compiler when coding with vector intrinsics or intrinsics in general.

Customized intrinsics are parameterized C macros that assembly instructions into the code at compile time (See Figure 7). Since customized intrinsics are replaced with assembly instructions at compile time, it is the same as assembly programming (at a slightly higher level of abstraction), which implies that the programmer retains the same level of control from programming in assembly. In addition, the parameters of the macros are variables holding the inputs and output of the assembly instructions. At compile time, the variables are replaced with actual register names. This leverages the register allocation algorithms in the compiler, thus relieving the programmer from having to manually perform the book keeping.

The template for a customized instruction is shown in Figure 8. We use inline assembly constructs so that the customized instructions are recognized by most C preprocessors and/or compilers. The `__asm__` construct tells the compiler that the arguments within the brackets represent the assembly instruction and the extra information required by the instruction. The first argument represents the assembly instruction which should follow a specific pattern, i.e. `vaddpd %[vsrca], %[vsrcb], %[vdest]`. The next arguments represent the operands for the assembly instruction. Whenever we desire to specify an output we have to mark the operand with an `=` sign, i.e. `[vdest] =x (dest)`.

```
#define VADD(srca,srcb,dest)              \
asm volatile(                               \
    "vaddpd %[vsrca],%[vsrcb],%[vdest]\n"   \
    : [vdest] "=x"(dest)                    \
    : [vsrca] "x"(srca),                    \
      [vsrcb] "x"(srcb)                     \
    );
```

Figure 7: The code represents an example of the customized intrinsics, which is essentially a paramaterized C macro. The macro is marked as "volatile" to prohibit the compiler from optimizing the instruction(s). As such, the macro will be translated directly to the vector assembly instruction that we want to have in our kernel.

The above template offers us control over the code, because we are still writing the application using assembly. However, the compiler can still move these instructions around, trying to schedule them according to the scheduling algorithm specific for each compiler. A simple solution to disable code movement is to use the "volatile" construct which should lay in front of the assembly template, i.e.:

`volatile __asm__(...)`.

Therefore one could statically schedule the code for a specific architecture, knowing that the optimization done by the compiler will not affect the assembly inserted using the customized intrinsics.

```
__asm__( assembler template
         : output operands
         : input operands
         : optional list
         );
```

Figure 8: The template for writing inline assembly instructions that can be used within C programs. The first argument represents a string for the assembly instruction, while the next arguments represent the operands for the instruction

In order to relieve the programmer from the task of managing the use of the registers, we wrap the assembly constructs together using the macro definitions offered by the programming languages, such as C/C++. In C, we define the macros using the `#define` construct. We parameterize the macro by adding arguments, where these arguments will play the role of placeholders for the variables used in the application. This allows us to leverage the compiler register management capabilities. More importantly, this raises the level of abstraction since the actual assembly instructions are hidden from the programmer, and the use of variables instead of register names allows one to treat these customized intrinsics as regular intrinsic/function calls. Moreover, the customized intrinics permit us to schedule the code before compilaton, without any worry that the compiler might move instructions around.

One particularly useful extension to our custom macros is that we can implement our own intrinsics header file that mimics the compiler's built in intrinsics. This allows the programmer to transition existing vectorized code to use our macros.

3.2 A matrix-matrix multiplication example

Using the matrix-matrix multiplication kernel as an example, one could use vector intrinsics rather than the vector assembly instructions (See Figure 6). In this case, the compiler will have a more significant role, because it will have to translate the application to machine code. Based on the ex-

```
inline __m256d _mm256_load_pd(double * A){
  __m256d ret;
  VMOV(ret, A);
  return ret; }

inline __m256d _mm256_mul_pd(__m256d A, __m256d B){
  __m256d ret;
  VMUL(ret, A, B);
  return ret; }
```

Figure 9: In the case of legacy code we can create a drop-in replacement for the compiler intrinsics. The existing compiler intrinsics can be redefined using our custom macro instructions. This provides the benefit that the instruction will be preserved while keeping the same interface as the compiler intrinsics.

perience with matrix-matrix multiplication kernel, we have identified a class of vector intrinsics that are useful for the computation. The intrinsics can be classified in the following categories:

- **Load intrinsics** are instructions that move data from memory to registers, i.e. `_mm256_load_pd`. For example, architectures that offer AVX support permit one to store either 8 single precision floats or 4 double precision doubles. Moreover, different instructions can fill the entire vector register with the same data points or with completely different data points.
- **Store intrinsics** are similar to the load intrinics, they move data from memory to registers, i.e. `_mm256_store_pd`.
- **Permutation intrinsics** are instructions that perform in register permutations or data shuffling, i.e. `_mm256_shuffle_pd` or `_mm256_permute2f128_pd`. The permutations are used for efficient computation of the matrix-matrix multiplication kernel. The matrix-matrix multiplication requires a reduction operation, which is costly, therefore shuffle operations permits the movement of data around to do the reduction computation.
- **Compute intrinsics** are the instructions that perform the compute similar to the example presented in Figure 10. Additions and multiplications are the basic instructions used for matrix-matrix multiplication, i.e. `_mm256_add_pd` or `_mm256_mul_pd`. For newer architectures, one could use the fused-multiply add instructions, i.e. `_mm256_fmadd_pd`.

The vector intrinsics that fall in the 4 categories can be used to implement the kernel for the matrix-matrix multiplication. Vector intrinsics raise the level of abstraction and ease the burden on the programmer, such as the bookkeeping of the registers. Instead of using explicit vector registers, the programmer will have to declare the variables with a specific data type that will give enough information to the compiler to map the variables to the specific registers. The compiler uses a coloring algorithm when mapping the variables to registers. The goal is to keep the data as close as possible to the computation due to the low latencies of accessing data in registers. Spilling to main memory will increase the latency of bringing it back when computation requires it. Besides this optimization, the compiler also

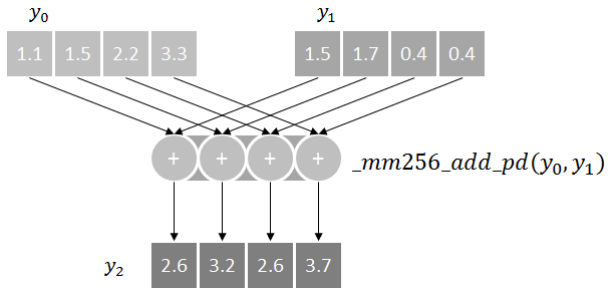

Figure 10: Vector intrinsic applied on two vector registers `y_0` and `y_1` of vector type `__mm256d`, which is specific for AVX 256-bit width four way double precision. The SIMD intrinsic describes a vector addition between the two vector registers. The output of the instruction is stored in the third vector register `y_3`. The compiler translates this instruction to the vector assembly instruction `vaddpd`.

applies scheduling of the instructions. In some situations the scheduling is appropriate for the underlying architecture. However, there are scenarios where the compiler may cause performance degradation especially when the code is already optimized and scheduled for a targeted architecture.

Typically the control over the code's schedule is moved given to the compiler. While writing the application in assembly inhibits the compiler from moving instructions around, using intrinsics makes it simpler to implement the problem but it also forces one to interact with the compiler to obtain the executable. Register coloring is a useful feature that reduces the burden on the programmer to keep track of how data is being moved between instructions. Scheduling on the other hand, may cause instructions to be moved around destroying the static schedule, if one is applied. Based on the advantages and disadvantages presented so far, one would desire the best of both worlds, the control offered by the assembly instructions and the ease of programmability offered by intrinsics and the compiler.

4. EXPERIMENTS

In this section we demonstrate the effectiveness of our custom macro instructions by comparing the performance of various implementation of matrix-multiplication implemented using our macros versus vector intrinsics and compiler auto-vectorization. The implementations that we selected are representative of how an expert would tune an optimize an implementation of a matrix-multiplication kernel. What we will show is that our custom macros preserve these hand optimizations.

Methodology.

For all of our experiments we compare the performance of various tuned and untuned matrix-multiplication kernels. Our implementations are compiled using either the GNU Cross Compiler version 4.8.3 and Intel Compiler version 14.0.3.174. The two machines used in are experiments are an Intel Xeon X7560 running at 2.27GHz (Nehalem) with a peak of 9.08 GFLOPS and an Intel Xeon E3-1225 running at 3.10 GHz (Sandy Bridge) with a peak of 24.8 GFLOPS.

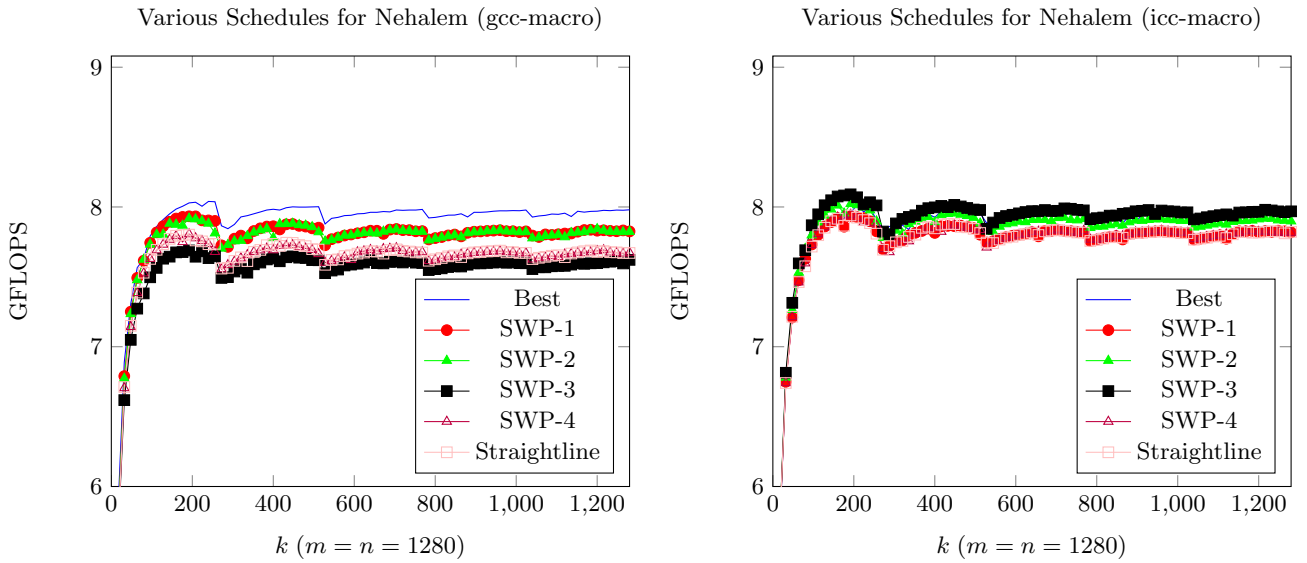

Figure 11: Here we demonstrate that our custom macro instruction wrappers preserve instruction order after compilation. We do this by implementing the same kernel with different instruction schedules and comparing their performance. The line labeled Best was scheduled using the software pipelining algorithm which overlaps instructions for different iterations and achieves the overall best performance. The lines labeled SWP-N are variants of best but with decreased amounts of instruction overlap. The larger the N in SWP-N the more overlap and the more likely the implementation will incur instruction stalls. The line labeled Straightline has no instruction overlap and does not hide the latency of each instruction like the other variants. what this demonstrates is that static scheduling affects performance even on out-of-order processors. On all plots peak performance is the top of the y-axis. Note on this plot the minimum value of the y-axis is offset to show the relative difference of the lines.

Auto-vectorization versus Expert.

In this first experiment, we establish the need for an expert for implementing a matrix multiply kernel. To demonstrate this we compare the performance of a straight line C implementation of a matrix-multiplication kernel against the same kernel that was expertly implemented using our custom macro instructions and compiled using gcc. The straight line C implementations were compiled using the auto-vectorization optimization such that the inner-most loop of the kernel was transformed to use SIMD instructions. In the expertly implemented kernel, we determined an efficient instruction schedule that minimized stalls and used our custom macros to implement it.

The results (Figure 1) can be interpreted as follows: on the X-axis we vary the size of the k dimension where $m = n = 1280$. This value is selected so that when k is large we see the behavior of the operation as it operates on data in the memory for large values of k while at the same time when k is small we also see the behavior of the operation when it operates on cache resident data. On the y-axis we measure GFLOPS, where the top of the y-axis represents the peak performance of the machine. Each line represents an implementation. The expert tuned kernel reaches near the peak performance of target hardware. However, the gcc implementation only achieves slightly above 25% of the peak of the machine. The icc performs better, but still falls below the expert implementation. When we inspect the assembly code emitted by the compilers it is clear that compiler does not make efficient use of the registers and resorts to spilling and filling from memory. This is because the matrix-multiplication kernel is a tightly constrained operation and requires careful instruction scheduling in order to prevent spilling while still achieving high performance. Even with auto-vectorization, compilers fall short on delivering the performance that an expert programmer can achieve.

Static Scheduling Matters.

In the first experiment we demonstrated that high performance can be achieved from a kernel implemented in C using our custom macros. In this second experiment, Figure 11, we show that static instruction scheduling is still necessary for performance and our custom instructions preserve the relative instruction order after being compiled by the gcc and icc. Even in the absence of spilling, the same instructions scheduled in different ordering achieve different performances. This suggests that static scheduling impacts performance of tightly constrained numerical kernels even on out-of-order processors. In both plots the straightline implementation is a compiler scheduled unrolled implementation.

The line labeled Best is statically scheduled in a software pipeline fashion [7] to maximize theoretical performance. Software pipelining is a approach for instruction scheduling that is typically used for Very Long Instruction Word (VLIW) processors that hides the latency of instructions in a loop by restructuring the loop by interleaving non-conflicting instructions from different iterations. Using this approach the latency of each instruction is hidden by over-

lapping instruction from different iterations. By decreasing how much the instructions overlap we get the lines labeled SWP-N. These are variants of Best, where the larger the number is the less instruction overlap is performed and the more instruction stalls occur.

The line labeled Best uses a schedule that achieved the best overall performance between the two compilers. This performance was achieved on the gcc but not on the icc. We suspect that the optimizations performed by the icc are not as effective on already optimized code. Furthermore, we suspect that the icc is very aggressive with its transformation because the performance of the SWP-N variants are much closer to each other than in the gcc plot. In the gcc plots, the less overlap in the code (SWP-N with larger numbers and straightline) the lower the performance because the instruction latency is not hidden. We believe that the reason why the SWP-3 performance is worse than SWP-4 and the straightline might be because this implementation clusters large instructions together which would require multiple cycles to decode.

When we examine the assembly generated by the compilers, the ordering of the instructions are preserved. The results show that our macros can affect the instruction schedule in a predictable manner and that static scheduling impacts performance.

Experts needed for Scheduling.

In the previous experiment we demonstrated that static scheduling affects performance even on out-of-order processors. In this following experiment we will demonstrate that given our custom macro instruction an expert can schedule a matrix-multiply kernel that achieves better performance than a compiler. We do this by comparing an expert scheduled kernel that was implement using our macros, in order to preserve the selected schedule, against a compiler scheduled implementation using compiler intrinsics. This allows the compiler to determine an ordering of those selected instructions. In both implementations the resulting code has the exact same instructions, but potentially in a different ordering.

In Figure 12 we compare the performance of the best scheduled implementation using our custom macros versus a compiler scheduled implementation using compiler intrinsics. The lines labeled Intrinsic Compiler Scheduled represent the performance of a compiler scheduled, straight-line implementation of the matrix multiply kernel that uses the built-in compiler intrinsics. This implementation gives the compiler the freedom to schedule the instructions in the kernel as it sees fit. The lines labeled Custom Macro Scheduled are the implementations that have been scheduled for performance and implemented using our macros to preserve said schedule. For both systems and both compilers the scheduled implementation using the custom macros outperforms the compiler scheduled implementation. We suspect that for tightly constrained kernels the heuristics used by the compilers are not as effective as software pipelining.

Compiler Intrinsics are not enough.

In the previous experiment, we established that for a high performance implementation of matrix multiplication an expert cannot rely on the compiler to schedule the instructions in the implementation. In this experiment, Figure 13, we compare the effectiveness of our custom macros against the built in compiler intrinsics. For each compiler and system combination we use the software pipelined, expert implemented scheduled from the previous experiment (Figure 12) and implement it with our custom macros and the compiler intrinsics. What we want to test is if the compiler preserves the instruction ordering when intrinsics are used or if our custom macros are needed to preserve ordering.

On the Nehalem there is a significant performance difference between the two implementations on both compilers. Even though both implementations have the same instruction order, the compiler reorders the intrinsic implementation, but does so sub-optimally. On the Sandy Bridge the performance difference is slight. We examined the assembly code generated by both the icc and gcc and in the two cases ordering of the instructions are not the same as the initial ordering, so the intrinsics do not maintain the ordering. In the previous experiment (Figure 12) the ordering was not close enough to an optimal one and from the compiler scheduled implementation the compiler did not achieve the same performance as the expert. We suspect that because of its large reorder windows, the Sandy Bridge is less sensitive to the instruction order than the Nehalem.

5. CONCLUSION

In this paper, we show a simple solution where the programmers can have full control over the code without explicitly using assembly language. We propose parametrized C macros that wrap inline assembly instructions marked with the "volatile" construct. The macro instructions hide the low level details of the assembly language, permitting the programmer to use high level constructs such as variables and leave the mapping of the variables to the named registers in the scope of the compiler. Moreover, the "volatile" construct inhibits the compiler from touching the instructions and moving them around. Therefore, in the scenario that the code is automatically generated and scheduled, the programmer will have the guarantee that the compiler will not affect the code. We use these parametrized macro instructions within matrix-matrix multiplication kernels and show that they actually bring benefits, in comparison to the same code which makes use of the normal off the shelf vector instrinsics.

APPENDIX

A. REFERENCES

[1] Neon programmers guide.

[2] *Power ISA Version 2.07*. May 2013.

[3] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[4] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[5] K. Goto and R. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12:1–12:25, May 2008.

[6] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Sept. 2015.

[7] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 318–328, New York, NY, USA, 1988. ACM.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[9] http://xianyi.github.com/OpenBLAS/, 2012.

[10] F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for generating blas-like libraries. *ACM Trans. Math. Soft.*, 2013. Accepted.

[11] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

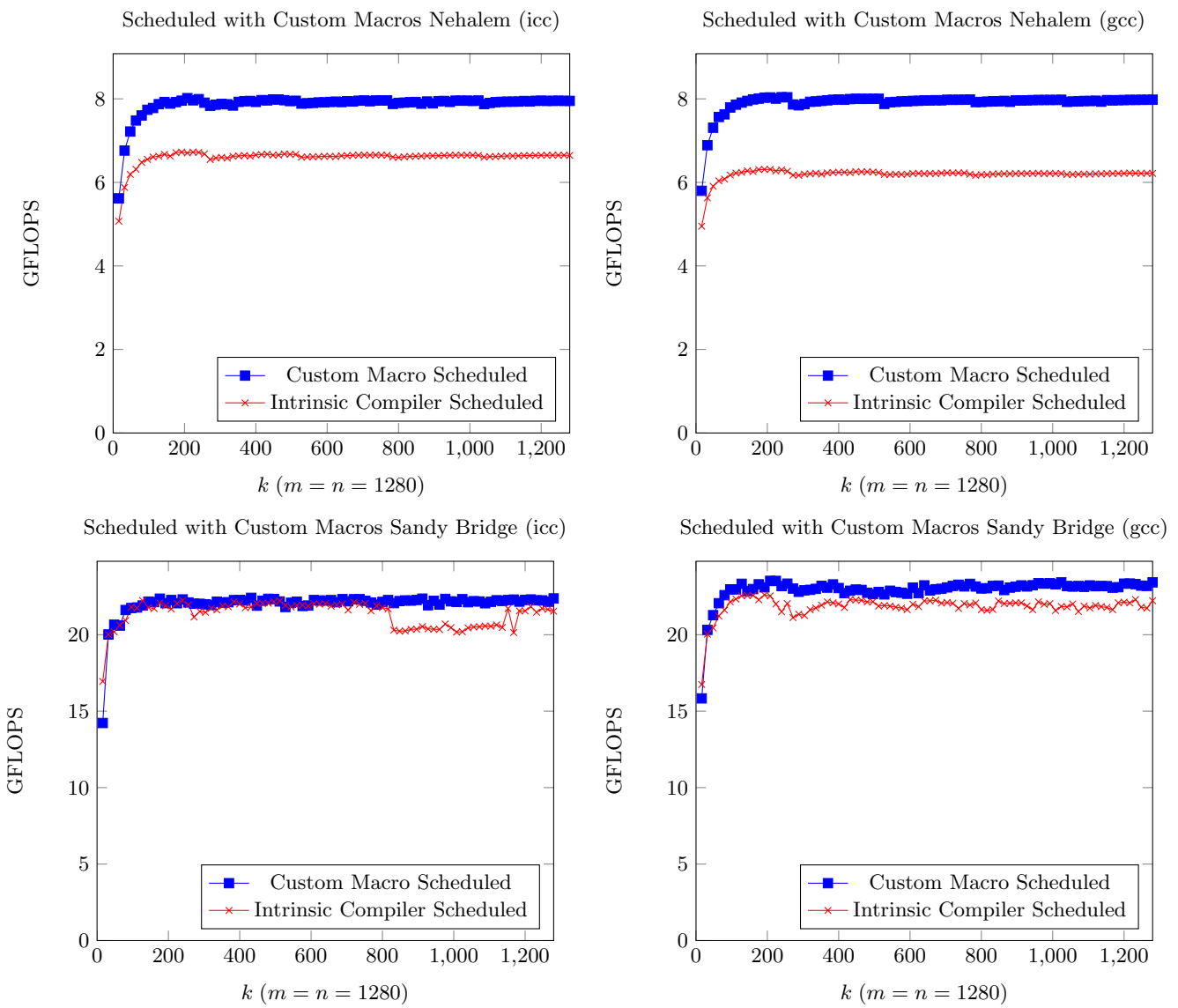

Figure 12: In these plots we show that even when given the same mix of instructions to implement a kernel, an expert implements a higher performance kernel than the compiler. For each of machine and compiler combinations we compare an implementation that is hand scheduled and coded using our custom against the same ordered set of instructions coded using compiler intrinsics. The hand scheduled instructions perform significantly better than the compiler scheduled implementations.

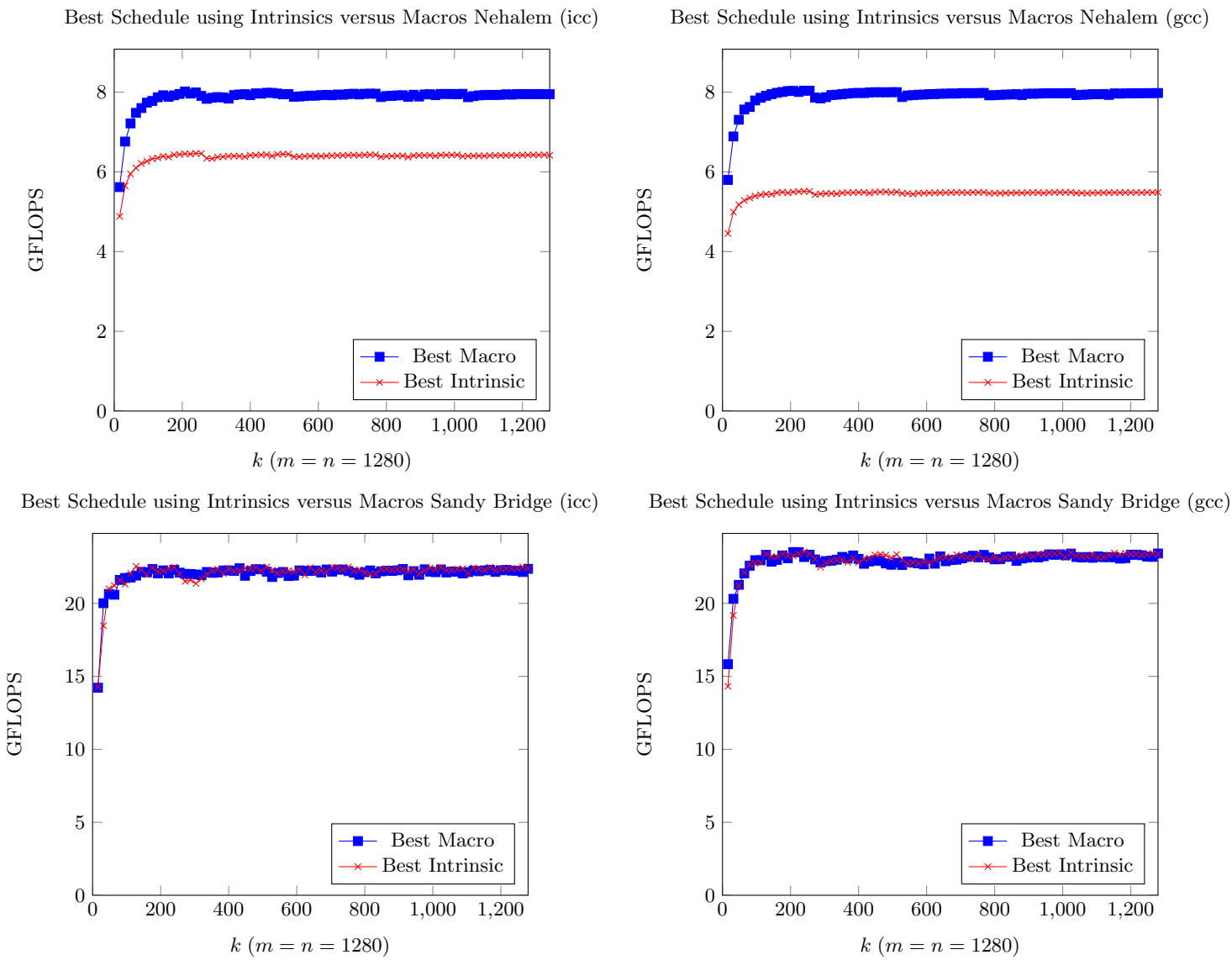

Figure 13: In these plots we show that compiler intrinsics do not preserve the implemented instruction schedule as well as our custom macro instruction wrappers. For each system and compiler combination we implemented the same matrix-matrix multiply with the same static schedule using our custom macro instructions and using the compiler intrinsics. The x-axis is the problem size and the y-axis represents performance in GFLOPS. The top of the plots represent the peak performance of the target machines. For the Nehalem there is a significant difference in the performance. On the Sandy Bridge the performance is comparable, but the assembly code produced by the compiled intrinsics does not preserve the order of the instructions. However, the compiler intrinsic implementations do not achieve this level of performance unless the instructions are scheduled.