Programming Dense Linear Algebra Kernels on Vectorized Architectures

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Jonathan Lawrence Peyton

May 2013

I dedicate this Thesis to my wife, Samantha. Thank you for being patient with me.

Acknowledgements

I would like to thank Dr. Gregory Peterson for all his edifying guidance through these years of graduate school and for taking me on as his student. I would also like to thank my wife, Samantha, for taking this journey with me from beginning to end. A special thanks goes to my parents, Ethelyn and Barry Peyton, and my mother-in-law, Betty Strachan, for all their support.

"Imagination is more important than knowledge. Knowledge is limited." - Albert Einstein

Abstract

The high performance computing (HPC) community is obsessed over the general matrix-matrix multiply (GEMM) routine. This obsession is not without reason. Most, if not all, Level 3 Basic Linear Algebra Subroutines (BLAS) can be written in terms of GEMM, and many of the higher level linear algebra solvers' (i.e., LU, Cholesky) performance depend on GEMM's performance. Getting high performance on GEMM is highly architecture dependent, and so for each new architecture that comes out, GEMM has to be programmed and tested to achieve maximal performance. Also, with emergent computer architectures featuring more vector-based and multi to many-core processors, GEMM performance becomes hinged to the utilization of these technologies. In this research, three Intel processor architectures are explored, including the new Intel MIC Architecture. Each architecture has different vector lengths and number of cores. The effort given to create three Level 3 BLAS routines (GEMM, TRSM, SYRK) is examined with respect to the architectural features as well as some parallel algorithmic nuances. This thorough examination culminates in a Cholesky (POTRF) routine which offers a legitimate test application. Lastly, four shared memory, parallel languages are explored for these routines to explore single-node supercomputing performance. These languages are OpenMP, Pthreads, Cilk and TBB. Each routine is developed in each language offering up information about which language is superior. A clear picture develops showing how these and similar routines should be written in OpenMP and exactly what architectural features chiefly impact performance.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Increasing high performance codes on upcoming architectures usually meant just waiting a few years for faster clock rates on single-processor computers. Unfortunately, the plateau of computer clock rates means software developers can no longer rely purely on Moore's Law [1] for performance improvements. This impasse has provided the impetus for reevaluation of Single Instruction Multiple Data (SIMD) and multi to many core architectures. This new computing philosophy now focuses on exploiting data and process level parallelism within processors rather than just faster single core processors [2]. There is plenty of evidence for this paradigm shift including larger vector sizes for emergent Intel® CPU Architectures, larger number of cores within a single chip, as well as utilization of GPU architectures which epitomize the mixture of numerous parallel ALU's with underlying SIMD architectural features [3, 4, 5].

One area of study highly affected by this paradigm shift is computational dense linear algebra. Because so much data parallelism exists within these problems, SIMD and many-core architectural features can be exploited easily to improve performance with every new architecture. In Section 1.1, the linear algebra problems this research focuses on are explained. In Section 1.2, the fundamental goals of the research are examined.

1.1 BLAS and LAPACK

Common linear algebra routines including LU decomposition and Cholesky decomposition are used to solve systems of equations. These fundamental computational kernels spend the majority of there time doing useful computations in Level 3 BLAS [6] by blocking the matrix to factor, doing a small LU or Cholesky decomposition, and then updating the rest of the matrix by calling GEMM or SYRK. Because the GEMM and SYRK routines can achieve 85% - 95% of peak on typical computer architectures, the LU or Cholesky decomposition will achieve good performance. The LAPACK [7] library describes this general idea of exploiting Level 3 BLAS' high performance. The benefits of this layered approach are twofold, high performance and portability. The high performance is achieved as described above, and the portability is achieved by relying on a ubiquitous underlying interface to matrix-matrix operations (BLAS).

The BLAS are an interface description of three different matrix/vector operations: 1) vector-vector, 2) matrix-vector, and 3) matrix-matrix operations. Implementations of the Level 3 BLAS can achieve much higher performance than the Level 2 and Level 1 routines can when optimized for a particular machine. Highly tuned libraries can achieve over 90% of peak performance on CPU architectures for the Level 3 BLAS routines[8]. When developing Level 3 BLAS routines, it has been noted by some [9, 10] that *all* Level 3 routines can be coded in terms of GEMM. Putting all the matrix-matrix operations in terms of GEMM gives performance benefits for all routines by optimizing only one. In this research, when developing triangular solve (TRSM) and symmetric rank-k (SYRK), this method of putting all routines in terms of GEMM is used.

1.2 Goal Of Research

This research focuses on developing the described GEMM, SYRK, and TRSM routines above on three different Intel® Architectures on a single node of a supercomputer using four different shared-memory parallel programming languages. All three routines are used

to create the LAPACK description of Cholesky decomposition to demonstrate there usage in a real application. For the GEMM routine in particular, a progression of performance enhancements is noted and the effort to get the best results is examined with respect to the architectural features of the systems as well as the parallel language features. The end result is a description of the effort required to achieve different levels of performance for each of these architectures with each of the parallel languages. This research also offers a good idea of which parallel language one should use for this area of applications.

Throughout this research, the baseline implementation for comparison is Intel's Math Kernel Library (MKL). This library is Intel's implementation of various high performance kernels and codes including the BLAS and LAPACK interfaces. It is assumed that MKL has near optimal performance for Intel architectures and provides a solid baseline to compare performance numbers against.

Chapter 2

Background on GEMM, TRSM, SYRK, and POTRF

All the routines described below are actually missing a letter prefix to there name. For example, the GEMM routine will be called with either a 'D' or 'S' in front of it creating the real routine call DGEMM or SGEMM. So TRSM will be DTRSM or STRSM, SYRK will be DSYRK or SSYRK, and POTRF will be DPOTRF or SPOTRF. This letter prefix indicates whether the matrices are single precision (4 bytes per element) or double precision (8 bytes per element) matrices. This research implements both the single and double precision versions of all routines. Below, they are described generically without this prefix. Much of this information can be found in [6, 11]. This section does not describe the algorithms of these routines and gives only the high level interface description of the dense linear algebra routines programmed for this research.

The GEMM routine is described as the general matrix-matrix multiply between two rectangular (or square) matrices A, B where that result is either stored directly in, or updates a matrix, C. The full BLAS description is shown concisely and programmatically as

$$C \leftarrow \alpha AB + \beta C$$

Where C is an $m \times n$ matrix, A is an $m \times k$ matrix, B is a $k \times n$ matrix and α and β are scalars. In this research the special case of $\alpha = -1.0$ and $\beta = 1.0$ is examined. This special case is the only used case for this research, and it is also one of the most often used special cases in general for creating the other Level 3 BLAS routines and for use in LAPACK. To update an element, c_{ij} in the C matrix, just apply:

$$c_{ij} \leftarrow \beta c_{ij} + \alpha \sum_{n=1}^{k} a_{in} b_{nj} \tag{2.1}$$

This routine is so critical to maximizing performance for dense linear algebra that its results and algorithmic techniques will be emphasized the most. All other BLAS routines can be described in terms of this routine. And the other routines can benefit from very similar optimizations discussed later. More importantly, many LAPACK definitions, including the paramount LU routine GETRF, use this routine as the primary matrix updating mechanism during execution as a way to speed up matrix factorizations/decompositions. For these reasons, GEMM will receive more focus than the other routines. In some sense, TRSM and SYRK are both applications of the GEMM routine as well as the LAPACK routine, POTRF. The importance of this routine cannot be stressed enough.

The TRSM routine is described as back substitution with multiple right hand sides using a triangular matrix, A, and a rectangular matrix B. The full BLAS description is shown programmatically as

$$B \leftarrow \alpha A^{-1} B$$

Where B is an $m \times n$ matrix, A is an $m \times m$ triangular matrix, and α is a scalar. The whole interface described in BLAS accounts for which side the A matrix is on, if the diagonal for the A matrix is all ones, and if the A matrix is upper or lower triangular. In this research the special case where A is on the left hand side, is non-unit on the diagonal, is lower triangular,

and where $\alpha = 1.0$ is the case studied. To update an element, b_{ij} in the B matrix, just apply:

$$b_{ij} \leftarrow \alpha \frac{b_{ij} - \sum_{n=1}^{i-1} a_{in} b_{nj}}{a_{ii}} \tag{2.2}$$

The SYRK routine is described as a matrix-matrix multiply between a matrix A and its transpose A^T in which that result is either stored directly in or updates a symmetric matrix C. The full BLAS description is shown mathematically as

$$C \leftarrow \alpha A A^T + \beta C$$

Where C is an $n \times n$ symmetric matrix, A is an $n \times k$ matrix and α and β are scalars. When programming a SYRK routine, only half of the C matrix gets updated, either the upper or lower triangular part (including the diagonal). The special case where the upper part of the C matrix is updated, $\alpha = -1.0$ and $\beta = 1.0$ is examined for the same reasons as GEMM. Similar to GEMM, to update an element, c_{ij} in the C matrix, just apply the equation:

$$\begin{gathered} c_{ij} \leftarrow \beta c_{ij} + \alpha \sum_{n=1}^{k} a_{in} a_{jn} \\ \text{or} \\ c_{ij} \leftarrow \beta c_{ij} + \alpha \sum_{n=1}^{k} a_{in} a_{nj}^T \end{gathered} \tag{2.3}$$

The LAPACK routine, POTRF, describes the Cholesky decomposition of a positive-definite symmetric matrix C into a lower triangular matrix L and an upper triangular matrix L^T where

$$C = L L^T$$

Where L^T is the transpose of the matrix L. Since the upper and lower triangular matrices L^T and L are transposes of each other, only one has to be calculated and stored in practice. In the LAPACK definition it states that either L or L^T is stored where the C matrix resides

in memory, hence either the upper or lower triangular part of C is changed. In this research it is assumed the upper triangular part of C is changed. This implies finding L^T.

Chapter 3

Background on Intel Architectures Used

This section will describe the three Intel Architectures tested for this research. Section 3.1 discusses deriving the computational characteristics of the Nehalem and Sandy Bridge architectures. Section 3.3 shows how to derive MIC performance and distinguishing hardware characteristics for the MIC architecture.

3.1 Nehalem and Sandy Bridge Computational Characteristics

These two Intel Architectures used, Nehalem and Sandy Bridge, are superscalar, x86 architectures featuring ALUs capable of executing one add instruction and one multiply instruction simultaneously every clock cycle [12]. This information provides an easy way to calculate peak GFLOPS (Billions of Floating Point Operations Per Second) for each machine simply using these equations:

$$(n \text{ cores}) \times (\text{CR(GHz)}) \times (\text{VL(singles)}) \times (2 \text{ IPC}) = \text{ Peak GFLOPS single precision} \quad (3.1)$$

and

$$(n \text{ cores}) \times (\text{CR(GHz)}) \times (\text{VL(doubles)}) \times (2 \text{ IPC}) = \text{ Peak GFLOPS double precision} \quad (3.2)$$

Where CR is the clock rate in GHz, VL is the vector length in either singles or doubles, IPC is the instructions per cycle.

As an example, a single core of the Nehalem machine as specified below in Table 3.3 is capable of

$$(1 \text{ core}) \times (3.6 \text{ GHz}) \times (4 \text{ floats}) \times (2 \text{ IPC}) = 28.8 \text{ GFLOPS single precision} \quad (3.3)$$

and

$$(1 \text{ core}) \times (3.6 \text{ GHz}) \times (2 \text{ doubles}) \times (2 \text{ IPC}) = 14.4 \text{ GFLOPS double precision} \quad (3.4)$$

In the equations above, the (2 IPC) refers to the simultaneous add and multiply instructions. The (4 floats) and (2 doubles) refers to the vector length of 128 bits.

The description in Table 3.1 is for a single core of each processor. When using multiple cores on the Nehalem architecture, the clock rate decreases according to the number of cores being used and the loads on each core. This feature is called Intel® Turbo Boost and allows the clock rate to vary with respect to the amount of work being done on a processor and the number of processors being used. Turbo Boost might seem a misnomer because it appears that the clock rate falls. This is misleading because the numbers in Table 3.1 are for only

Table 3.1: Peak GFLOPS for a single core on each different architecture

Architecture	Peak GFLOPS single precision	Peak GFLOPS double precision
Nehalem	28.8	14.4
Sandy Bridge	48.8	24.4
Intel MIC	33.5	16.75

a single core running. Turbo Boost is in effect for those numbers and increases the baseline clock rate from 3.3 GHz to 3.6 GHz (a 9.1% difference).

Because this feature varies the clock rate, a special program was developed to measure an "empirical clock rate". This program creates n number of threads and explicitly sets each thread's hardware affinity to a different core. Each thread then runs a sequence of unrolled, independent multiply and add instructions with zero memory references. The result of counting the number of computations and dividing by the execution time gives the "empirical computational peak" for n number of cores. From Equation 3.1 or 3.2 the clock rate can be derived. The final empirical clock rate is calculated by running the special program 25 consecutive times for each core count, then averaging the last 10 samples. Once the clock rate is known, the empirical computational peak can be calculated for both double and single precision. Figure 3.1 shows the Turbo Boost feature. In Table 3.2 the calculated empirical computational peak and empirical clock rates for various number of cores is shown.

The Sandy Bridge Turbo Boost feature was turned off and had a steady clock rate of 3.05 GHz independent of the number of cores running. This made the calculations for peak single and double computation bandwidth much simpler using Equations 3.5 and 3.6. Giving us

$$(16 \text{ cores}) \times (3.05 \text{ GHz}) \times (8 \text{ floats}) \times (2 \text{ IPC}) = 780.80 \text{ GFLOPS single precision} \quad (3.5)$$

and

$$(16 \text{ cores}) \times (3.05 \text{ GHz}) \times (4 \text{ doubles}) \times (2 \text{ IPC}) = 390.40 \text{ GFLOPS double precision} \quad (3.6)$$

3.2 Running Programs on MIC

The Intel MIC architecture is a coprocessor architecture[13] connected via a PCIe interface. A heterogeneous environment with highly superscalar CPUs and one or more MIC coprocessors can exist in a system. MIC can have a similar purpose as a GPU accelerator

by offloading highly parallel tasks from the CPUs onto the MIC coprocessor with the MIC coprocessor returning the results when the task is finished. This model of execution is called the offload model. However, the MIC coprocessor can run a small operating system on the card allowing users to log onto the MIC card directly. Programs can be uploaded to the card and run natively. This avoids all transfer costs associated with the offload model and reduces the problem to focus exclusively on the architecture. *All programs are run in native mode in this research.*

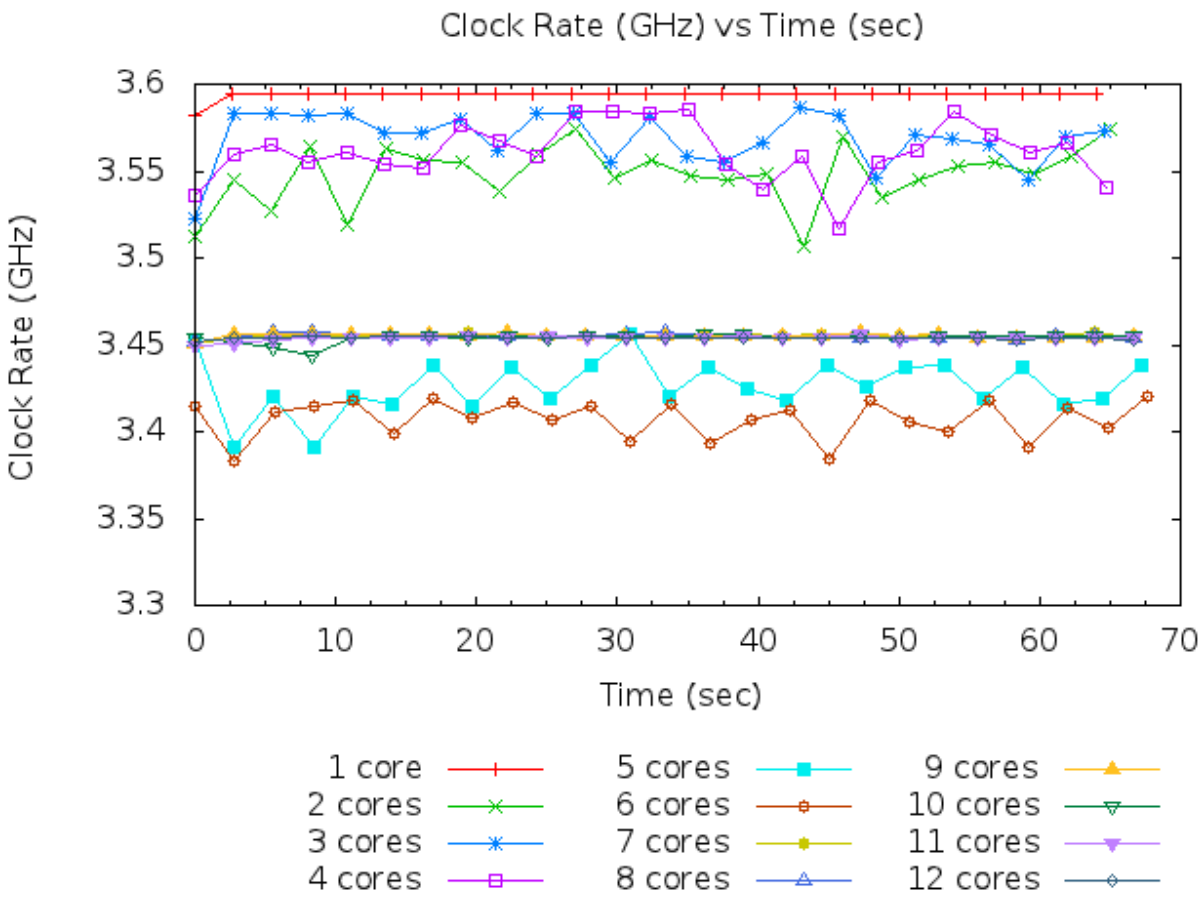

Figure 3.1: Clock rate (GHz) for Nehalem for various number of cores. It is unknown why for five and six cores, the clock rate is at its lowest. This was run numerous times with the same results.

3.3 MIC Performance

The single core performance for MIC is designated above as 34.56 GFLOPS single precision and 17.28 GFLOPS double precision. One peculiar feature of getting this performance is the required use of Intel® Hyper-Threading Technology. Hyper-Threading is a hardware performance enhancement that allows more than one thread context to execute efficiently on a single core. The Nehalem and Sandy Bridge have this feature, but in all test runs using the feature, it performed worse than just using a single thread per core. This is not the case for Intel MIC. Intel MIC's instruction issue stage requires at least two hardware thread contexts to choose from for full utilization of a core. This means if only one thread is executing on a single core, then the potential performance is cut in half. Each core for Intel MIC can support four hardware contexts efficiently. Every clock cycle, in a round-robin fashion, an instruction will be issued from a different thread context.

Since MIC does not have the Turbo Boost feature that Nehalem does, its computational peak for any number of cores is easier to calculate. From Equations 3.1 and 3.2,

$$(60 \text{ cores}) \times (1.047 \text{ GHz}) \times (16 \text{ floats}) \times (2 \text{ IPC}) = 2010.24 \text{ GFLOPS single precision} \quad (3.7)$$

and

$$(60 \text{ cores}) \times (1.047 \text{ GHz}) \times (8 \text{ doubles}) \times (2 \text{ IPC}) = 1005.12 \text{ GFLOPS double precision} \quad (3.8)$$

It should be noted that this specific equation shows the computational peak when using all 61 cores of the MIC processor, but for any number of cores, n (which require at least two threads for full utilization), one can simply substitute the 61 cores for n number of cores.

3.4 Summary of All Machines

Table 3.3 shows an overview of the most important features of each architecture.

3.4.1 Caches

For all machines in this research, the L1 and L2 caches are private per core caches. The L3 cache for both Nehalem and Sandy Bridge are per processor caches. The MIC architecture does not have an L3 cache.

3.4.2 Superscalar

Both the Nehalem and Sandy Bridge architectures are highly superscalar, and both are capable of issuing up to six instructions per cycle. Cores within the MIC architecture are not superscalar. Each core is actually a modified Pentium IV in-order core capable of two instructions per cycle. MIC does not allow any two instructions per cycle though. There are two pipelines in MIC featuring a full-fledged pipeline and a very restricted pipeline. the full-fledged pipeline can execute any instruction given while the very restricted pipeline is mostly used for prefetching data.

3.4.3 Vector Units

Each architecture has a different vector register set with different bit lengths. The Nehalem architecture features vector registers that can hold four single precision or two double precision values. The Sandy Bridge vector registers can hold eight single precision or four double precision values, and finally, the Intel MIC architecture has vector registers capable of holding sixteen single precision or eight double precision values. For each architecture, there exist common arithmetic, load/store, and shuffle instructions for manipulating the vector registers. More details on this are in Section 4.

Table 3.2: Approximate clock rate for Nehalem as well as the derived peak single and double precision computation bandwidth. Clock rate results were calculated by taking 25 samples and averaging only the last 10 and computational bandwidth results were calculated using Equations 3.1 and 3.2.

# Cores	Nehalem Approximate Clock Rate	Nehalem Peak (single/double)
1	3.59	28.76/14.38
2	3.55	56.79/28.39
3	3.57	85.70/42.85
4	3.56	113.77/56.89
5	3.43	137.15/68.58
6	3.41	163.51/81.76
7	3.46	193.50/96.75
8	3.45	221.07/110.54
9	3.45	248.76/124.38
10	3.45	276.38/138.19
11	3.45	303.96/151.98
12	3.45	331.56/165.78

Table 3.3: Overview of machines used. All L1 and L2 caches are private per core caches while all L3 caches are per processor caches. The clock rates vary based upon the number of cores used and

Arch.	Num. Procs.	# Total Cores	Clock Rate (GHz)	Hyper-Threading Used	L1 Cache (KB)	L2 Cache (KB)	L3 Cache (MB)	Vector Length (bits)
Nehalem	2	12	3.3-3.6	No	32	256	12	128
Sandy Bridge	2	16	3.05	No	32	256	20	256
Intel MIC	1	60	1.047	Yes	32	512	N/A	512

Chapter 4

Serial Implementations

The serial implementations of GEMM, TRSM, and SYRK are used as a key building block for any parallel implementation. For this reason, the serial implementation of these routines is paramount to achieving good performance for any parallel implementation. Many factors contribute to a serial implementation's ability to achieve near optimal performance including, but not limited to, blocking, inner kernel structure, the TLB, and vectorization. In this chapter the details of the serial implementation of GEMM, TRSM, SYRK, and POTRF are investigated. Many of the ideas are pulled from [8].

4.1 SIMD Hardware Background

In all the Intel Architectures, SIMD units exist and can be utilized to increase computational throughput. An abstract SIMD unit takes a single instruction (i.e. add) and applies it to multiple pieces of data. This data is typically floating point numbers. A common way of implementing a SIMD unit is for vector registers to hold multiple contiguous elements and for the programmer to use special vector instructions with these vector registers. For example, if $a = (a_0, a_1, a_2, a_3)$ and $b = (b_0, b_1, b_2, b_3)$ where a and b are vector registers holding four elements each, then adding them gives $c = a + b = (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$. This example illustrates the throughput increase. Four elements are added together instead of just one.

```
\\ SSE Intrinsics:
__m128 a,b,c;
a = _mm_load_ps(aindex);
b = _mm_load_ps(bindex);
c = _mm_add_ps(a, b);
_mm_store_ps(cindex, c);
```

```
\\ AVX Intrinsics:
__m256 a,b,c;
a = _mm256_load_ps(aindex);
b = _mm256_load_ps(bindex);
c = _mm256_add_ps(a, b);
_mm256_store_ps(cindex, c);
```

Listing 4.1: SSE and AVX Intrinsics

```
__m512 a,b,c;
a = _mm512_load_ps(aindex);
b = _mm512_load_ps(bindex);
c = _mm512_add_ps(a, b);
_mm512_store_ps(cindex, c);
```

Listing 4.2: MIC Intrinsics

Intel has different vectors for its varying microarchitectures. As can be seen from Table 3.3, Nehalem has 128-bit vectors called Streaming SIMD Extensions (SSE). These vector registers can hold either four single precision values or two double precision values. Sandy Bridge has larger, 256-bit vectors called Advanced Vector eXtensions (AVX). These vector registers can hold either eight single precision values or four double precision values. Finally, Intel MIC has even larger 512-bit vectors that can hold either sixteen single precision values or eight double precision values. Despite the vector size, all of these architectures offer the common arithmetic instructions as well as load, store, and shuffling instructions to modify the vector registers.

To use these vector registers, intrinsics were used rather than straight assembly. Intrinsics are slightly higher-level macros that wrap around the assembly language constructs. They allow common variable declarations without the need to explicitly allocate registers to each variable. The compiler is left to reorder intrinsic function calls as well as allocate the registers for each intrinsic function call. Some example code is shown below for clarity.

Table 4.1: Description of parameters for GEMM. $C = \alpha AB + \beta C$. A and B are source matrices while C is a destination matrix.

Term	Description
m	The number of rows in the A and C matrices.
k	The number of columns in the A and rows in B.
n	The number of columns in the B and C matrices.
M_c	Blocking dimension along m.
K_c	Blocking dimension along k.
N_c	Blocking dimension along n.
N_r	Register blocking factor along N_c.

4.2 Developing Linear Algebra Routines

In this research, all routines (GEMM, TRSM, SYRK, POTRF) were developed on the Sandy Bridge architecture first. The GEMM routine was created first, then TRSM, then SYRK, and finally POTRF. After all the routines were created for this architecture, they were ported to the Nehalem and MIC architectures. After this porting process was finished, the routines were then further optimized for that particular architecture. On the Sandy Bridge, the development process started from scratch and each step of increased performance was noted. The following sections will discuss each step of effort to achieve better performance generically for the GEMM routine. Section 4.8 discusses how TRSM is developed using the principles from GEMM and Section 4.9 discusses these principles with regards to SYRK. It is also reminded to the reader that final comparisons for all routines were to Intel's Math Kernel Library (MKL) which gives near optimal results for Intel x86 architectures.

4.3 Terminology

Throughout this paper numerous terms will be used. A source matrix is any read-only matrix for any of the routines. A destination matrix is the one and only matrix that is both a read and write matrix. In Figures 4.1, 4.6, 4.9, many of the dimensions are described with the appropriate symbol by the matrix feature. A table is also provided for descriptive purposes.

Table 4.2: Description of parameters for TRSM. $B = \alpha A^{-1}B$. A is a source matrix, while B is a destination matrix.

Term	Description
m	The number of rows and columns in A and number of rows in B.
n	The number of columns in the B matrix.
K_c	Blocking dimension along m.
N_c	Blocking dimension along n.

Table 4.3: Description of parameters for SYRK. $C = \alpha AA^T + \beta C$. A is a source matrix, while C is a destination matrix.

Term	Description
n	The number of rows in A and C, number of columns in C.
k	The number of columns in A.
M_c	Blocking dimension along m.
K_c	Blocking dimension along k.
N_c	Blocking dimension along n.
N_r	Register blocking factor along N_c.

4.4 Blocking For Cache

Perhaps the most well-known method to increasing performance for linear algebra kernels is blocking matrices for better cache reuse. This idea has been around for a long time [14, 8, 15, 16, 17] and has evolved to more complex data structures [14]. The purpose of blocking matrices is to take advantage of fast cache accesses as opposed to slow memory accesses. All the routines discussed in this research can benefit from blocking because elements from the source matrices are reused during computation. When first attempting to increase performance, the square blocking method[18] was used. This blocking method is simple and easy to implement, but does not take advantage of multi-layer caches and is suboptimal. Instead the best blocking method for x86 architectures observed was from [8, 19]. This blocking method as shown below in Figure 4.1 gives results on par with MKL. Looking at Figure 4.1, the first for-loop breaks the problem up into a series of panel-panel matrix multiplies by iterating over k. The second for-loop further reduces the problem by

iterating over n. In practice the entire submatrix A_{panel} can fit inside the L3 cache. The third for-loop reduces the problem by iterating over m and it should be seen that the submatrix B_{block} is put into either the L2 or L1 cache. In practice, B_{block} can be fit into the L2 cache and streamed to the execution units at nearly the same rate the execution units can compute. The fourth and final for-loop iterates over N_c and reduces the problem to its final state before an unrolled inner kernel executes the computations: $C_{slither} += A_{slither} B_{block}$. In this final state, the submatrix $A_{slither}$ can fit into L1 cache and $C_{slither}$ can fit into the registers. It can be seen from Figure 4.1 that A_{panel} is reused $\frac{n}{N_c} - 1$ times, B_{block} is reused $\frac{m}{M_c} - 1$ times, and $A_{slither}$ is reused $\frac{N_c}{N_r} - 1$ times. Register blocking will be talked about in Section 4.7.

4.4.1 Nehalem/Sandy Bridge Specific Blocking

For these two architectures, blocking is done on all three levels of cache. An observation that agrees with [8] is that no more than half the cache should be used. So B_{block} should only take up to half the L2 cache, $A_{slither}$ should only take up to half the L1 cache. This puts restrictions on the values of M_c, K_c, N_c, and N_r. From Figure 4.1, there is $K_c \times N_c$ elements the L2 cache corresponding to the B_{block}. There is also $M_c \times K_c$ elements in the L1 cache. Using only half of these caches gives us:

$$K_c \times N_c \times E_{size} \leq \frac{L2_{size}}{2} \tag{4.1}$$

and

$$M_c \times K_c \times E_{size} \leq \frac{L1_{size}}{2} \tag{4.2}$$

Where E_{size} is the size of the element in bytes and Lx_{size} is the size of the L1 or L2 cache in bytes.

4.4.2 MIC Specific Blocking

The MIC architecture is handled differently in terms of blocking. Since the architecture relies on Hyper-Threading, the L1 and L2 caches will be shared between multiple threads.

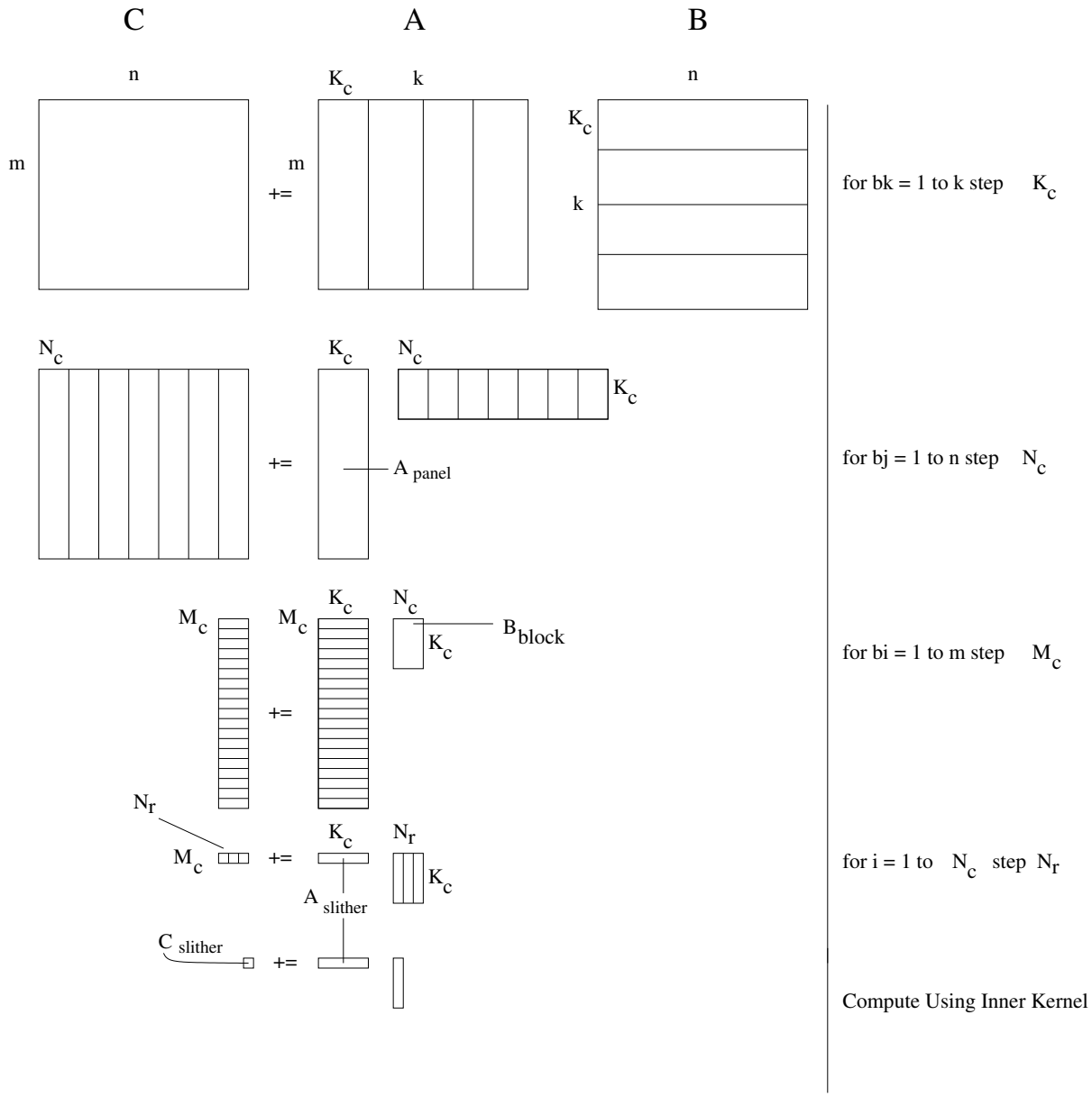

Figure 4.1: The GotoBLAS blocking scheme for GEMM.

Later on it will be shown that a core has best performance when running four threads per core for MIC. With this many threads running on a single core, and all these threads using the same L1 and L2 cache, the restrictions on K_c, N_c, and N_r hinder performance given that only half the cache should be used as noted above. For instance, if four threads are running on a single core of MIC, then the restriction in Equation 4.2 will not be met. The restriction

for MIC becomes:

$$(K_c \times N_c + M_c \times K_c) \times E_{size} \times \text{num}_{\text{threads}} \leq \frac{L2_{size}}{2} \tag{4.3}$$

Where $\text{num}_{\text{threads}}$ is the number of threads per core (typically less than or equal to four). It will be shown in Section 4.6 why this is acceptable for MIC and how the L1 cache is used.

4.5 Packing Data

Another optimization that greatly affects performance is packing the source matrices. Packing is defined as taking a row-major submatrix and copying it to an auxiliary buffer in such a way that the inner kernel accesses all data elements in a contiguous fashion. This can be seen in Figure 4.2. There are two huge benefits from doing this operation. First and foremost, contiguous memory accesses are most beneficial to the Translation Lookaside Buffer (TLB). The TLB is a "cache" for memory pages and is referenced for each memory access. If a memory reference is not in the TLB, then a page walk occurs and the page is brought in to take an entry in the TLB. Typical TLBs will have 64 to 128 entries that can each reference 4KB of contiguous data. If all the data in the auxiliary buffers is contiguous and fits in the TLB, then few TLB misses, which are expensive in terms of latency, will occur. It has been shown that a large component of overhead and latency involving GEMM is because of this TLB [20, 14, 8]. A second benefit is exploiting spatial locality from caches. Because elements are accessed contiguously, whole cache lines brought into the different caches will contain elements that will be accessed in the near future.

4.6 Prefetching Data

Perhaps the most machine dependent optimization explored was the prefetching of data. The idea behind prefetching is gathering data in the correct cache just before the data is referenced and is illustrated in Figure 4.3. There were two locations where prefetching

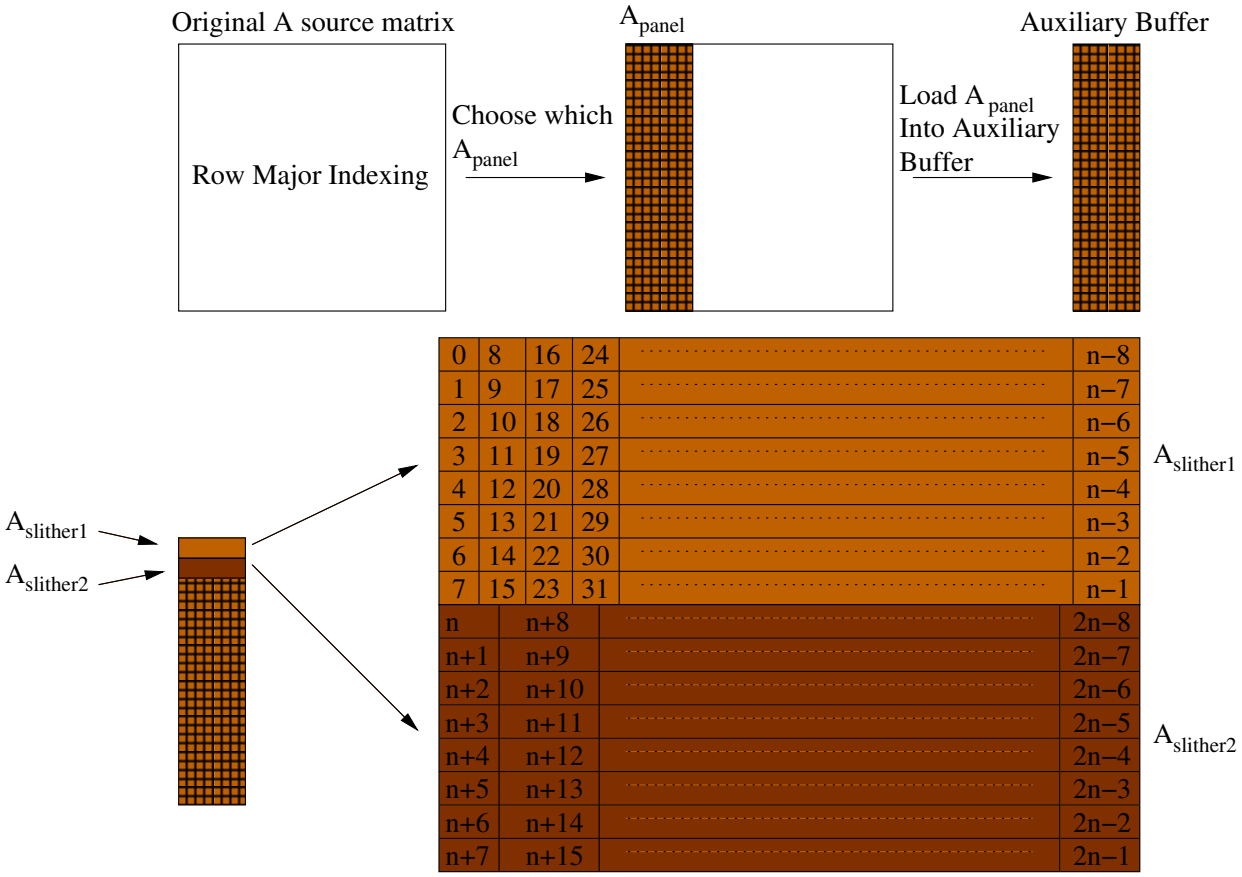

Figure 4.2: Illustration of packing a submatrix into an auxiliary buffer. The numbers indicate the index of the element.

helped. Looking at Figure 4.1, right after the fourth for-loop, prefetching the next $M_c \times N_r$ elements of the $C_{slither}$ matrix used into the L1 cache. Also, prefetching elements of the B_{block} matrix from the L2 cache into the L1 cache is used during the unrolled compute loop. In fact, prefetching the elements of B_{block} plays a crucial role in being able to stream elements from L2 to the execution units mentioned in Section 4.4. The end result of using all the prefetching is that all relevant data is in the L1 cache when it is referenced and hence will give near optimal performance.

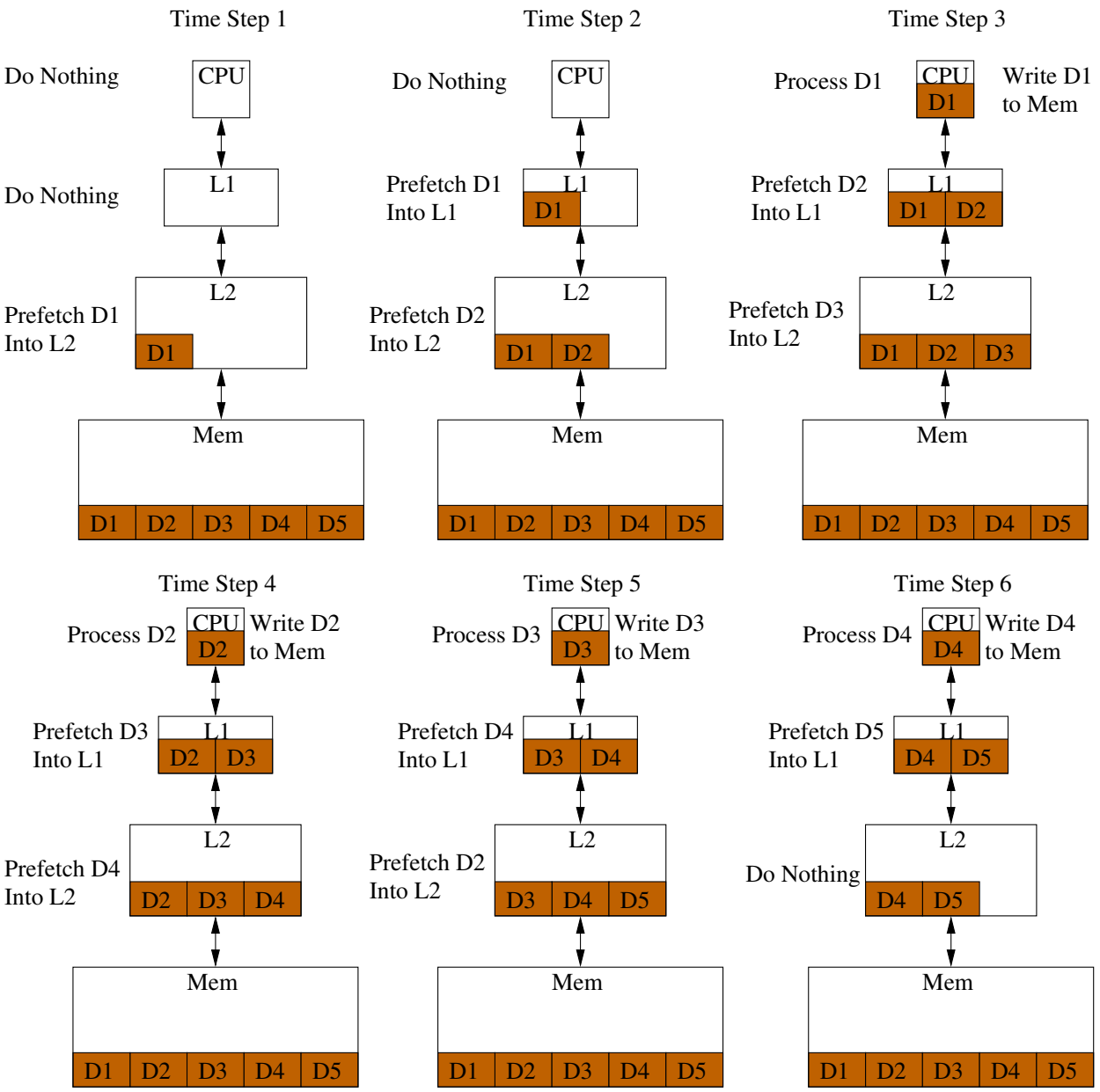

Figure 4.3: Illustration of prefetching data through multi-level cache system. There are six time steps shown. Notice the natural software pipeline that occurs.

4.6.1 Nehalam/Sandy Bridge Specific Prefetching

For these two architectures, the prefetching for $C_{slither}$ mentioned above is used. The Nehalem architecture does not use the prefetching of B_{block} but Sandy Bridge does use the prefetching of B_{block}.

4.6.2 MIC Specific Prefetching

The MIC architecture is peculiar about prefetching. The MIC architecture allows prefetch instructions to be issued concurrently with almost all other instructions. This ability allows many more prefetch instructions to execute than on both Nehalem and Sandy Bridge. To take advantage of this, a hierarchical, three stage pipeline is used. First a data element should be prefetched into the L2 cache from memory before use, then that element is prefetched into the L1 cache from the L2 cache, then the element should be processed and written back to memory. Since the caches have to handle two or more threads, prefetching from all the matrices is performed. The final version of GEMM for the MIC architecture has three streams set up that are executing this three stage pipeline. One for the $C_{slither}$ submatrix, another for the B_{block} submatrix and lastly for the $A_{slither}$ submatrix. This is how the blocking problem mentioned in Section 4.4 is handled. To summarize the L1 cache's purpose on MIC, prefetching grabs values from the L2 cache from these various submatrices prior to their use by the inner kernel and puts them in the L1 cache. The L1 cache is reduced to a 32 KB, software controlled buffer that is predominantly filled by prefetching instructions rather than explicit load instructions.

4.7 Inner Kernel

Possibly the most important part to any linear algebra routine is the inner kernel that performs the computations. For this reason, the highly tuned and optimized inner kernels of both GotoBLAS (for Nehalem) and MKL (for Sandy Bridge and MIC) were examined on the assembly level to help expose the most important features concerning prefetching and

register use. In [8], the inner kernel is given a few generic guidelines. In this research, a few more guidelines are discovered and appended to their list. It should be seen that the size of M_c and N_r are critical to the performance for the inner kernels. Using one half the available vector registers to hold values of $C_{slither}$ is the guideline in GotoBLAS. This is also used in MKL as well. The key observation when examining the inner kernels of MKL and GotoBLAS were that M_c is maximized and N_r is minimized giving a more rectangular shape to the register blocking than the description in [8] ($M_c \approx N_r$). The reasoning behind this register blocking scheme is that M_c is still small (from 8 to 16) which still enables good reuse of B_{block} and if N_r is minimized then $A_{slither}$ is reused the maximum amount of time in the L1 cache. Another benefit from this structure is that the TLB entries holding values of C will have better reuse than a smaller M_c value. With this rectangular structure, there is less stress put on the L2 cache because the elements from the B matrix are streamed more slowly to the execution units. Instead, there is more stress put on streaming the elements from the $A_{slither}$ submatrix which are already in the L1 cache. Even though maximizing M_c causes more load instructions to be issued, superscalar architectures are capable of hiding these extra loads behind the add and multiply instructions.

A helpful example might show how the inner kernel compute stage shown in Figure 4.1 is executed. Figure 4.4 magnifies this unrolled inner kernel. Using the Nehalem vector register set as an example, there are 16 registers holding two doubles each. Eight will be used for holding values of C and the remaining eight will be used to hold values from A and B. The registers are named v0, v1, v2, ... , v15 below. The pseudo-code in Listing 4.3 shows a basic inner kernel structure that is not unrolled for values $M_c = 8$ and $N_r = 2$.

From the top part of Figure 4.4, one can show that there are far more loads of elements in $A_{slither}$, in L1, per computation than of elements in B_{block}, in L2. Hence, the L1 cache is accessed far more frequently than the L2 cache. It can also be seen that there are even fewer memory references to elements of C per computation. The following equations use the expression, $2 \times m \times n \times k$, to count the number of computations computed for a matrix

```
set_all_registers_to_zero();
prefetch_next_elements_of_C_to_L1();
for(i = 0; i < Kc; i++) {
// prefetches future elements of B from L2 to L1
        prefetch_into_L1(bindex, i);

        // Step 1
        v4 = vector_load(bindex);
        v0 = vector_load(aindex);
        v1 = vector_load(aindex+2);
        v2 = vector_shuffle(v0, swap_pattern);
        v3 = vector_shuffle(v1, swap_pattern);
        v8 += v0*v4;
        v9 += v1*v4;
        v10 += v2*v4;
        v11 += v3*v4;
        // Step 2
        v0 = vector_load(aindex+4);
        v1 = vector_load(aindex+6);
        v2 = vector_shuffle(v0, swap_pattern);
        v3 = vector_shuffle(v1, swap_pattern);
        v12 += v0*v4;
        v13 += v1*v4;
        v14 += v2*v4;
        v15 += v3*v4;
        aindex += Mc;
        bindex += Nr;
}
// these memory locations have been prefetched
shuffle_v8_to_v15();
store_v8_to_v15_to_C();
```

Listing 4.3: Inner kernel with M_c maximized and N_r minimized. aindex and bindex refer to the auxiliary buffers holding the packed versions of A_{panel} and B_{block} respectively. v8 through v15 hold values that will be accumulated to the C matrix. Each vector_load() instruction loads 2 contiguous double precision values. Each vector_shuffle() instruction swaps the values and returns this new vector. This inner kernel structure applies to both Nehalem and Sandy Bridge while Listing 4.5 refers to the MIC inner kernel structure

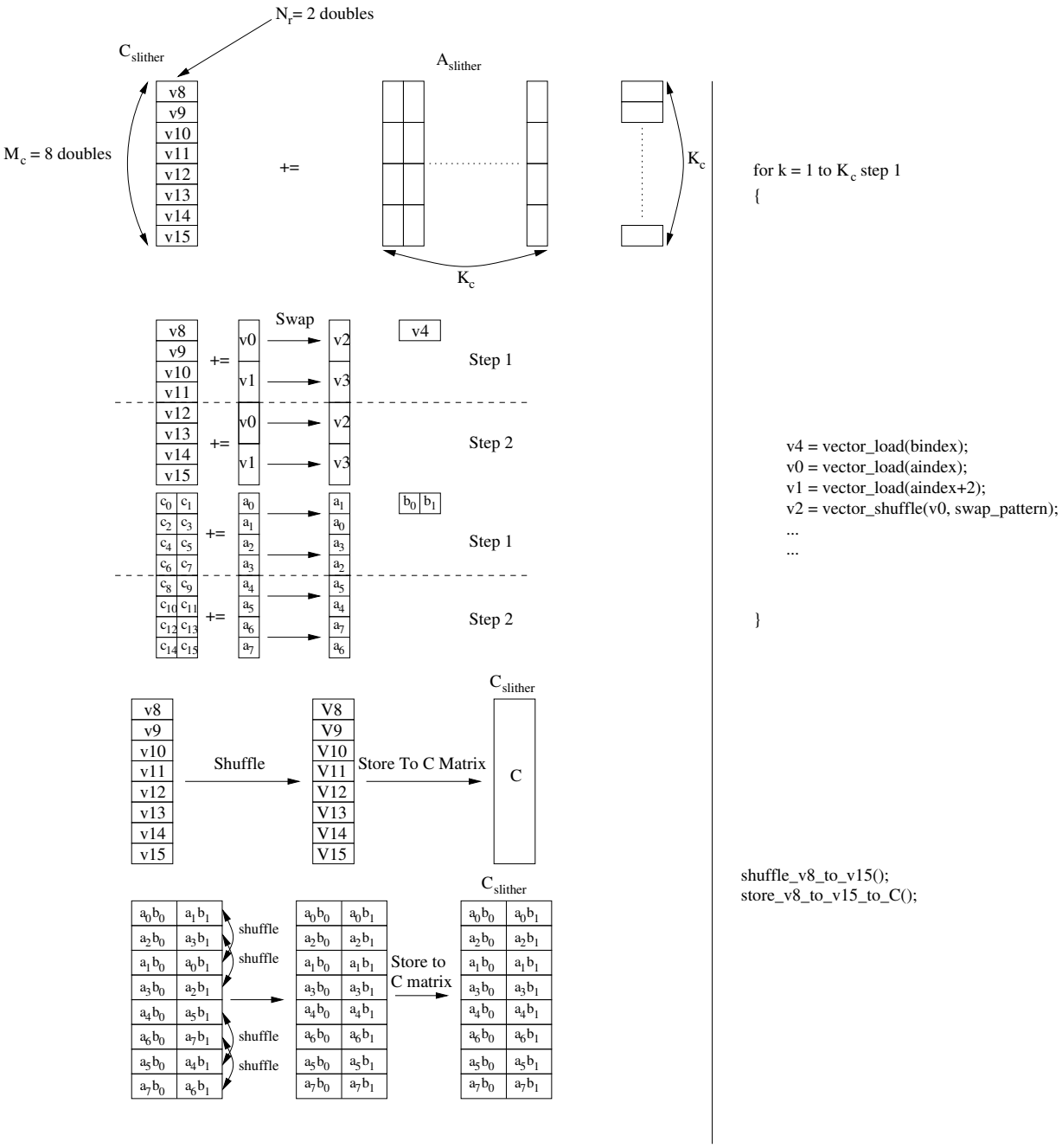

Figure 4.4: Illustration of the Compute Using Inner Kernel step from Figure 4.1. The code illustrated is from Figure 4.3.

multiply operation. The computations per L1 reference is given by

$$R_{L1} = \frac{2 \times N_r \times M_c \times K_c}{M_c \times K_c} = 2 \times N_r \text{ computations per L1 reference} \tag{4.4}$$

Table 4.4: Values of the blocking terms for GEMM and SYRK. The format is (single/double)

Architecture	M_c	K_c	N_c	N_r
Nehalem	8/8	256/256	256/128	4/2
Sandy Bridge	8/8	256/256	128/64	8/4
MIC	24/24	256/256	512/256	16/8

The computations per L2 reference is given by

$$R_{L2} = \frac{2 \times N_r \times M_c \times K_c}{N_r \times K_c} = 2 \times M_c \text{ computations per L2 reference} \tag{4.5}$$

The computations per memory reference is given by

$$R_{memory} = \frac{2 \times N_r \times M_c \times K_c}{N_r \times M_c} = 2 \times K_c \text{ computations per memory reference} \tag{4.6}$$

Equation 4.6 implies maximizing K_c while holding the restriction of Equation 4.1. This is how K_c is determined for all machines. Equations 4.4 and 4.5 offer an insight explaining why M_c should be maximized and N_r should be minimized. Since N_r is minimized, R_{L1} will be minimized. Similarly, if M_c is maximized, then R_{L2} will be maximized. This implies that the L1 cache will take the maximum number of requests per computation and the L2 cache will take the minimum number of requests per computation allowed by this cache and register blocking scheme. Because L1 latency is always smaller than L2, and L1 bandwidth is certainly greater than or equal to L2, this is ideal.

4.7.1 Nehalem Specific Kernel

For all architectures, the values of all important terms are in Table 4.4.

Particular features of Nehalem limited the use of intrinsics and compiler generated inner kernels. Hence, inline assembly was used instead. There were three reasons for this decision. All these points were observed by examining the assembly code of MKL and GotoBLAS.

1. The indexing for $A_{slither}$ and B_{block} caused code enlargement of the inner kernel.

2. The order of instructions is more important for this older architecture than for the other newer architectures.

3. The superscalar architecture cannot be exploited as well with regards to shuffle instructions if using intrinsics.

4. Particular registers give instructions that are only smaller.

The third point is the most important and critical. The superscalar nature of the Nehalem architecture allows up to six instructions to execute during each cycle[12]. Looking at Listing 4.3, there are four main instruction types that we would like to run concurrently.

1. Vector Load/Prefetch

2. Vector Shuffle

3. Vector Add

4. Vector Multiply

Fortunately, there exists separate pipelines for each of these four instruction types. Ideally, at least one add and one multiply instruction will execute each cycle. This would give the maximum computational throughput. The obstacle put forth by this architecture regards the shuffle instruction. Nehalem distinguishes between floating point shuffles and integer shuffles. Unfortunately, Nehalem runs floating point shuffles on the same execution unit as the multiply instruction. This conflict of the resource hinders the ability to execute a multiply instructions nearly every clock cycle. The work around is to use the separate execution unit that is dedicated to integer shuffles. All one has to do is ensure that byte order is preserved for both single and double precision values. Doing this allows all four instructions types to have their own independent execution unit. Using intrinsics restricted the vector registers to a type of single or double precision in higher level C code. The compiler refused to use a floating point vector register as an integer vector register without an explicit conversion instruction. Not even typecasting was allowed. Hence, assembly must be used.

```
# %xmm1 = %xmm1 + %xmm0; (2 operands)
movaps %xmm1, %xmm2 # (save the value of xmm1 if needed again)
addps %xmm0, %xmm1

# %xmm2 = %xmm1 + %xmm0; (3 operands)
addps %xmm0, %xmm1, %xmm2
```

Listing 4.4: Using two operand vector instructions vs. three operand vector instructions. With two operands, one of the source registers is overwritten (%xmm1) and must be copied before hand if its value is needed again. With three operands, a separate destination register is used (%xmm2) and both source registers keep their value after the add operation.

The indexing of the elements from $A_{slither}$ and B_{block} also played a roll in using assembly. For the Nehalem architecture, using constant memory indexing with values between -0x80 and 0x70 will create instructions that are four bytes long instead of five bytes long. While this seems tedious, four byte instructions lend themselves well to instructions fetches that are aligned on 16 byte addresses.

Performance for Nehalem was far more dependent on the instruction order than for Sandy Bridge or MIC. So compiler generated assembly was always slower than good hand coded assembly. One major reason for this is that Nehalem uses two register operands for vector instructions with one source operand always being overwritten (destroyed). The compiler always generated unnecessary move/copy instructions which MKL and GotoBLAS avoided.

The most bizarre feature seen from examining MKL and GotoBLAS was that lower register numbers allowed for instructions (add, multiply, shuffle) to be smaller in size than if using larger register numbers. For example, using registers xmm0–xmm7 instead of xmm8–xmm15 allowed instructions to be smaller by exactly one byte. Again, smaller instructions are always better because of the ability to fetch more instructions, but this research offers no reason other than observation about why using smaller numbered registers gives smaller instructions.

4.7.2 Sandy Bridge Specific Kernel

The Sandy Bridge inner kernel was written using the intrinsics in Listing 4.1. Although the Sandy Bridge and Nehalem architecture feature similar superscalar architectures, the Sandy Bridge is more amenable to GEMM kernels. It has two load execution units instead of one, and also has a separate floating point shuffle execution unit that Nehalem does not. These new execution units allowed the compiler to generate inner kernels which are on par with MKL. Another improvement over Nehalem is the use of three register operand vector instructions. This allowed the source operands to retain their values, so unnecessary move/copy instructions were not created.

4.7.3 MIC Specific Kernel

The MIC inner kernel is fundamentally different than the Sandy Bridge and Nehalem architectures because MIC lacks the superscalar abilities of both those architectures. MIC is also different in that Fused Multiply-Add (FMA) instructions are used instead of separate multiply and add instructions. To achieve maximum performance on the MIC architecture, the processor must execute one FMA instruction every clock cycle. Because MIC cannot execute multiple vector instructions at one time, using shuffle instructions becomes incompatible with achieving good performance. Also, load instructions must be kept to a minimum. MIC offers a workaround to allow an ideal inner kernel structure of unrolled FMA instructions. When executing any three operand arithmetic instruction on MIC, a temporary shuffling is allowed for one source register which they call swizzling [21]. This temporary shuffling only lasts for that single instruction and does not change the source register permanently. This swizzling is how shuffling is executed with no cost in terms of latency. MIC also offers a three cycle latency to the L1 cache. As discussed in Section 3.3, multiple threads are run on a single core in a round robin fashion. Each cycle a different thread context executes an instruction. If three or more threads are running on a single core, then the three cycle L1 latency can be hidden by this scheduling. Using three or more threads per core presents a problem though. All threads will share the private L1 and L2

cache which restricts the values of K_c and N_c. To help alleviate this problem, the software prefetching mechanism as shown in Figure 4.3 is used so relevant data is in the L1 and L2 caches. All these attributes lead up to an inner kernel shown in Listing 4.5 with the swizzle operation illustrated in Figure 4.5.

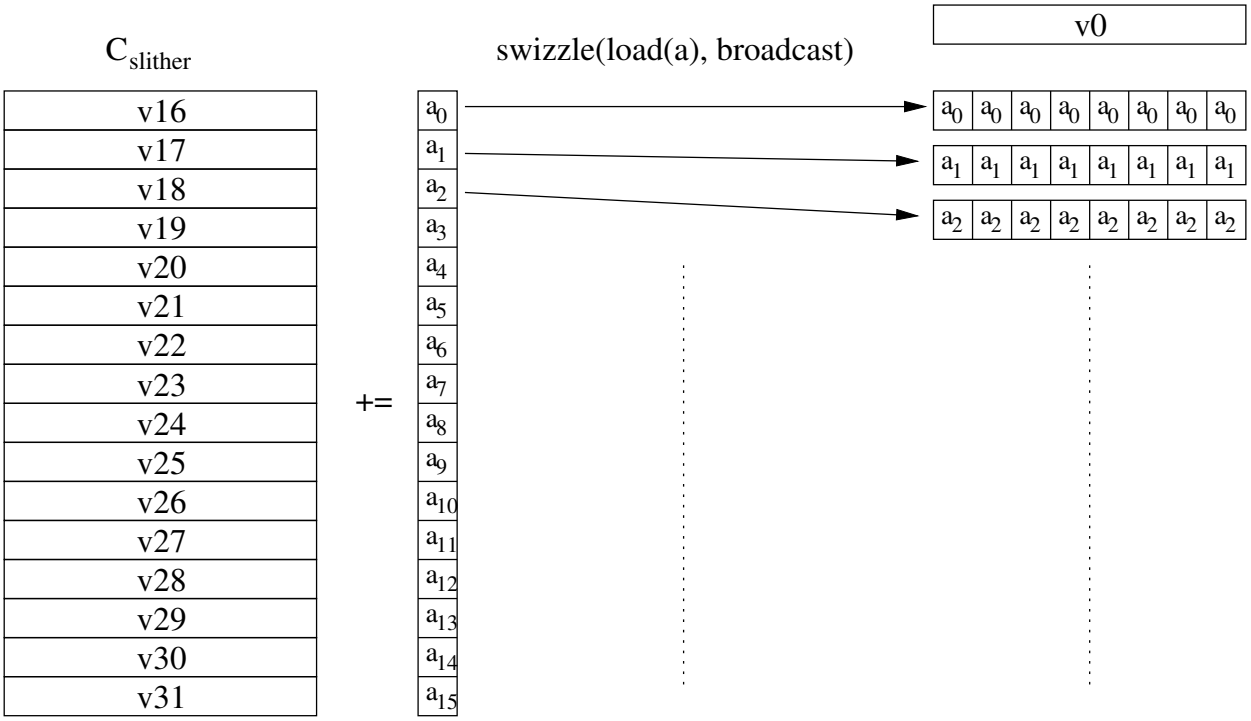

Figure 4.5: The MIC swizzle operation takes a memory location and returns a vector with the value at said memory location broadcast to all elements in the return vector.

4.8 TRSM Development

In the sections below, the ideas from GEMM are applied to the development of TRSM. These concepts include blocking, packing, and inner kernel structure. This section will also show how GEMM is reused to do the bulk of the work in the TRSM kernel.

```
set_all_registers_to_zero();
prefetch_next_elements_of_C_to_L1();
for(i = 0; i < Kc; i++) {
// prefetches future elements of A and B from memory to L2
        prefetch_into_L2(bindex+64);
        prefetch_into_L2(aindex+64);
// prefetches future elements of A and B from L2 to L1
        prefetch_into_L1(bindex+16);
        prefetch_into_L1(aindex+16);

        v0 = vector_load(bindex);
        v16 = v16 + v0 * swizzle(load(aindex),broadcast);
        v17 = v17 + v0 * swizzle(load(aindex+1),broadcast);
        v18 = v18 + v0 * swizzle(load(aindex+2),broadcast);
        v19 = v19 + v0 * swizzle(load(aindex+3),broadcast);
        v20 = v20 + v0 * swizzle(load(aindex+4),broadcast);
        v21 = v21 + v0 * swizzle(load(aindex+5),broadcast);
        v22 = v22 + v0 * swizzle(load(aindex+6),broadcast);
        v23 = v23 + v0 * swizzle(load(aindex+7),broadcast);
        v24 = v24 + v0 * swizzle(load(aindex+8),broadcast);
        v25 = v25 + v0 * swizzle(load(aindex+9),broadcast);
        v26 = v26 + v0 * swizzle(load(aindex+10),broadcast);
        v27 = v27 + v0 * swizzle(load(aindex+11),broadcast);
        v28 = v28 + v0 * swizzle(load(aindex+12),broadcast);
        v29 = v29 + v0 * swizzle(load(aindex+13),broadcast);
        v30 = v30 + v0 * swizzle(load(aindex+14),broadcast);
        v31 = v31 + v0 * swizzle(load(aindex+15),broadcast);
        aindex += Mc;
        bindex += Nr;
}
// these memory locations have been prefetched
store_v16_to_v31_to_C();
```

Listing 4.5: MIC inner kernel pseudo code. Notice all the memory references and prefetches. The swizzle function returns a vector with all elements the same as the element at location of aindex plus the offset.

4.8.1 Blocking

The TRSM blocking scheme can be seen in Figure 4.6. A_{block} can fit into the L2 cache while B_{block} can fit into the L1 cache. At first, it seems that B_{block} isn't reused, but later it will be shown that the inner kernel reuses elements from B_{block} and needs them packed in L1. A_{block} is reused $\frac{n}{N_c} - 1$ times.

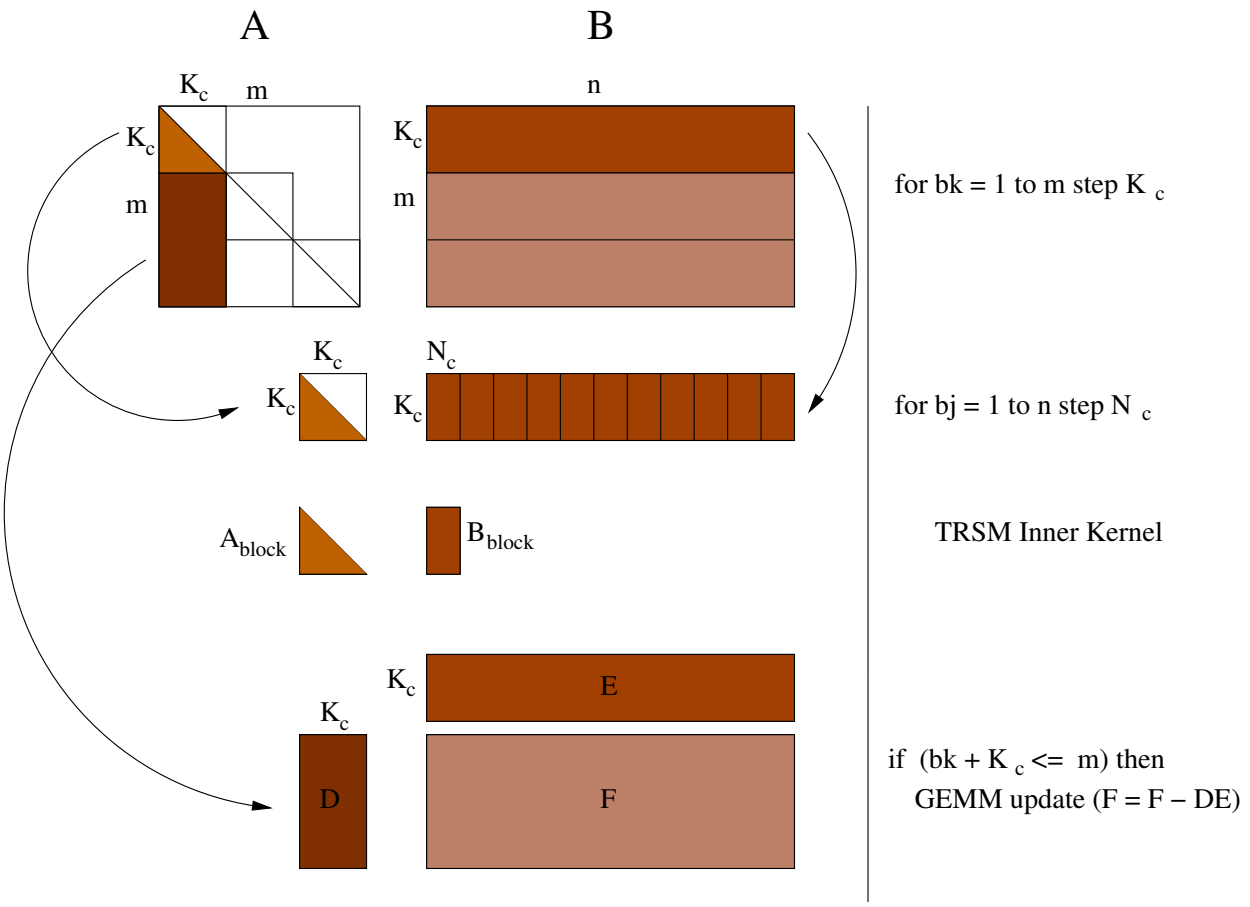

Figure 4.6: TRSM blocking scheme used in this research.

Table 4.5: Values of the blocking terms for TRSM. The format is (single/double)

Architecture	N_c	K_c
Nehalem	8/4	256/256
Sandy Bridge	16/8	256/256
MIC	32/16	256/256

4.8.2 Packing

The use of packing the A_{block} submatrix is similar to GEMM but there is an added benefit of packing for TRSM. If you pack A_{block}, then the minimum number of divisions are performed, and the problem has the same structure as a GEMM (fused multiply-adds). Looking at Equation 2.2, there is exactly one division per element b_{ij}. By executing $a_{iir} = \frac{1}{a_{ii}}, i \in \{1, 2, ..., m\}$, which is one division per diagonal element, the new equation

$$b_{ij} = a_{iir}\alpha \left(b_{ij} - \sum_{n=1}^{i-1} a_{in} b_{nj} \right) \tag{4.7}$$

can be used which only uses multiply and add instructions. Since there is one division per a_{ii} where $i \in \{1, 2, ..., m\}$, there are exactly m divisions used. So by storing $\frac{1}{a_{ii}}$ instead of a_{ii} in the auxiliary buffer holding A_{block}, the problem reduces to only multiply and add instructions. Avoiding divisions is beneficial for computational throughput because in Intel systems (and many others) floating point division is an expensive operation that cannot be pipelined and has a long latency compared to multiply (30–50 cycles for division vs. 5 cycles for multiplication).

4.8.3 Inner Kernel

The TRSM inner kernel uses many of the same principles as GEMM. For instance, half of the registers are dedicated to holding values of the destination matrix. The TRSM inner kernel itself is a series of small triangular solves followed by small matrix-matrix multiplies. Looking at Figure 4.8, it can be seen why B_{block} is stored in L1 and how the problem is a

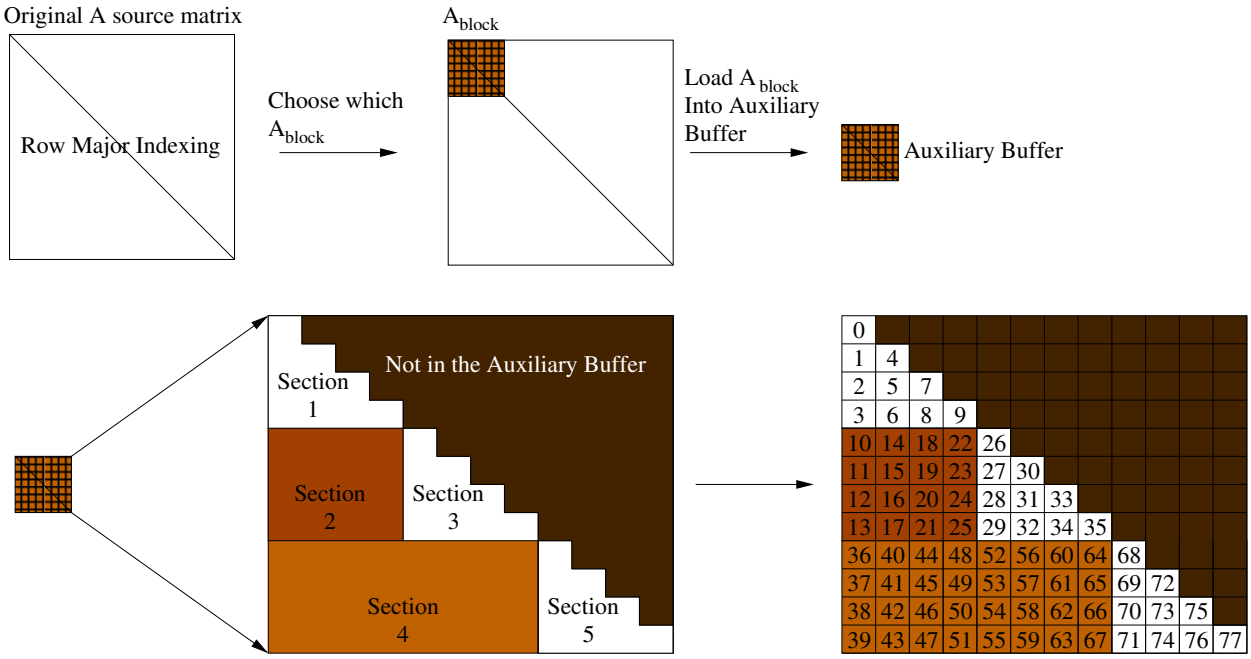

Figure 4.7: TRSM packing of A_{block}. Each section is contiguous with section 1 being first, section 2 begin second, etc. The diagonal elements are holding the values $\frac{1}{a_{ii}}$ instead of a_{ii}. The reason for this is described in Section 4.8.2.

series of small triangular solves and matrix-matrix multiplies. The first step is executing the small GEMM on section 2 and then executing the small TRSM on section 3. From Equation 4.7 it should be noted that to calculate an element $b_{ij}, i = m_i$, all previous rows of B, where $1 \leq i < m_i$ must be calculated. These previous rows are stored in B_{block} which resides in L1 cache for fast access.

4.9 SYRK Development

Since SYRK is nearly identical to GEMM with regards to packing, blocking, and prefetching, only the inner kernel is discussed. As mentioned before, SYRK only updates either the lower or upper half of the destination matrix. With this restriction, a special modified GEMM inner kernel was used for the diagonal part of the destination matrix. Otherwise, the normal GEMM inner kernel is used. This can be seen in Figure 4.9. If $C_{slither}$ is triangular instead

of rectangular, then the special inner kernel. The only difference between the GEMM and SYRK inner kernels is that a special temporary buffer is created to aid in the transfer and calculation of the destination matrix. This way only the upper or lower half of the destination matrix is updated which follows the definitions in BLAS.

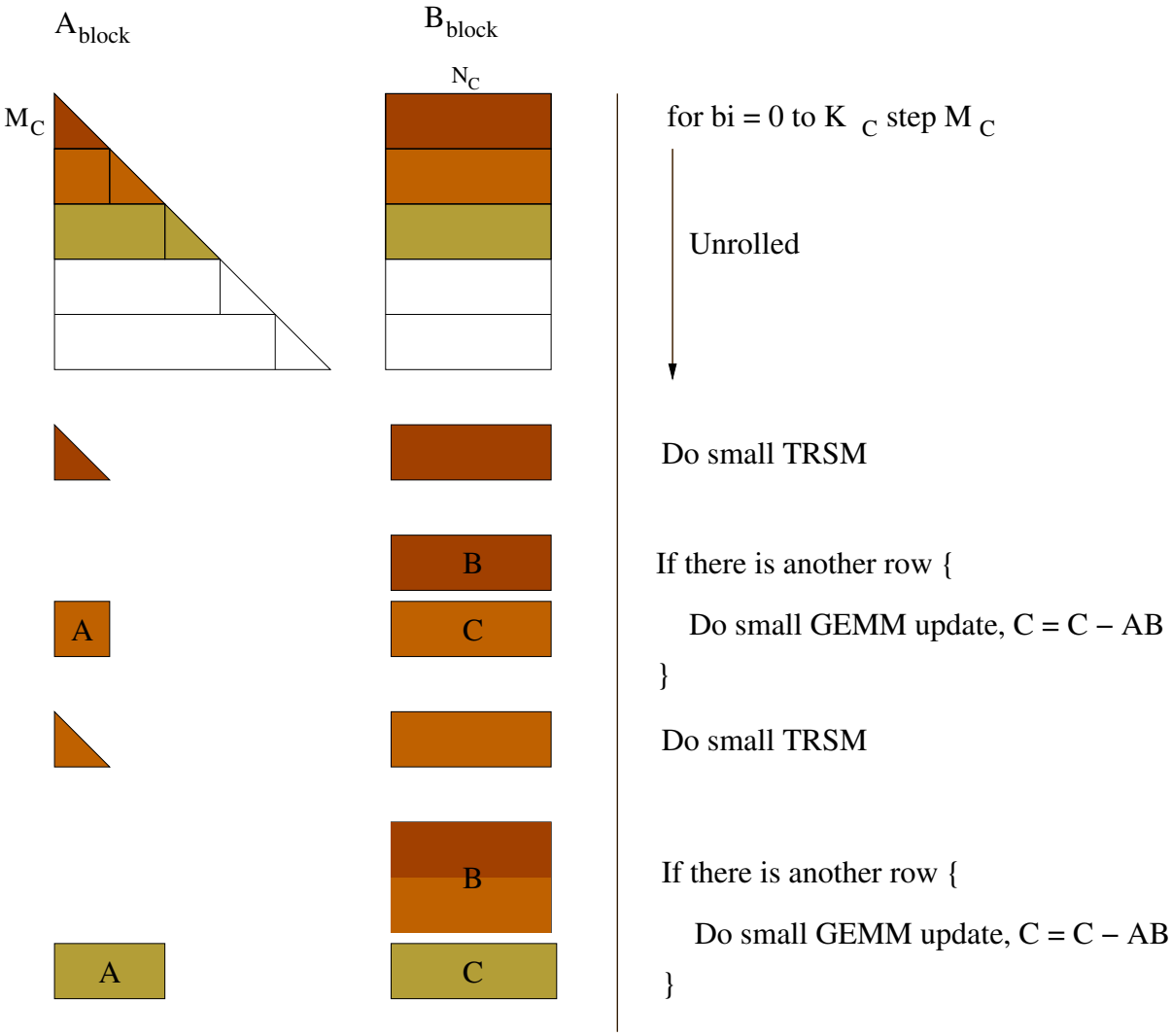

Figure 4.8: TRSM inner kernel. Elements of B_{block} are reused for each small GEMM update suggesting a packed auxiliary buffer should be used. Notice that even in the TRSM kernel, GEMM comprises a significant proportion of the computational time compared to the actual TRSM part of the kernel.

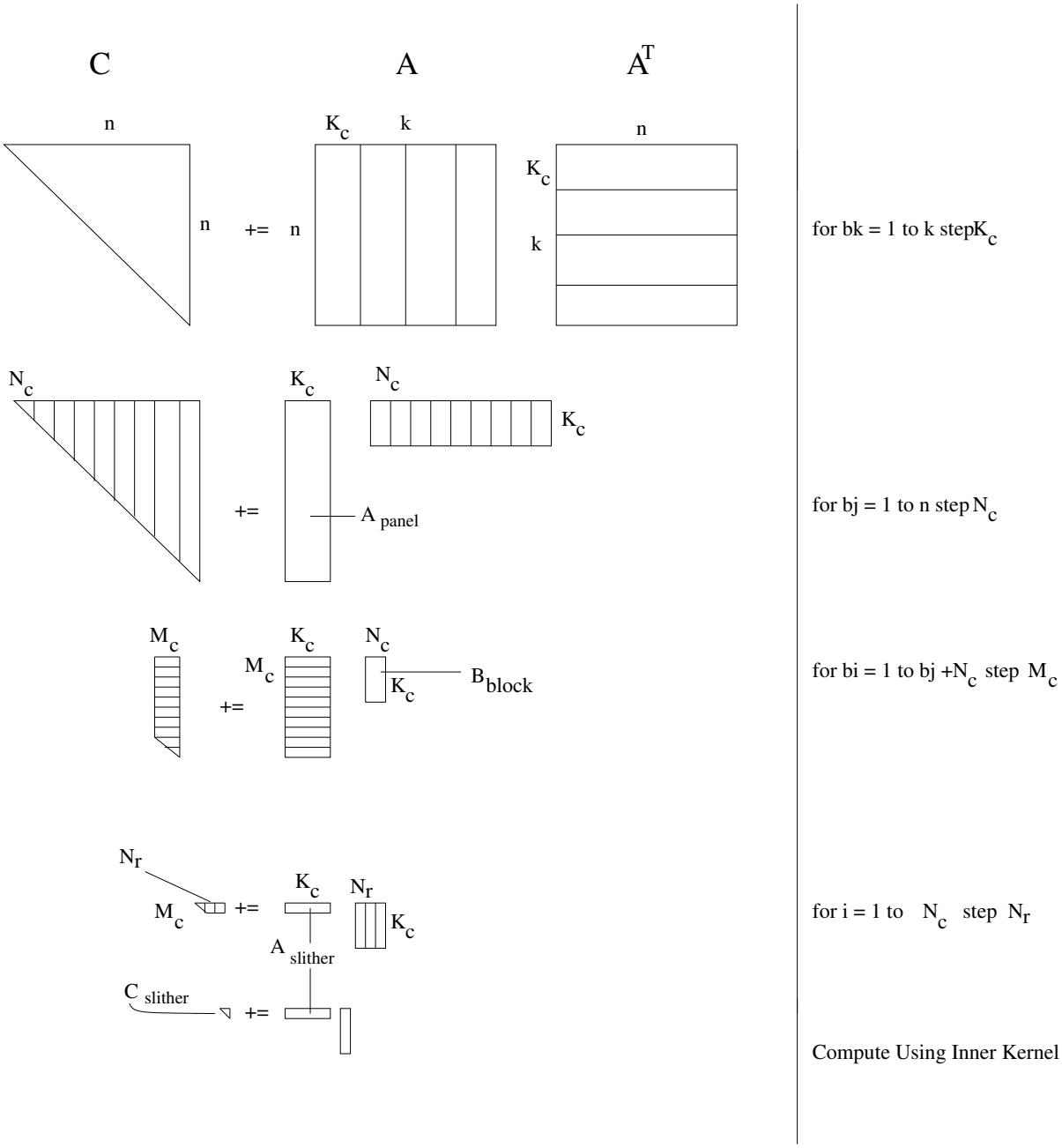

Figure 4.9: SYRK blocking method which is nearly identical to GEMM.

4.10 POTRF Development

The POTRF routine is a LAPACK routine which spends the majority of its runtime in the TRSM and SYRK routines. Its algorithm is described in the LAPACK definitions

```
for bk = 0 to n step NC
        POTF2 (A_11 = U_11^T U_11)
        if(part of matrix remains to be decomposed) {
                TRSM (A_12 = U_11^T U_12 → U_12 = (U_11^T)^-1 A_12)
                SYRK (A_22 = U_12^T U_12 + U_22^T U_22 → U_22^T U_22 = A_22 − U_12^T U_12)
        }
endfor
```

Listing 4.6: LAPACK POTRF algorithm. NC is block size used.

and is rewritten here for convenience in Listing 4.6. The algorithm is a blocked algorithm which exploits the usage of the Level 3 BLAS routines. By using the Level 3 BLAS routines, the entire algorithm benefits by spending the majority of its running time in these computationally efficient routines. The basic idea of this blocked algorithm is to calculate a small Cholesky block, and then update the rest of the matrix using Level 3 BLAS routines and recursively doing this until the entire matrix is decomposed. A step of the algorithm is shown in Figure 4.10 for convenience. The POTF2 routine is a small routine which

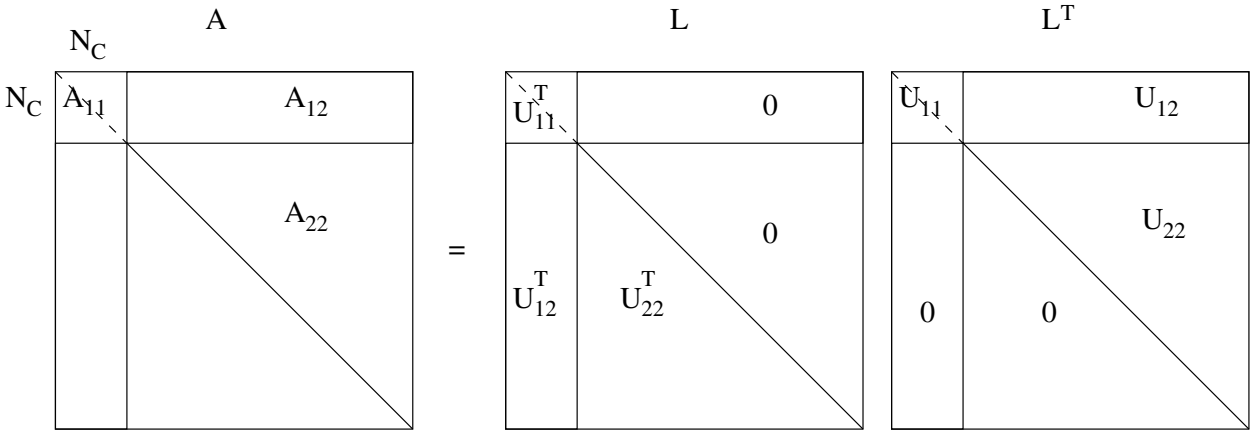

Figure 4.10: Cholesky (POTRF) decomposition corresponding to the algorithm in Listing 4.6

calculates the Cholesky decomposition for a small square submatrix. Its implementation is

not a main focus because of the little percentage of time spent in this part of the algorithm when run.

4.11 Single Core Results

In the graphs below, the performance results for the serial versions of all the linear algebra routines are shown. All the graphs' max y-axis value is set to the peak for a single core of the corresponding machine they were run on. Every point on each graph represents an average of five runs for the given size or core count. The results in Figures 4.11, 4.12, 4.13, and 4.14 all show the produced Level 3 BLAS codes achieve at least 90% of peak performance. The POTRF codes all get at least 85%, near 90% for the double precision, with performance results on par with MKL.

On MIC, the results are for a single core with four threads running. This is a better performance indicator for a core of MIC than simply a single thread running on the lone core. The results for MIC can be seen in Figures 4.15 and 4.16. The Level 3 BLAS routines' performance is near 70% with DGEMM hitting almost 80% and SGEMM hitting 75%. MKL's SGEMM routine approaches an impressive 90% and DGEMM achieves 85%. It should be noted that while the programmed DGEMM and SGEMM routines did not near MKL's performance like they did on Sandy Bridge and Nehalem, the other two Level 3 BLAS routines did achieve on par with MKL, as well as the SPOTRF and DPOTRF routine.

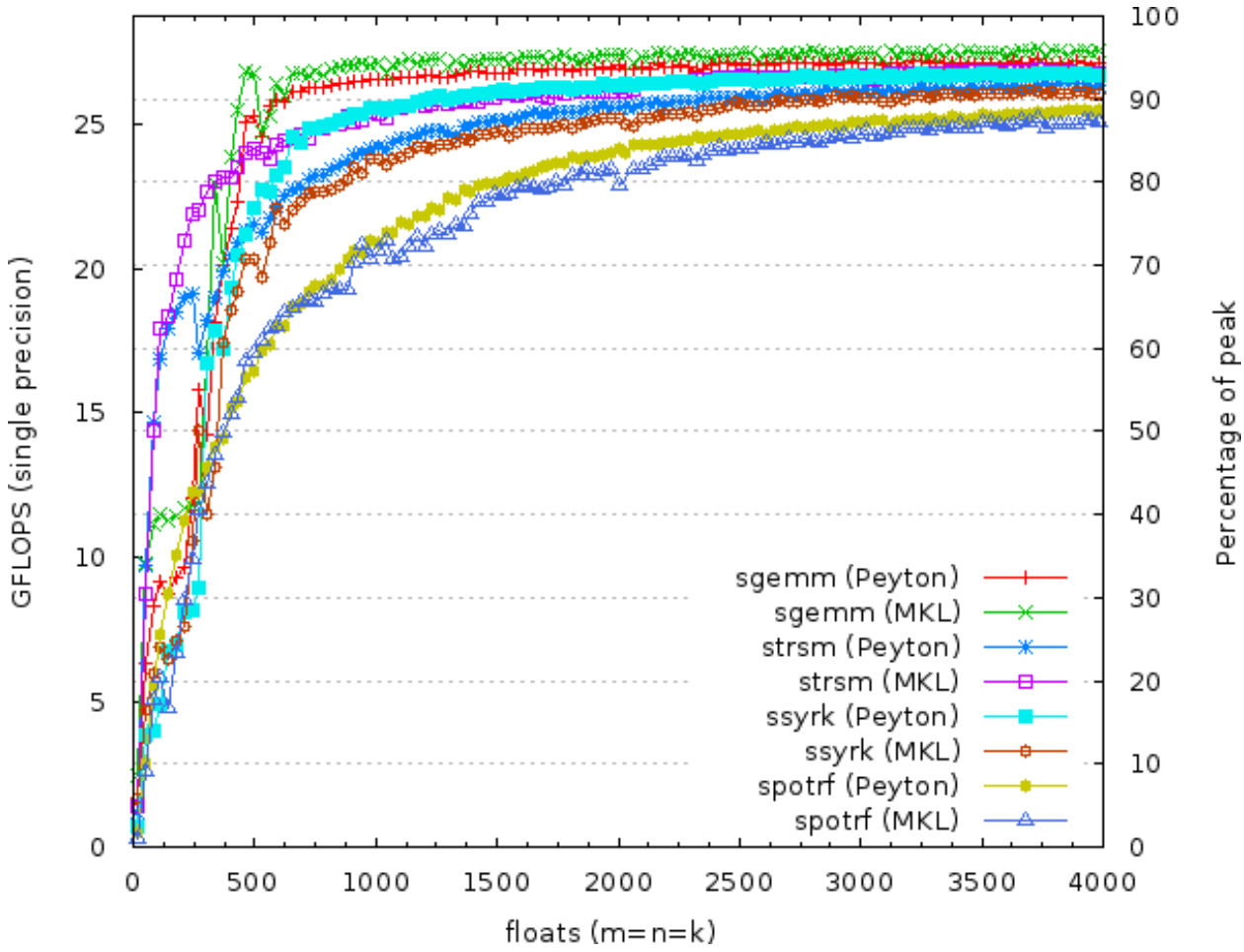

Figure 4.11: Single precision, single core for the Nehalem architecture.

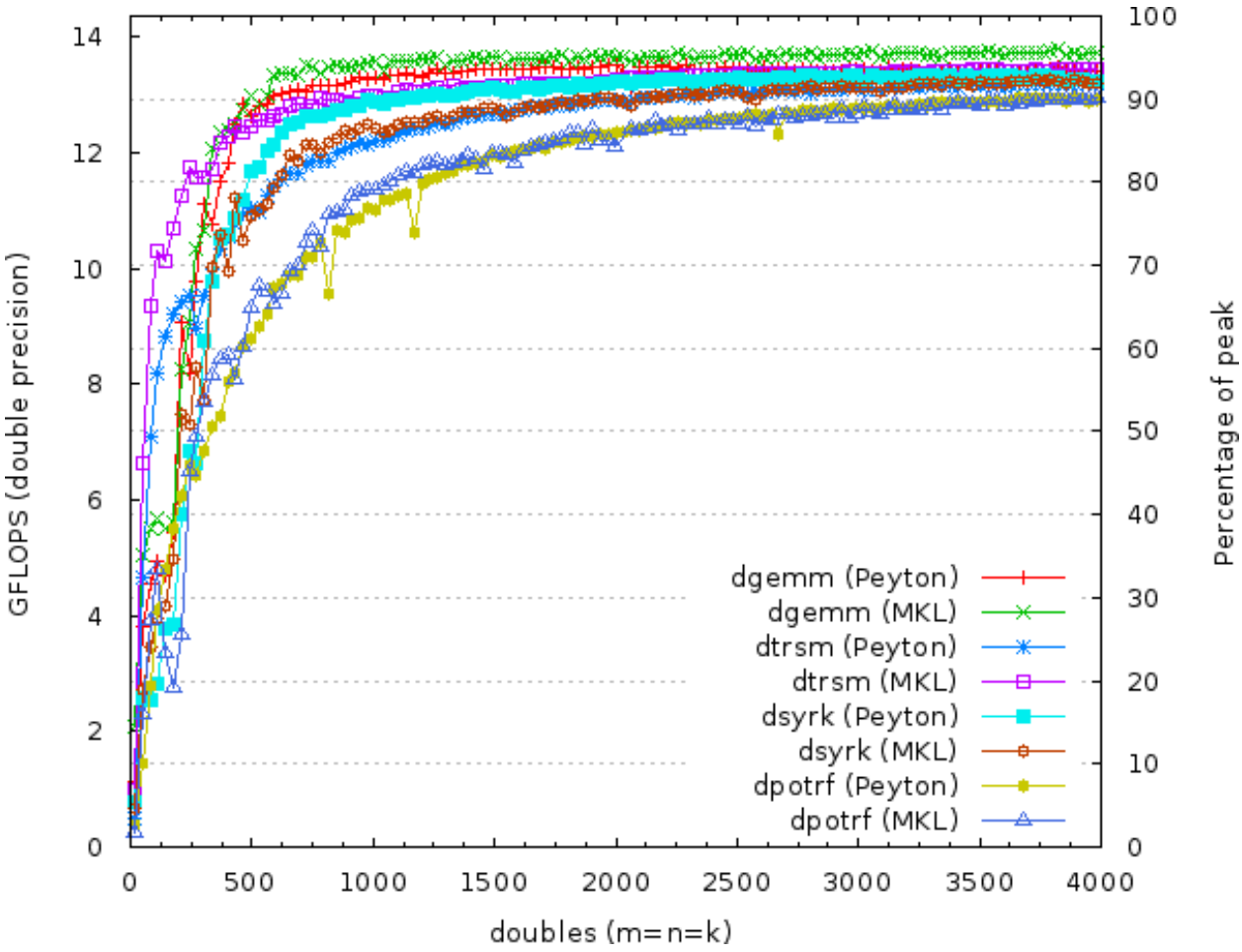

Figure 4.12: Double precision, single core for the Nehalem architecture.

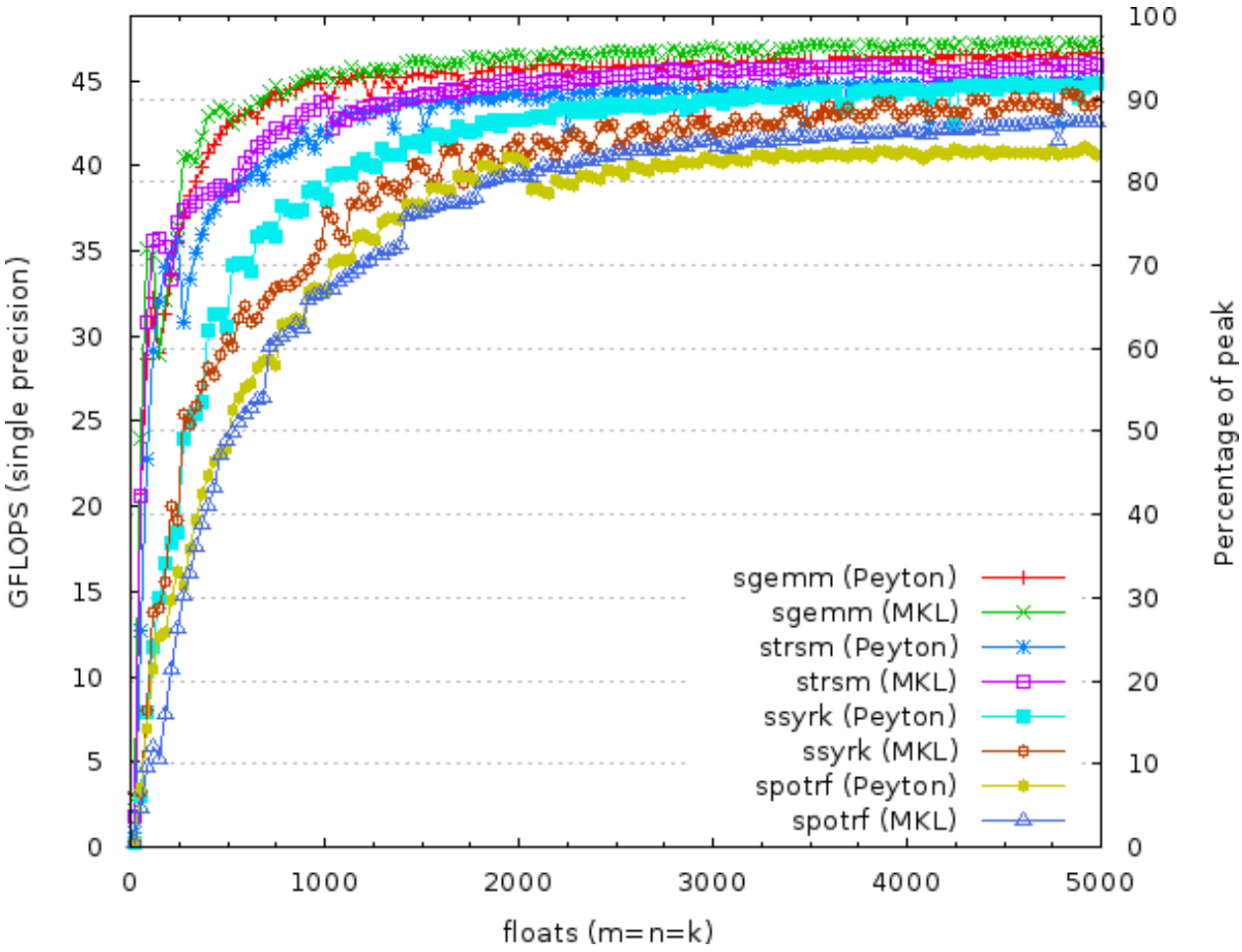

Figure 4.13: Single precision, single core for the Sandy Bridge architecture.

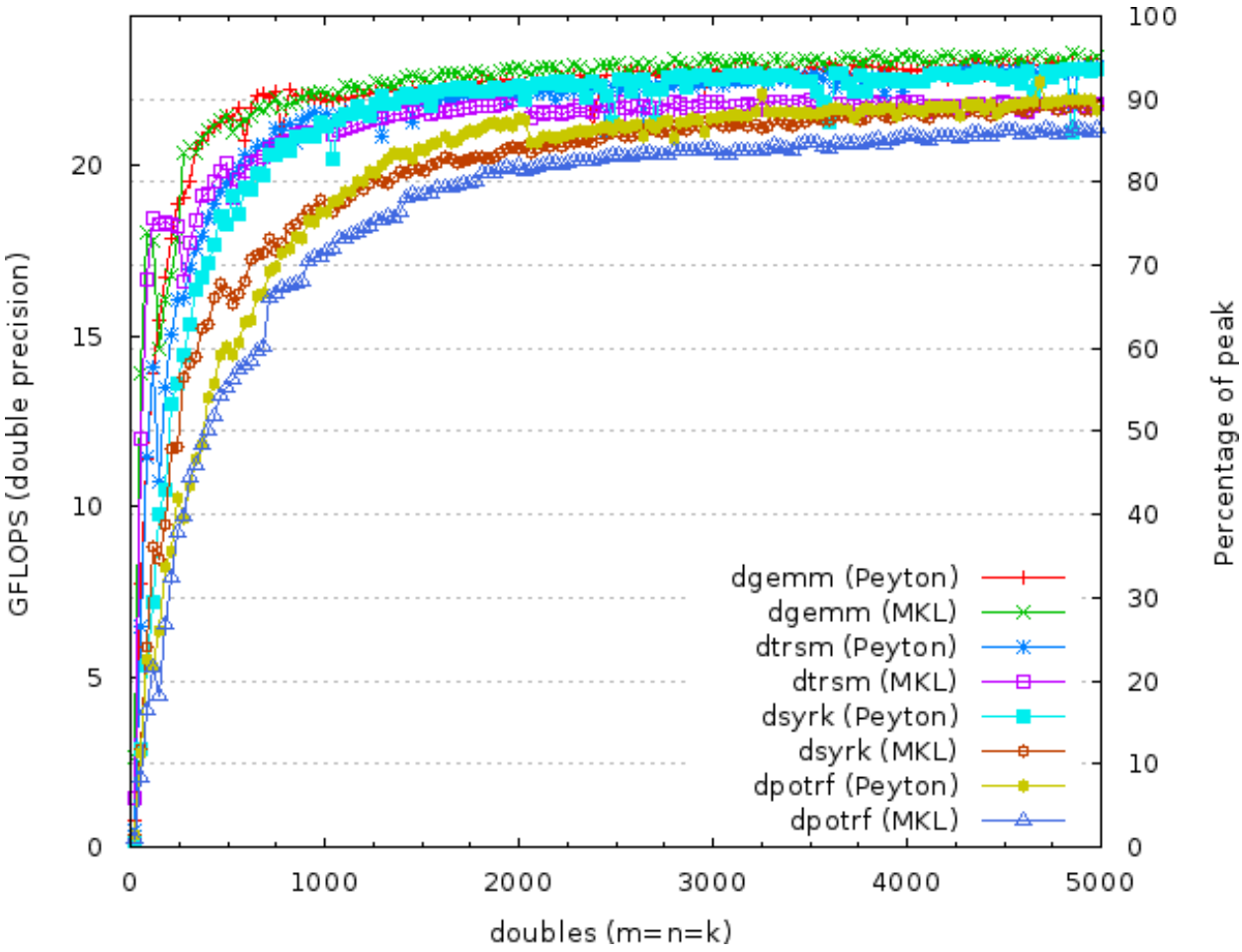

Figure 4.14: Double precision, single core for the Sandy Bridge architecture.

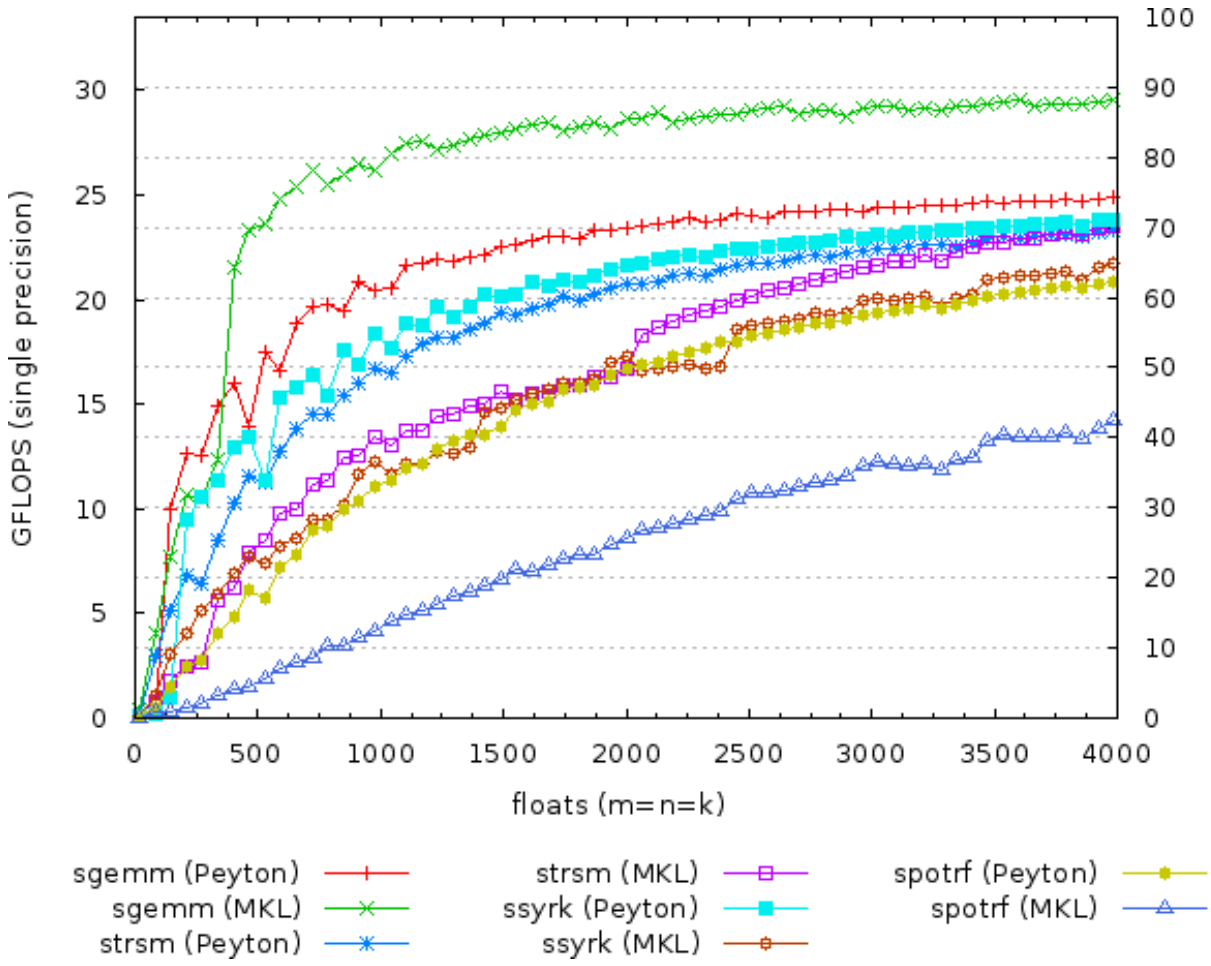

Figure 4.15: Single precision, single core for the MIC architecture.

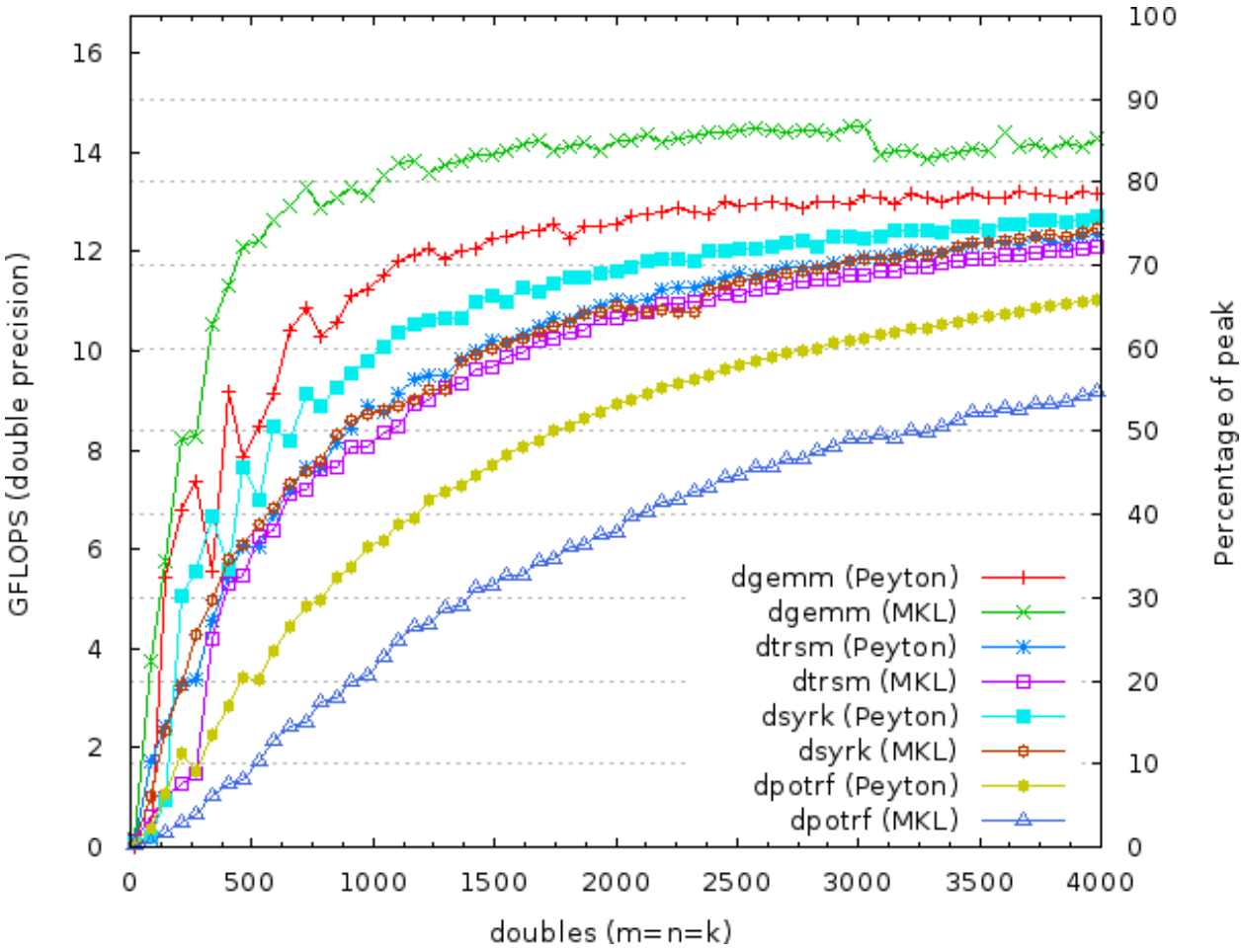

Figure 4.16: Double precision, single core for the MIC architecture.

Chapter 5

Parallel Languages

This section offers an overview of the parallel languages used to implement the routines. The four languages discussed are OpenMP (Section 5.1, Pthreads (Section 5.2, Intel Cilk (Section 5.3), and Intel TBB (Section 5.4. All the languages described below are shared-memory parallel languages.

5.1 OpenMP

OpenMP is a collection of compiler directives, library routines, and environment variables that control threads for a shared-memory system [22]. This API allows multiple threads to execute a section of code concurrently. A popular use of OpenMP is to parallelize a for-loop with each thread taking a chunk of the iteration space. This corresponds to the popular `#pragma omp parallel for` compiler directive for OpenMP. Another way to use OpenMP is to fork threads in a parallel region and use their identifiers (thread 0, thread 1, thread 2, etc.) as keys to determine which data that thread works on. A reason to bring up these examples of OpenMP is to show its relative ease of use. Only one line of extra code above a for-loop is all one needs to parallelize their code. This programming model was created by the HPC community in an effort to help ease the programming headaches caused by lower level programming models with respect to portability and programming difficulty [23].

The primary fork-join model used by OpenMP is illustrated in Figure 5.1. This model

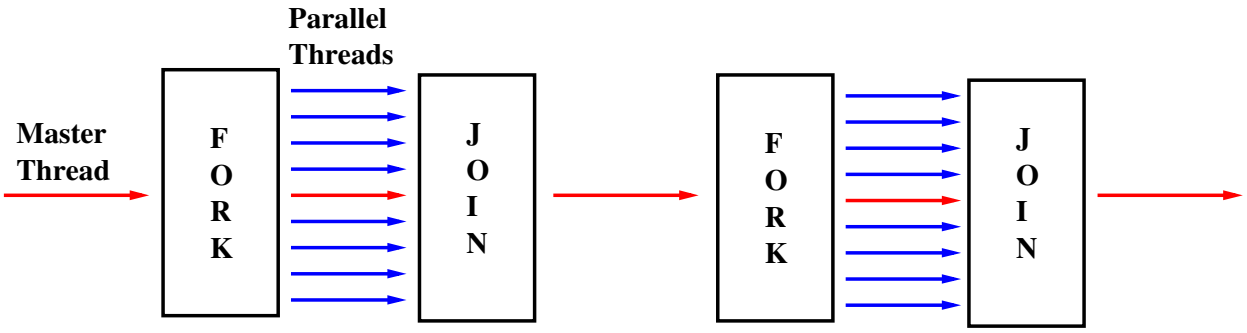

Figure 5.1: The fork-join multithreaded model.

shows the fundamental idea behind shared memory systems. One master thread will execute a serial section. This master thread then forks off multiple threads to execute a new parallel region that all threads will execute concurrently. If there exists enough physical cores, the threads will each get their own core and execute in parallel. It is this parallel execution that will give speed up over executing on a single core.

Concerning the linear algebra routines in this research, the most important factor to performance involved the environment variable `GOMP_CPU_AFFINITY` This environment variable specifies which threads map to which cores. If this variable is not set, often times the system will not utilize all the cores. This will be talked about again later, but its importance cannot be emphasized enough.

5.2 Pthreads

Pthreads (Posix Threads) [24] is the lowest level multithreaded programming model used in this research. It offers mutexes, condition variables, semaphores, and explicit thread creation and destruction through function calls. With so much power granted to the user, albeit at a costly programming price, this model would appear to always have a best solution to any problem. A problem with this assumption is the ability to program parallel sections of code correctly and efficiently is sometimes hard to attain. These difficulties can lead to hours and hours of debugging and tuning. In this research, an implementation of a

fork-join thread pool is used similar to OpenMP. This thread pool offers similar features of OpenMP including thread number assignment, changing the number of threads run on a particular parallel section, and barriers. A benefit of using this pool structure is that threads are created only once and last until the pool is destroyed. This avoids the unnecessary creation and destruction of OS threads between parallel BLAS calls. Again, concerning the linear algebra routines, Pthreads also offers a way to map threads to certain cores using the `pthread_setaffinity_np()` function.

5.3 Intel Cilk

Intel® Cilk [25] is an extension to C and C++ that offers a quick and simple way to improve performance on multicore systems. There are only three new keywords (`cilk_for`, `cilk_spawn`, and `cilk_sync`) for multicore solutions and a simple array notation language to exploit data parallelism. In this research only the keywords are used with the intent to mimic OpenMP's `#pragma omp parallel for` and `#pragma omp barrier` directives. Intrinsics are used for data parallelism instead of the Cilk array notation to provide a fair comparison. One major difference between Cilk and Pthreads or OpenMP is the scheduling used by the runtime environment. Intel Cilk uses a dynamic task-stealing runtime system to help with load balancing, and there is no explicit way to ensure all core utilization because thread affinity does not exist for Intel Cilk. Unfortunately, at the time of this writing, the Intel Cilk libraries were not properly working on the MIC architecture and are not included in the MIC results.

5.4 Intel TBB

Intel® Threaded Building Blocks (TBB) [26] is the last language explored in this research. TBB is a library of classes and routines for parallel constructs. An example of a parallel construct would be the `parallel_for` class offered to parallelize a typical for-loop where every iteration is independent of each other. It offers far more than Cilk in terms of

programming power and pre-made parallel algorithms (i.e., `parallel_sort`). TBB programs are written in C++ and use highly templatized class constructs. This style of programming offers a lot of power and flexibility with minimal coding verbosity. Similar to Cilk, it uses a dynamic, task-stealing runtime system as well for load balancing. The biggest difference between TBB and the other languages is its reliance on C++. Although a scheduler exists that emphasizes thread affinity, it does not give any leeway for choosing which threads run on which cores. So for Intel TBB, there is no guarantee that all cores will be utilized, only that threads will not migrate to different cores. Similar to Cilk, at the time of this writing, the Intel TBB libraries were not properly working on the MIC architecture and are not included in the MIC results.

Both Cilk and TBB's runtime system provide flexibility not seen in OpenMP. The goal of Cilk and especially TBB is the ability to write arbitrarily complex parallel tasks with arbitrary synchronization. This paradigm focuses on the parallelism rather than the implementation of the parallelism. This paradigm abstracts away OS threads and instead focuses on tasks that are mapped onto threads by the runtime system. This abstraction also takes care of load balancing by having OS threads "steal" tasks so threads, and hence cores, are not idle.

For the routines programmed in this research, the parallelism is simple and easy to implement. Load balancing and mapping of tasks to threads reduce to simple partitioning of the destination matrix and assigning these partitions (the computational tasks) to threads using a one-to-one mapping. Even when synchronization is implemented, it is again simple only requiring barriers. With this high data independence and easy load balancing, the complicated dynamic runtime systems for Cilk and TBB aren't utilized and can only hinder performance.

Chapter 6

Parallel Implementations

This section will focus on the details and issues of implementing the linear algebra routines for shared-memory multicore systems. For all routines, the common approach of partitioning the destination matrix into independent submatrices to avoid explicit synchronization is chosen.

6.1 Language Details

Optimizing the parallel languages for these routines was easy because of their embarrassingly parallel nature. Each language had an easy way to partition the C matrix and assign threads to each section. OpenMP's parallel section pragma was used to create n threads with identifiers from 0 to $n-1$. Each thread is then assigned an independent section of the C matrix according to this thread identifier. For synchronization in OpenMP, the `#pragma omp barrier` was used to synchronize all threads within the parallel region. Pthreads offered the exact same paradigm as OpenMP but for synchronization, an explicit barrier using condition variables was coded because no pragma's exist like they do in OpenMP. TBB and Cilk offered parallel-for constructs which schedules different chunks of iterations to threads. A trick used for this was to use a for-loop that had an iteration space from 0 to $n-1$ and then assigned threads only one iteration. This method is somewhat contrary to the purpose of TBB and Cilk but offered the best results. These languages are implemented for for-loops with many iterations in mind to take advantage of complicated dynamic load balancing techniques which

are unnecessary for these linear algebra routines. Synchronization in TBB and Cilk was also implemented as separate parallel-for sections because these languages provide an implicit barrier between parallel-for sections.

Controlling exactly the number of threads used and explicitly setting their hardware affinity showed itself as a paramount issue for parallel performance. OpenMP and Pthreads both offer easy ways to control both of these variables. TBB and Cilk only offered ways to control the number of threads. With no real way to control hardware affinity, threads would often be assigned to the same core and hence, no speedup. This problem arises with OpenMP and Pthreads as well when affinity and thread count are not explicitly accounted for. The only way to control hardware affinity in TBB and Cilk was to use Pthreads functions within a parallel-for section. Doing this allows each thread to set its own affinity based on the iteration number (from 0 to $n-1$). While this way did not create problems, it is not desirable to use different parallel languages within each other.

6.2 Hardware Details

Let N_{cores} be the number of cores for an architecture and let $N_{threads}$ be the number of threads executing the linear algebra problem concurrently. Then physically, the approach taken for the Nehalem and Sandy Bridge architectures is to have $N_{cores} = N_{threads}$. Using hardware affinity (if allowed by the parallel language), assign one thread to one core statically and allow all threads to run to completion. Each thread will have its own submatrix to compute. For the MIC architecture, the approach taken is to have $N_{threads} = 4 \times (N_{cores} - 1)$. The only difference is four threads are created per core. The use of $N_{cores} - 1$ is because a specific core is not used in the MIC architecture to allow the OS and scheduler to run undisturbed. Again, each thread has its own submatrix to compute but because MIC uses Hyper-Threading for performance, multiple threads must be assigned to a single core.

6.3 GEMM Details

Partitioning the GEMM routine is simple. Splitting the destination matrix into separate submatrices for each thread to execute is the approach used in this research. Because each element in a matrix-matrix multiply is dependent only on elements from the read-only source matrices as can be seen from Equation 2.1, data races are avoided and hence, synchronization is avoided between threads during computation. This embarrassingly parallel nature provides for good scalability and easy programmability.

Trial and error was used to determine the best way to partition a matrix. Formally, to partition the destination matrix, C, simply divide it by t_r along the rows and t_c along the columns. Then concurrent threads will compute the $t_r \times t_c$ submatrices in parallel (one submatrix per thread) and for optimal performance, on separate physical cores. Some example partitions are shown in Figure 6.1.

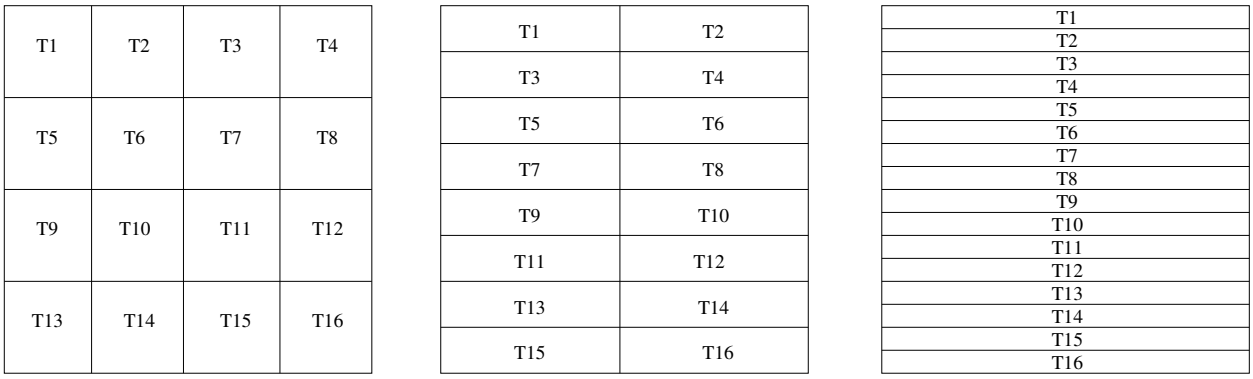

Figure 6.1: Three partitioning schemes for GEMM for 16 threads. Shown are 4×4, 8×2, and 16×1, this last one is also known as 1-D partitioning. Each section is then assigned a single thread as designated by Tx where x is a thread number.

For a complete study to determine how to achieve maximal performance, synchronization was explored to see its effects on the linear algebra routines. Synchronization can be added easily by loading two panels from A and B into packed auxiliary buffers before any computations take place and then computing the panel-panel matrix multiply. This

method was added using all four different programming languages and compared across the architectures vs. the non-synchronized version of GEMM. Figure 6.2 shows this synchronization idea and the results will be discussed in Section 6.7.

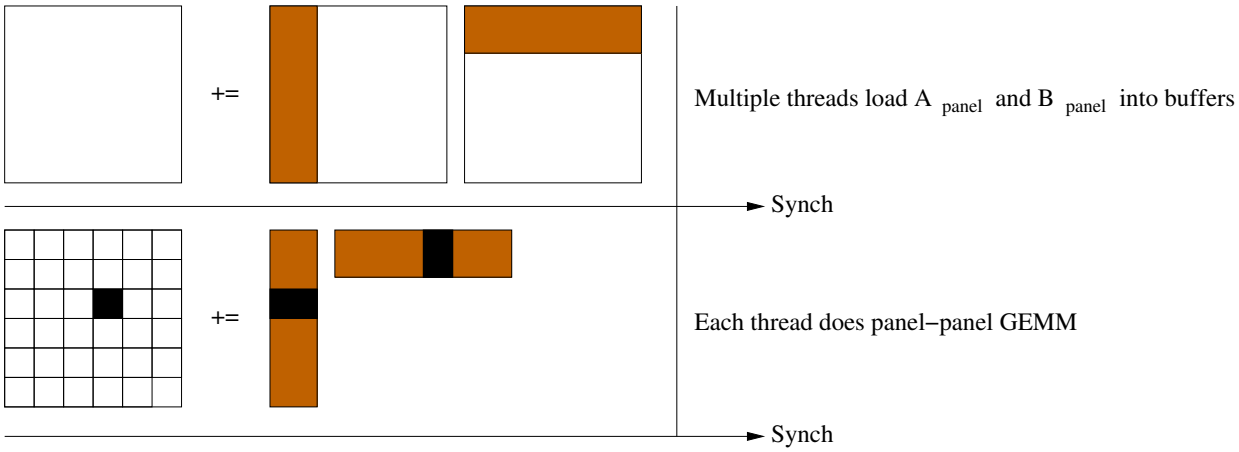

Figure 6.2: Synchronization for GEMM. All threads are used to load and pack entire A and B panels into auxiliary buffers. Then these threads partition the C matrix and update it. Two points of synchronization are needed to ensure no data races occur.

6.4 TRSM Details

Formulating the TRSM routine into a parallel routine produces a two step algorithm. Looking at Figure 4.6 and Figure 6.3 for reference, the first step is to assign $\frac{n}{N_{threads}}$ columns per thread. This step will compute a $K_c \times N_c$ panel of the B matrix. Between the first step and the next step is a point of synchronization between all the threads. The second step calls the parallel GEMM routine to update the rest of the B matrix. This two step process should be repeated until the B matrix is completely solved. This algorithm shows the only way to partition the TRSM panel-solving part of the algorithm (step one) is to partition along the columns in a one dimensional form. The GEMM update is partitioned as described above

in Section 6.3. Synchronization is implicit with this two step algorithm and must be applied using all four parallel languages.

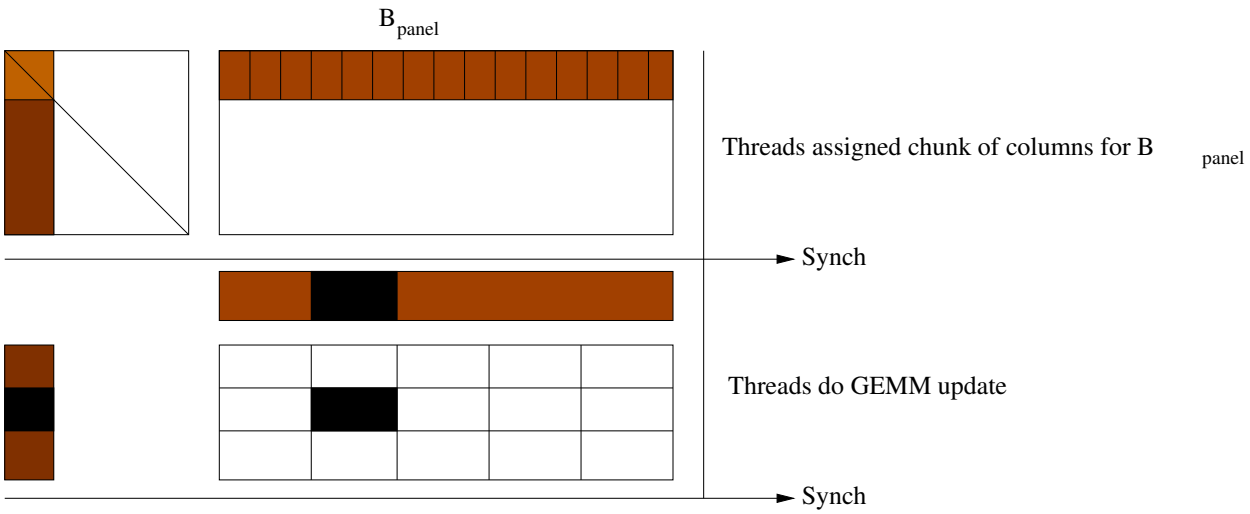

Figure 6.3: Parallel TRSM method. All threads compute chunk of B_{panel}, then threads execute GEMM update.

6.5 SYRK Details

The SYRK routine offers a harder challenge to load-balance because simple rectangular partitioning is ineffective and unbalanced. Because the SYRK routine only updates either the lower or upper triangular part of the destination matrix, simple rectangular portions of the destination matrix lead to unbalanced work among all the threads. To alleviate this problem, some simple arithmetic must be used to ensure a equal number of destination matrix elements are computed by each thread. The partitioning scheme in this research is to partition the matrix into $t_r \times t_c$ portions where the areas of the partitions are equal. Figure 6.4 shows an example 4×2 partitioning of an upper triangular matrix where areas $A_1 = A_2 = ... = A_8$. This problem can be constructed as a recursive problem as shown below. The recursive partitioning algorithm for an upper triangular partition is shown in

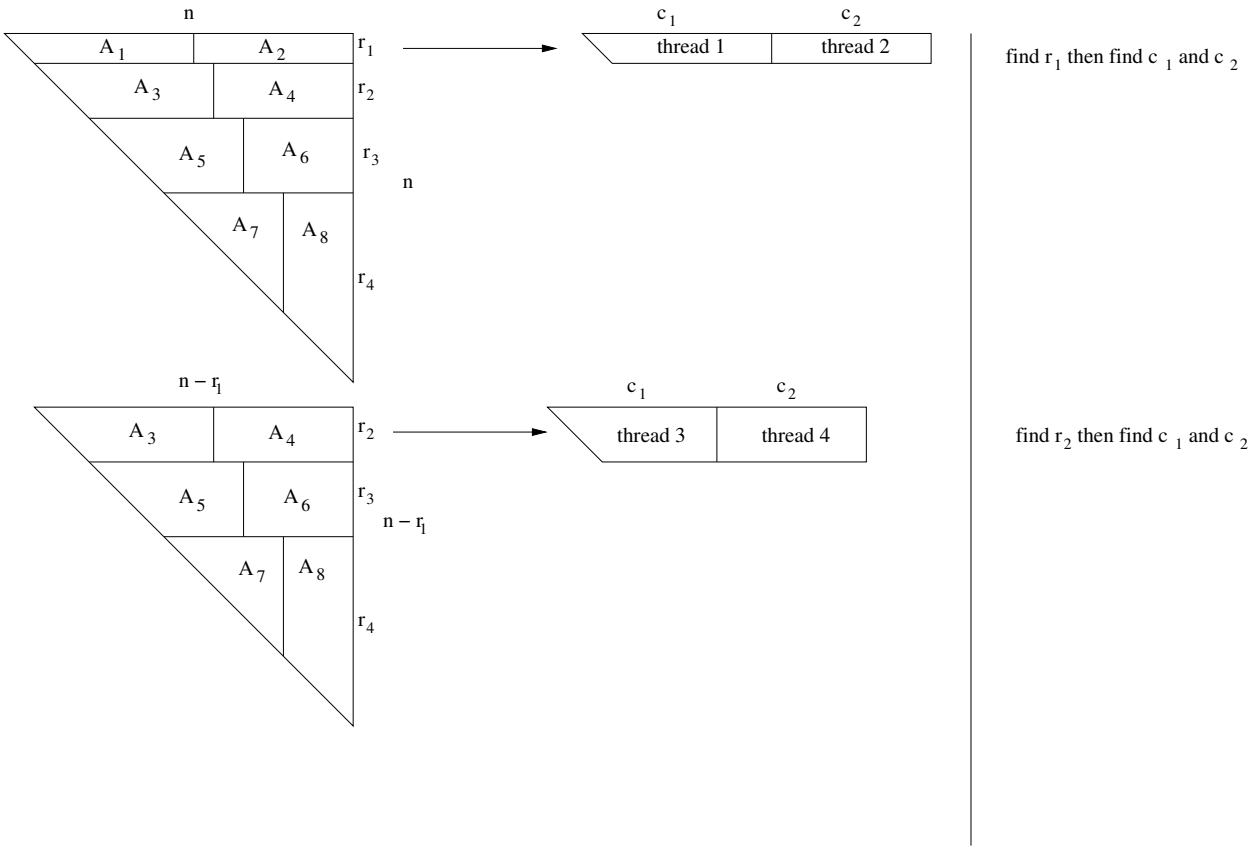

Figure 6.4: Recursively partitioning a $n \times n$ matrix for SYRK using a 4×2 scheme into eight equal areas, $A_1, A_2, ..., A_8$. The second part is reduced to finding a 3×2 scheme into six equal areas.

Figure 6.4. The algorithm will first find the top row panel of the destination matrix that follows the equation

$$\frac{1}{t_r}\frac{n^2}{2} = nr_1 - \frac{1}{2}r_1^2 \Rightarrow r_1 = n \pm \sqrt{n^2 - \frac{1}{t_r}}$$

The answer $n + \sqrt{n^2 - \frac{1}{t_r}}$ can be thrown out because r_1 cannot be larger than n. Now that r_1 is known, solving for c_1 and c_2 is simple. The term c_1 should be found by applying this equation

$$\frac{1}{t_c}\left(nr_1 - \frac{1}{2}r_1^2\right) = c_1 r_1 - \frac{1}{2}r_1^2 \Rightarrow c_1 = \frac{\frac{1}{t_c}\left(nr_1 - \frac{1}{2}r_1^2\right) + \frac{1}{2}r_1^2}{r_1}$$

Now that c_1 is known, finding c_2, c_3, ... , c_{t_c} is easy because the remaining area is a rectangle which can be partitioned by simple division. Figure 6.4 only has $t_c = 2$. Once r_1, c_1, c_2, ...

, c_{t_c} are found, then the problem reduces recursively to finding a partition of $(r_c - 1) \times t_c$ of the remaining triangular part of the destination matrix.

A final note about parallel SYRK is that some of the areas assigned to threads turn out to be rectangular. Hence, GEMM can be used to solve these areas. For the areas on the edge of the triangle though, the special SYRK kernel must be used so that only the proper part of the destination matrix is updated.

6.6 POTRF Details

The parallel version of Cholesky decomposition is composed solely of parallel versions of the BLAS routines, GEMM, TRSM, and SYRK mixed with a serial routine called POTF2. While GEMM is not explicitly called, it is the basis of TRSM and SYRK. This algorithm is exactly the same as shown in Listing 4.6. The fork-join model is used to implement this parallel algorithm and there are implicit barriers between BLAS calls.

6.7 Results

This section will discuss the results obtained when running the parallel codes for all architectures. As a reminder, each point on every graph represents an average of five runs of the same size or core count. The scaling graphs run the problem on a constant large size for each core count. For Nehalem and Sandy Bridge this is a $m = n = k = 9100$ and for MIC this size is $m = n = k = 12000$. Figures 6.5 and 6.6 show the results comparing all the routines next to each other for Nehalem. For all the Level 3 BLAS routines, the performance was over 80% peak with the GEMM routine reaching 86% peak performance. The POTRF routine for square matrices larger than 6000, the performance reached 75% peak. Both the aforementioned graphs use the OpenMP implementation because of it produced the best results overall. The next eight figures show the comparisons for the different parallel programming languages used. These language comparison graphs also show the results compared to Intel's MKL library. And a special note should be made of the POTRF

graphs because two versions of POTRF exist for MKL. One is the direct call to MKL's POTRF function (labeled *potrf(MKL)*) and the other is the LAPACK definition (labeled *potrf(MKL-LAPACK)*) of POTRF that call the BLAS routines explicitly. The difference between these two is subtle but important. MKL's direct library call is capable of calling "extended BLAS routines" which are BLAS routines that don't follow the exact BLAS standard interface according to [6]. For instance, the B matrix in the TRSM update of the POTRF algorithm could pack its values in an auxiliary buffer for the GEMM update to use helping avoid repacking this data by the GEMM routine. Also, some optimizations are done to avoid deallocation/reallocation of memory for better cache performance. Both the strict LAPACK version which calls BLAS and the direct MKL version are there for fair comparison. More importantly, the direct MKL version is most likely using a more sophisticated parallel algorithm that can avoid the pitfalls of the LAPACK definition. An example of this would be the look ahead algorithm described in [27, 28] which is used in the LINPACK[29] benchmark.

Figures 6.17 and 6.18 show the results comparing all routines for Sandy Bridge. The GEMM routines attain 90% peak with TRSM and SYRK achieving near 80% to 85% peak. POTRF is able to achieve over 70% peak for both single and double precision. The next eight figures show and support the same conclusions and patterns concerning the language comparisons. OpenMP does the best with Pthreads and Cilk giving second best results and TBB attaining the worst results. The scaling graphs show that the algorithms scale with an increase in number of cores used.

Figures 6.29 and 6.30 show the results comparing all routines for MIC. The GEMM routine can get near 75% peak for SGEMM and near 80% for DGEMM. Unfortunately, because the small POTF2 routine is serial in nature, an inordinate amount of time is spent in this routine for MIC. For example, with Sandy Bridge and Nehalem the percentage of execution time in POTF2 is less than 3% while on MIC, it is close to 20%. This results in SPOTRF achieving 40% peak and DPOTRF achieving 50% peak. Also, as a reminder to the reader, only OpenMP and Pthreads were implemented for MIC because Cilk and TBB did not work properly at the time of writing. Again, OpenMP does achieves the best

performance. With regards to the scaling results, MKL's TRSM routine did not scale past 32 cores, but MKL's GEMM routines scaled nearly perfectly.

For The Sandy Bridge and Nehalem architectures, the comparison between the OpenMP implementations programmed for this research show results that are on par with MKL. For the MIC architecture, MKL's GEMM routines simply perform better by 10%, but MKL's TRSM routines don't scale and hence perform 39% worse. While the SYRK and POTRF routines for MKL and the implemented SYRK and POTRF routines are right on par with each other.

The language comparison sorts itself out cleanly with a clear hierarchy. The best language to use for these types of codes is OpenMP. The next best thing was Pthreads, with Cilk and TBB following up in third and fourth respectively. This results hinged closely on the ability to assign threads to cores, called hardware or thread affinity. When a language allowed you to do this explicitly, such as OpenMP and Pthreads, then the performance was comparable to MKL. Cilk and TBB unfortunately do not have this feature. TBB has a scheduler that allows threads to stay on the same core but not a way to explicitly state which threads map to which cores. Cilk offers neither. TBB and Cilk are geared more towards having numerous work chunks that can be mapped and scheduled to a smaller number of cores. This problem was set up so that there was one chunk of the matrix per thread and so the schedulers for both TBB and Cilk were unnecessary. The hack to allow Cilk hardware affinity is to use a parallel `cilk_for` (the Cilk analog to OpenMP's `#pragma omp parallel for`) that iterates from 0 to $n-1$ threads. Set the `chunksize` to 1 and then within the `cilk_for` section call a `pthread_setaffinity_np()` command. This way n threads will set their own affinity to the proper core based on the thread number (0 to $n-1$). The same technique can be applied to TBB with the `parallel_for` class playing the role of `cilk_for`. Unfortunately, this did not work for TBB and the performance overall for TBB was significantly worse compared to OpenMP, Pthreads, and Cilk. TBB was unable to even eclipse the 50% mark for all the routines.

Another aspect of the language comparisons is programmability and general ease of use. While subjective to a certain degree, it is nonetheless important to gauge difficulty

in programming and seeing what this offers in terms of performance. OpenMP and Cilk are the easiest to program in with TBB coming in third and Pthreads in fourth. Some reasoning behind these choices is the language's verbosity, efficacy, and programmer efficiency. OpenMP proved to be extremely easy to incorporate because of compiler support and low verbosity. As well as ease of programming, OpenMP offered full support of the invaluable ability of hardware affinity through the use of environment variables. Cilk was as easy to program in as OpenMP because of direct analogs between the parallel programming constructs in the languages. Namely, `cilk_for` ↔ `#pragma omp parallel for` and `cilk_sync` ↔ `#pragma omp barrier`. As mentioned above, Cilk does not offer direct support for thread assignment to cores and hence was not as effective in performance. In addition, Cilk exhibited more erratic performance curves with higher variance than the other languages as indicated in the graphs. TBB was the most peculiar of the languages because of its association with C++. All parallel constructs are templatized C++ classes. While powerful, a tedious amount of programming for simple parallelizations like a parallel-for is required as compared to OpenMP or Cilk. Special classes that wrap around the BLAS routines have to be created which bloviates the code and felt unnatural. Similar to Cilk, TBB offers no direct way of hardware affinity. Pthreads was both difficult to program and long-winded. While performance was comparable to OpenMP, a thread pool was programmed which required the use of parallel programming primitives such as mutexes and condition variables. Debugging this type of code is both tedious and grisly because of the intrinsically difficult and error-prone nature of it. For these reasons, its difficulty of programming was ranked the hardest among the four different languages.

The third set of graphs in Figures 6.16 and 6.15 for each architecture show the basic scaling property of the routines vs. the number of cores used. These scaling graphs for Nehalem and Sandy Bridge are all computing a square problem where $m = n = k = 9100$ ($m = n = k = 12000$ for MIC) for the relevant routine. Near linear scaling for these problems should be achieved due to the embarrassingly parallel nature of dense linear algebra. And these graphs illustrate the routines' near linear growth with number of cores used. This characteristic proves important when trying to scale the problem for use with more cores.

Table 6.1: Parallel programming languages comparison. Performance ranks from 1 to 4 with 1 being the best performance. Programming Difficulty ranks from 1 to 4 with 1 being the easiest to program.

Language	Performance Rank	Programming Difficulty
OpenMP	1	1
Pthreads	2	4
Cilk	3	2
TBB	4	3

All these graphs for the Nehalem architecture have a corresponding graph for the Sandy Bridge architecture. Figures 6.17 and 6.18 compare the OpenMP parallel routines with each other followed by eight language comparison graphs and then finally finishes off with Figures 6.28 and 6.27 showing the scaling results for Sandy Bridge. From Figures 6.17 and 6.18, it can be seen that GEMM gets around 90% performance with TRSM following at around 83% and SYRK at 80%. The POTRF routine can achieve above 70% for large matrices. The language comparison graphs show that for isolated BLAS routines, OpenMP, Cilk, and Pthreads all perform about the same with TBB getting anywhere from 5% to 10% worse performance. When POTRF is called though, OpenMP performs the best followed by Pthreads, then Cilk and TBB performs the worst. An explanation for Cilk and TBB's performance drop when running the POTRF routine is the languages can not assign a thread to a core and hence will perform worse. This is brought on because when a thread is assigned a thread number in Cilk and TBB, it might not match the last time and so caches will be filled with irrelevant data and cache performance drops. This would also explain the higher variance in results, because sometimes the threads will match and sometimes they will not where as in OpenMP and Pthreads, the same threads will always be assigned the same thread number. The scaling properties on Sandy Bridge were similar to Nehalem and showed slightly sublinear scaling (Figures 6.28 and 6.27).

A special note must be made to emphasize that the results for Cilk and TBB were aided by a Pthread function call which somewhat violates the closedness* of the languages.

*closedness here refers to using only the definitions, keywords and other constructs of a language and incorporating no other language.

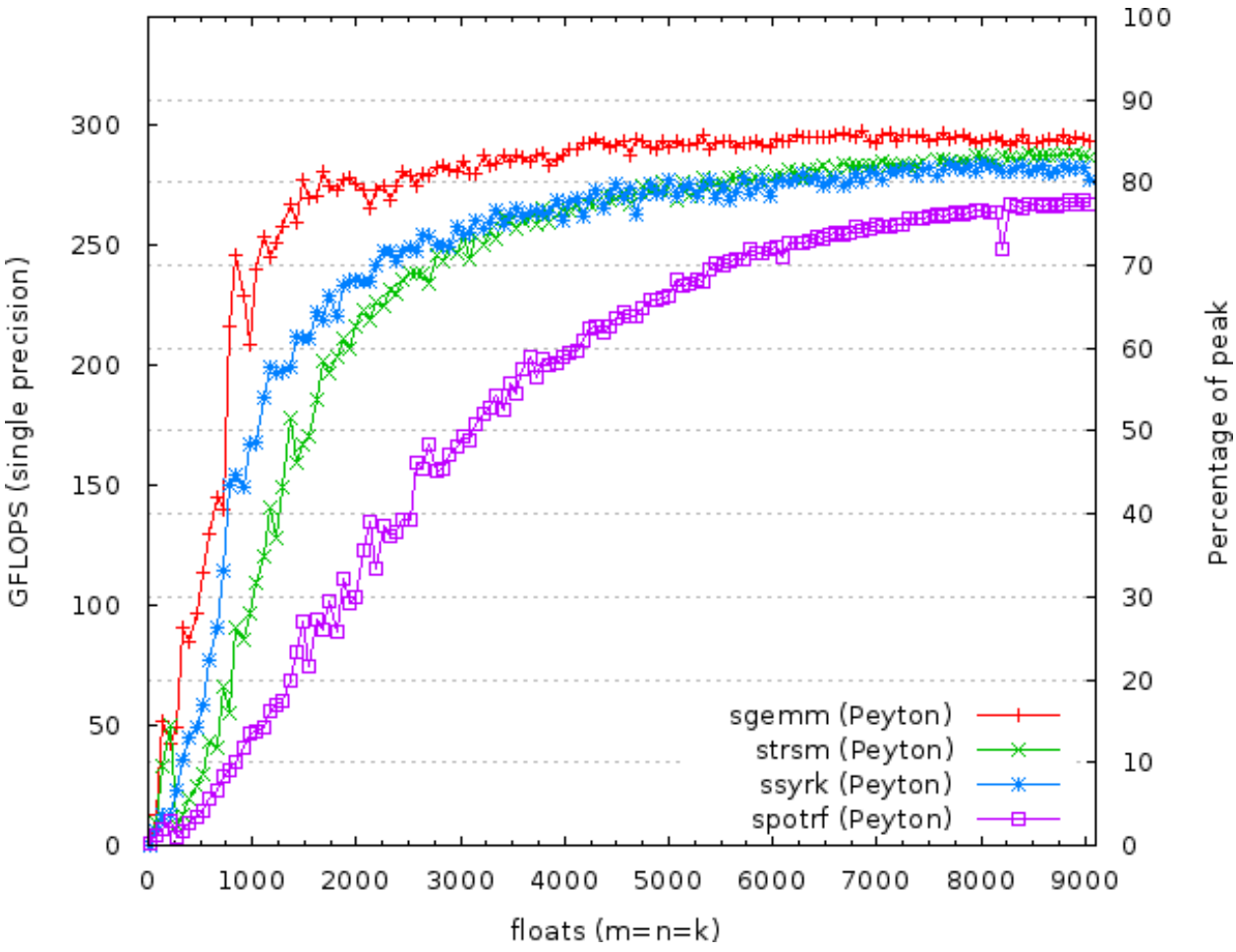

Figure 6.5: Single precision parallel codes using all cores on Nehalem

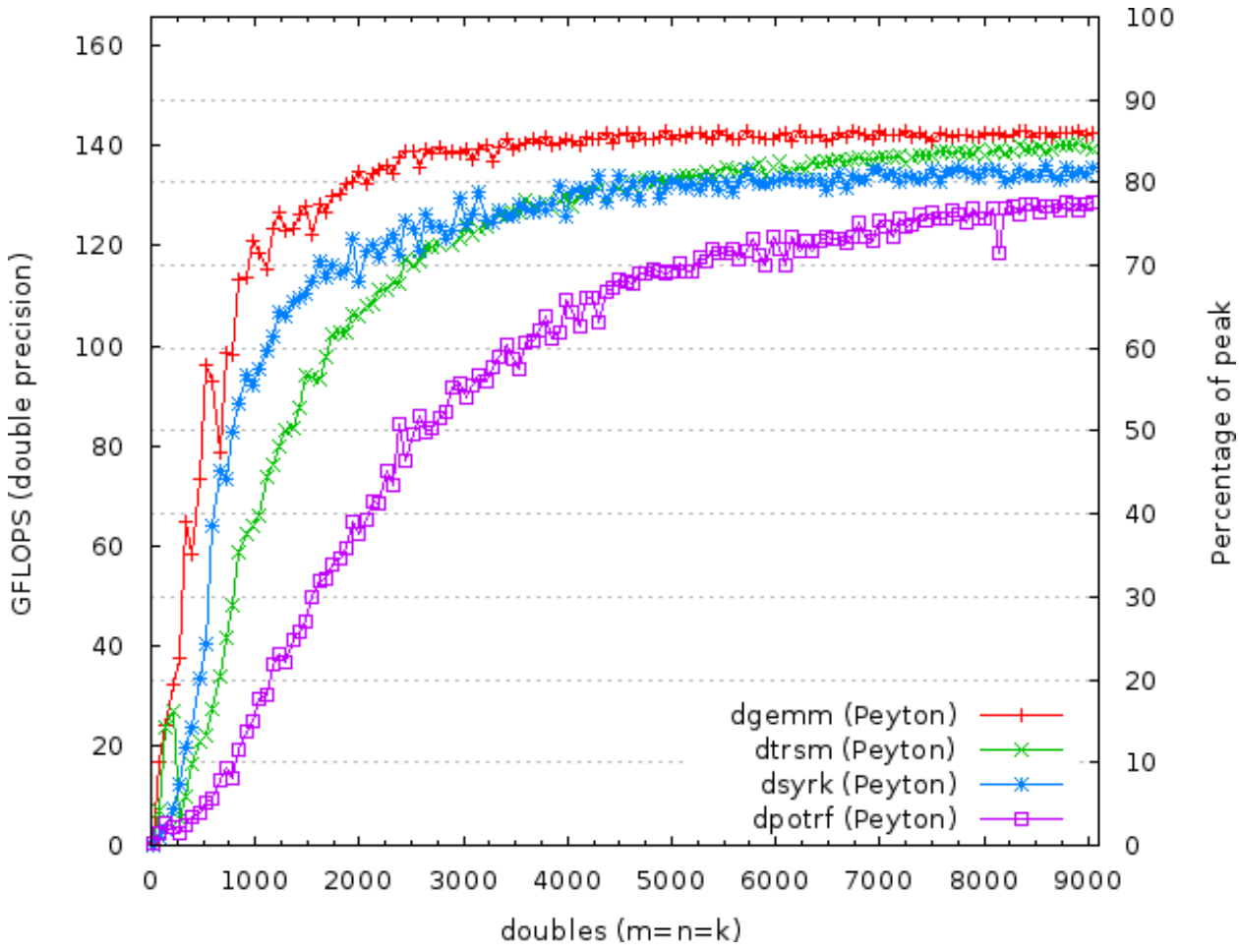

Figure 6.6: Double precision parallel codes using all cores on Nehalem

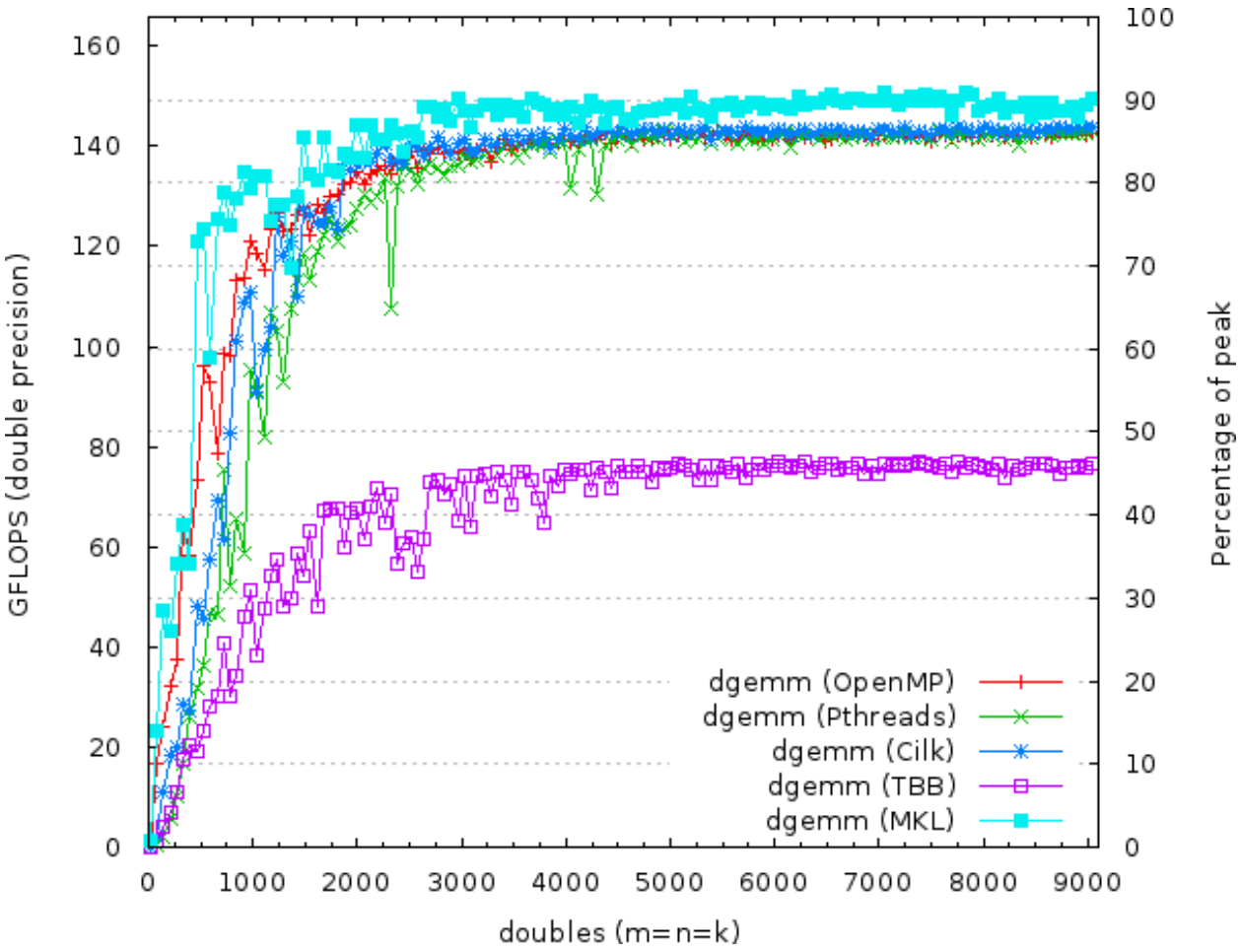

Figure 6.7: Double precision comparison of parallel languages for GEMM routine on Nehalem

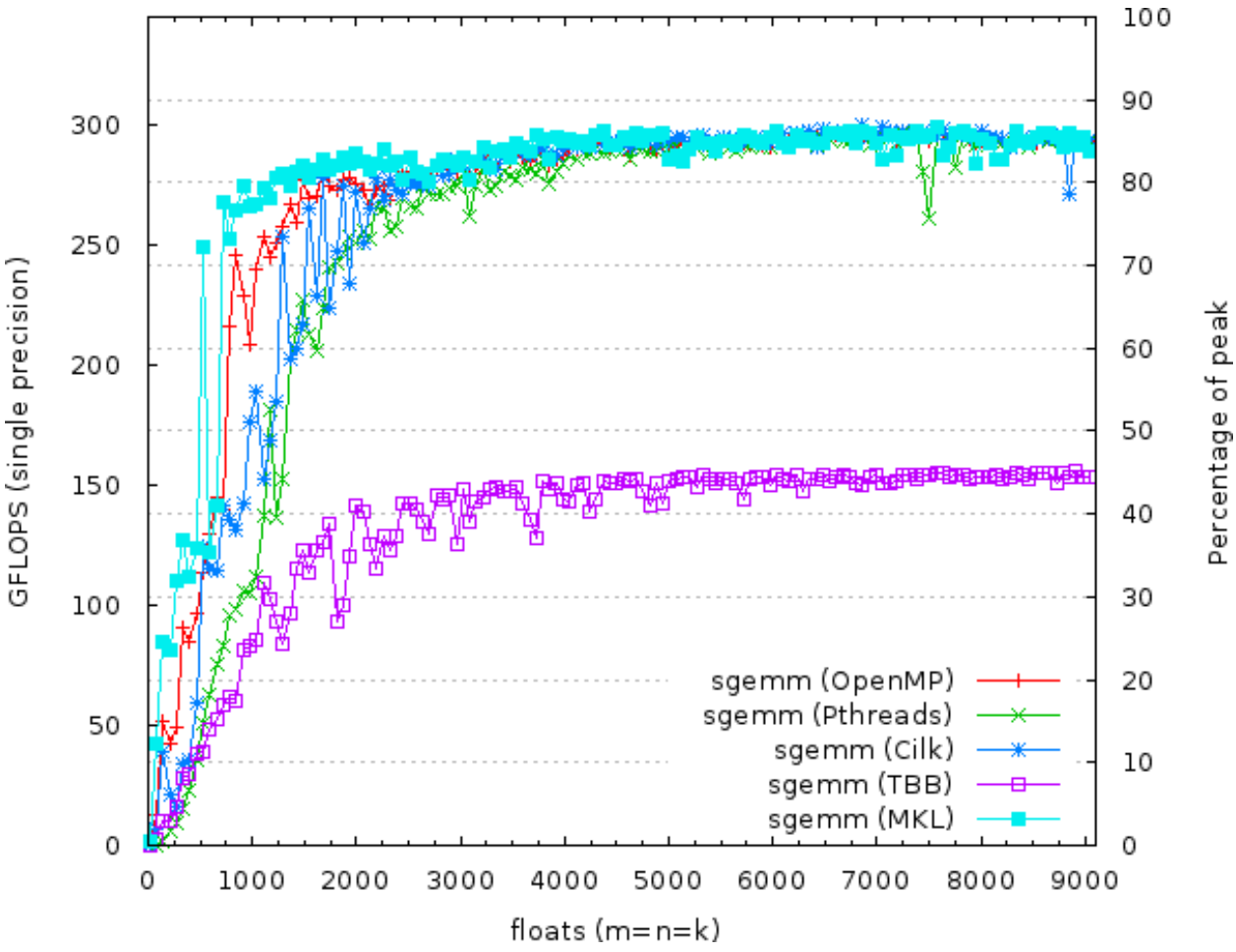

Figure 6.8: Single precision comparison of parallel languages for GEMM routine on Nehalem

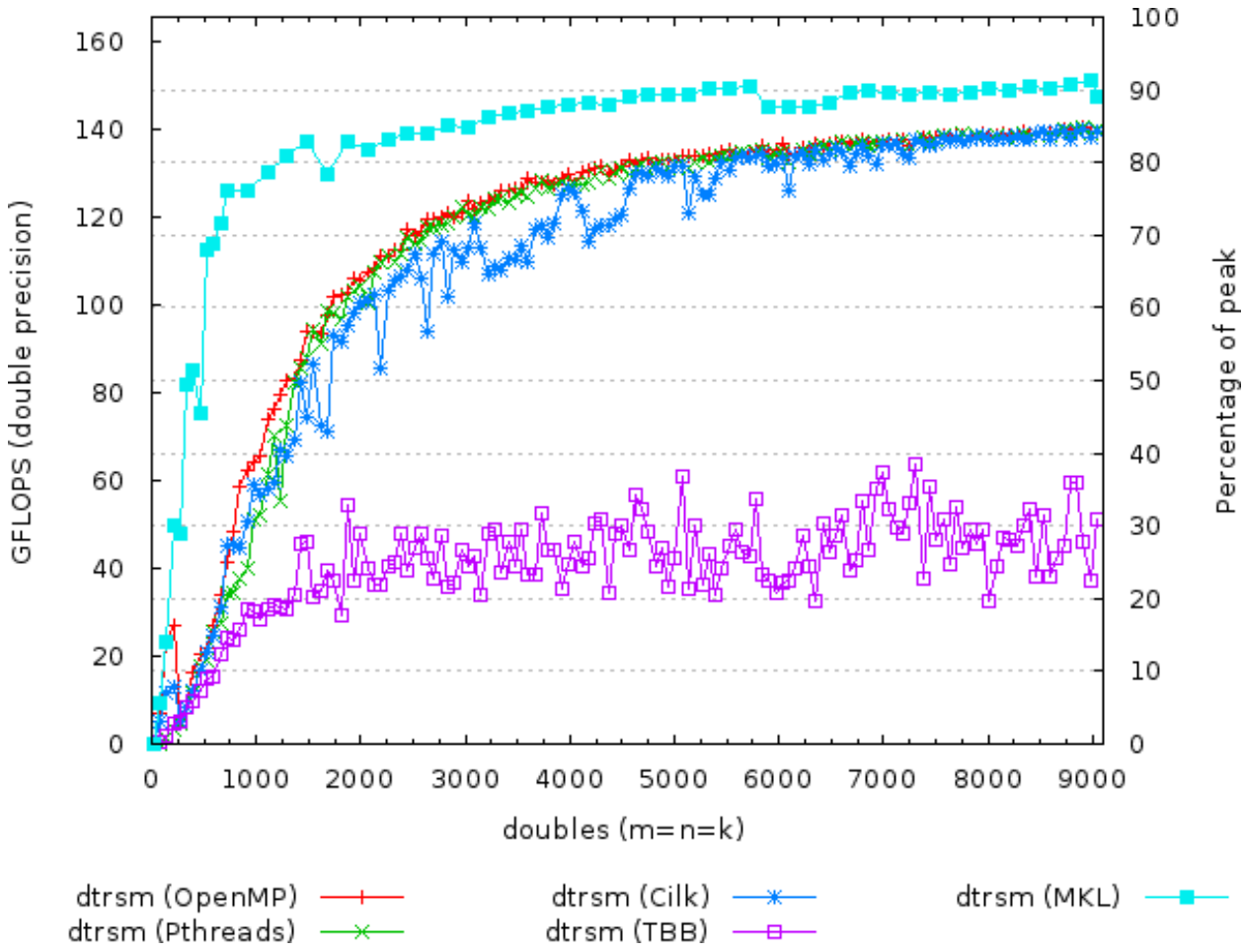

Figure 6.9: Double precision comparison of parallel languages for TRSM routine on Nehalem

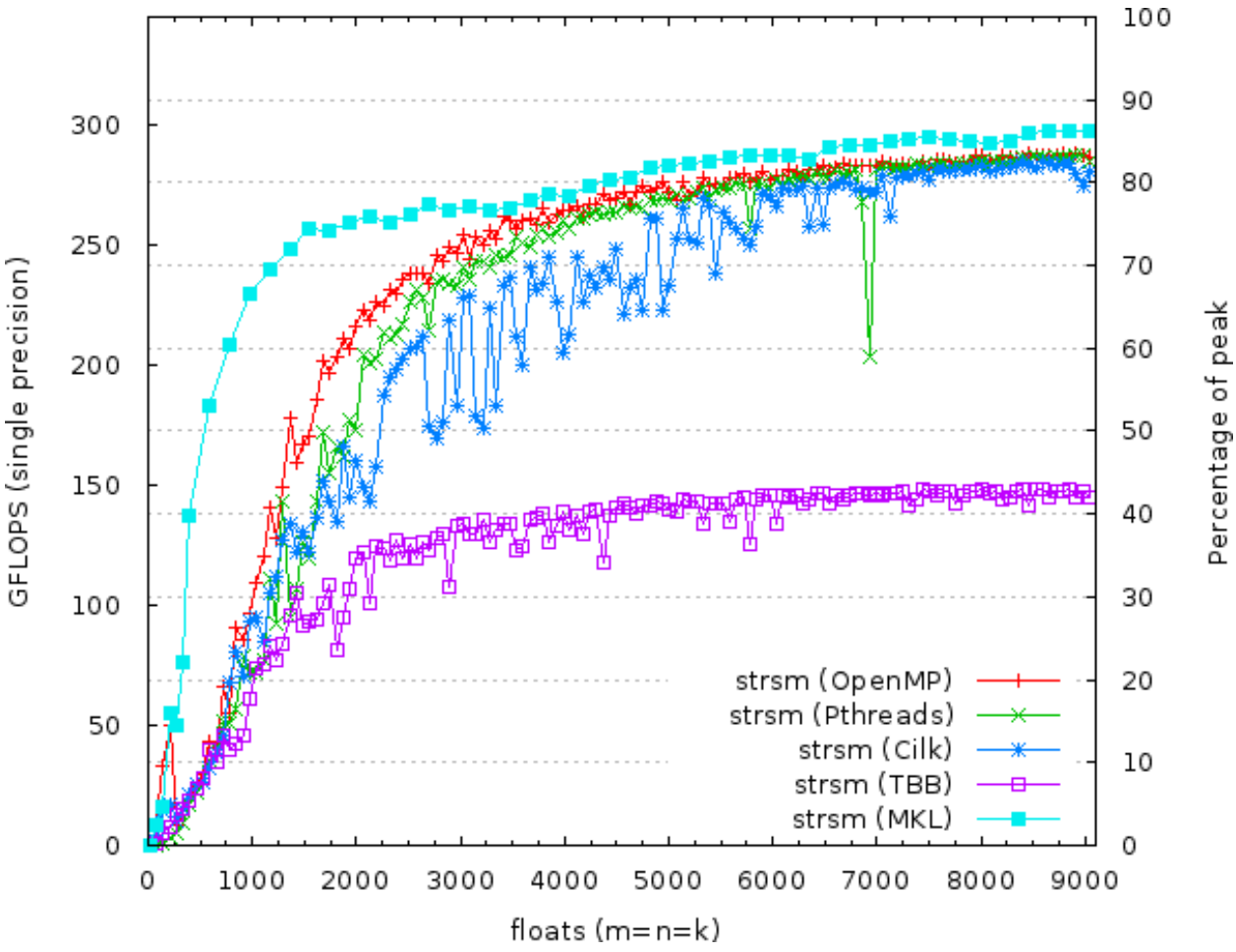

Figure 6.10: Single precision comparison of parallel languages for TRSM routine on Nehalem

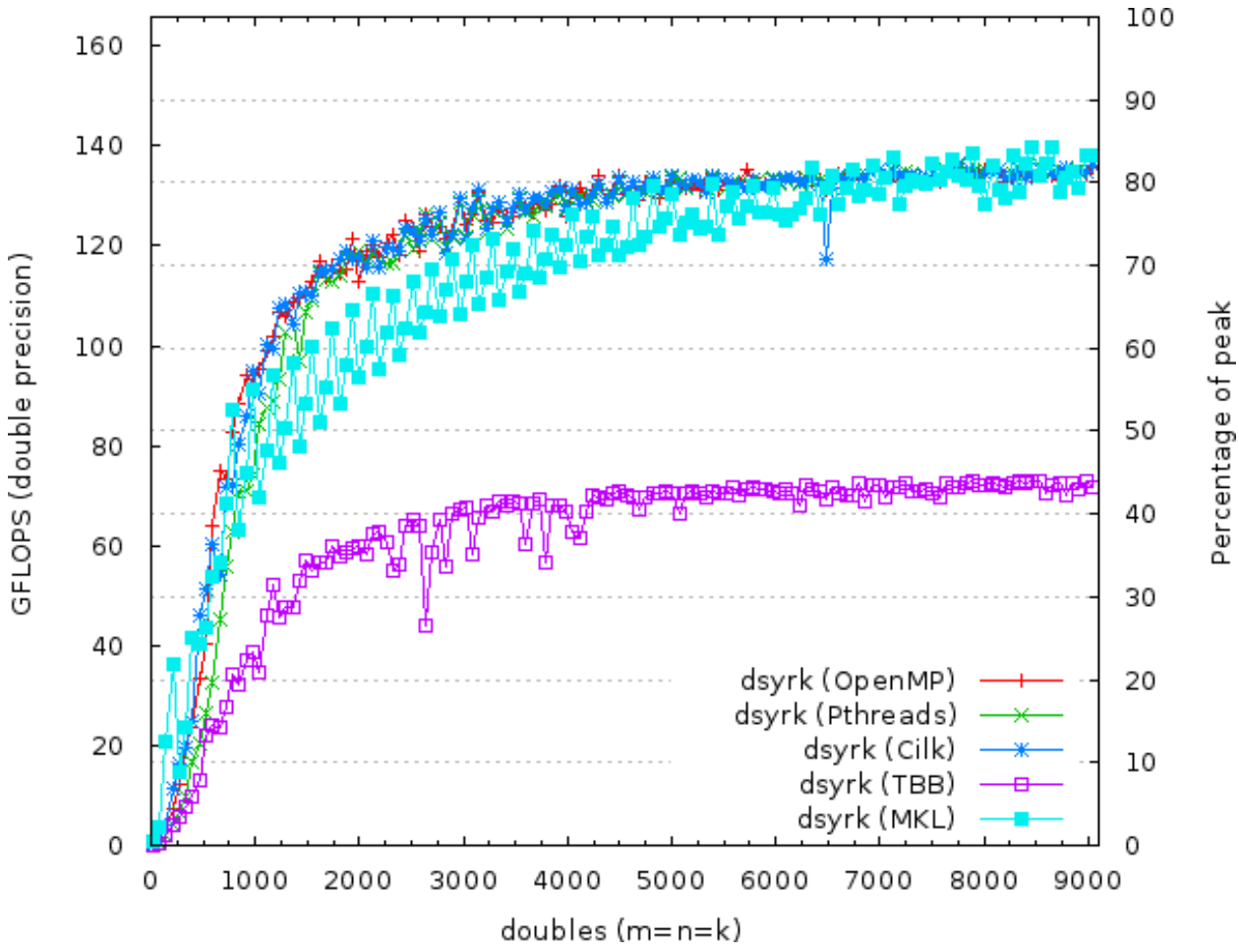

Figure 6.11: Double precision comparison of parallel languages for SYRK routine on Nehalem

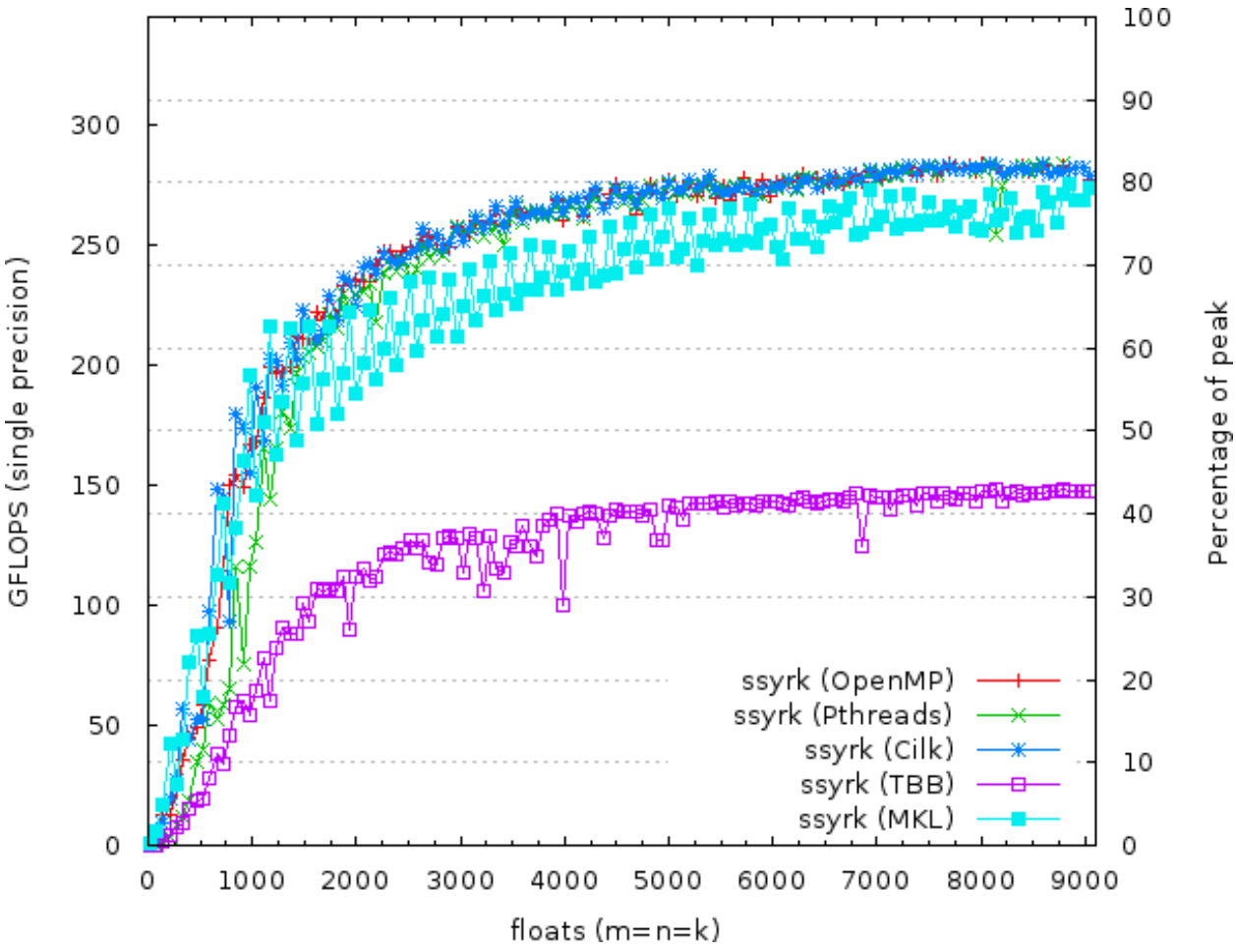

Figure 6.12: Single precision comparison of parallel languages for SYRK routine on Nehalem

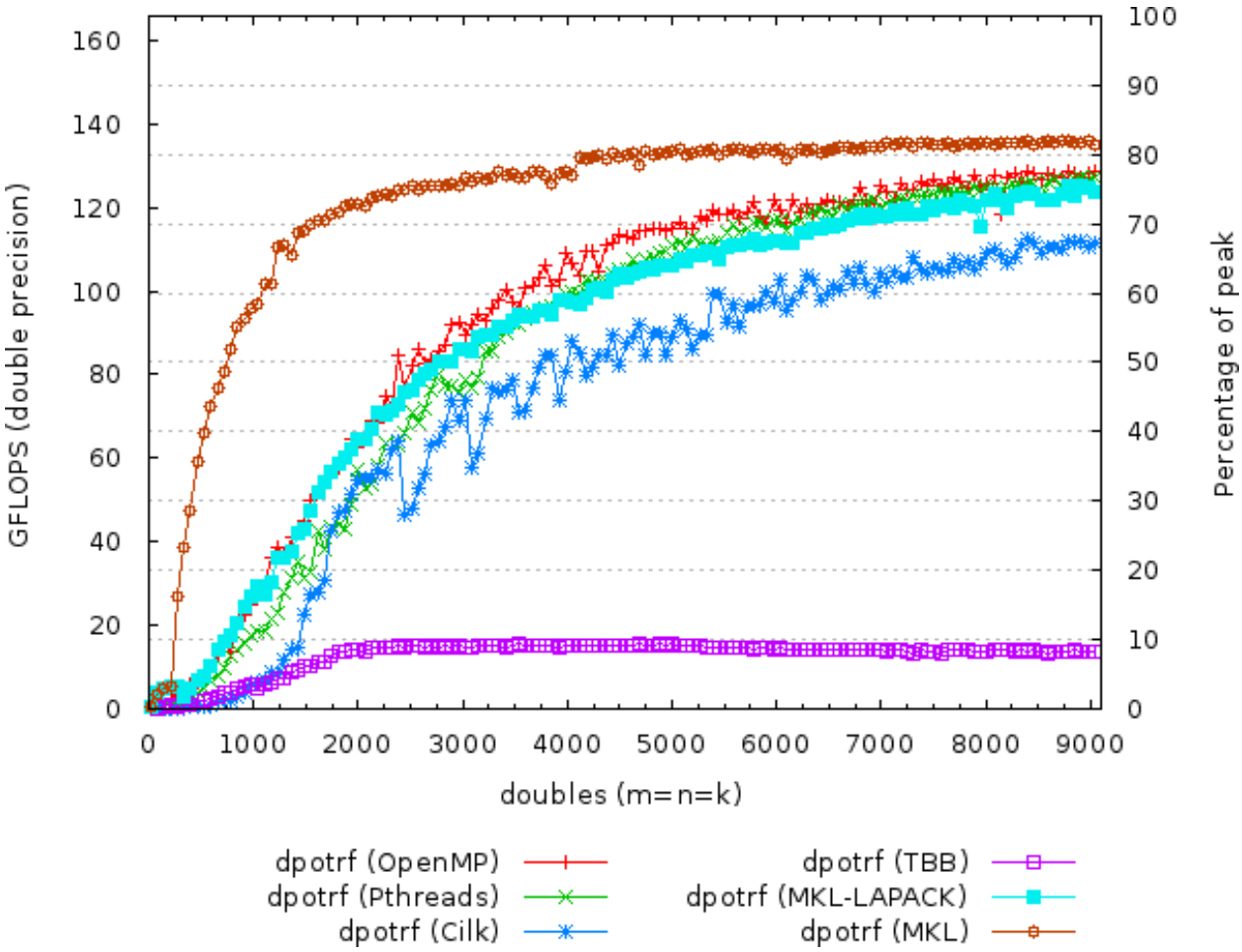

Figure 6.13: Double precision comparison of parallel languages for POTRF routine on Nehalem

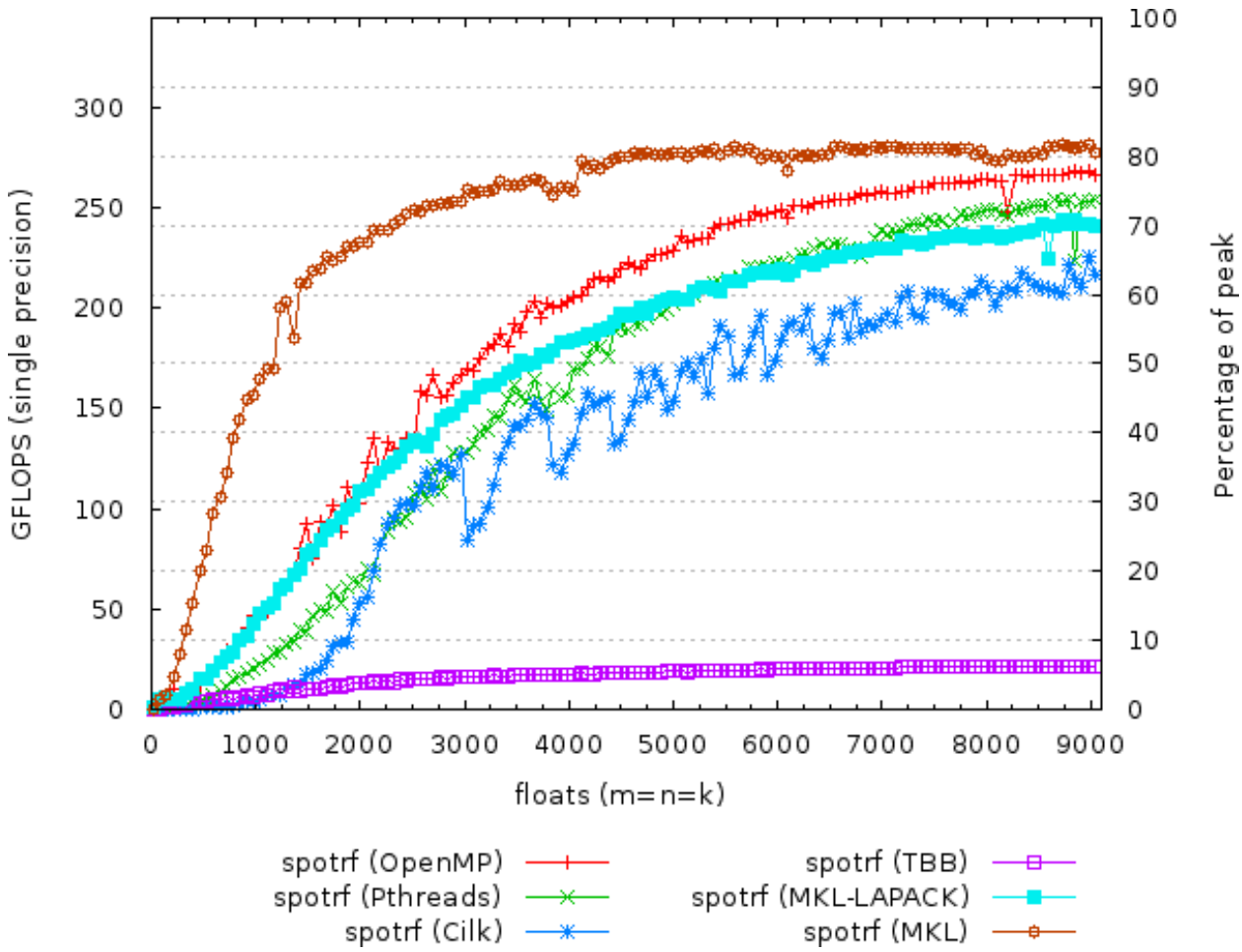

Figure 6.14: Single precision comparison of parallel languages for POTRF routine on Nehalem

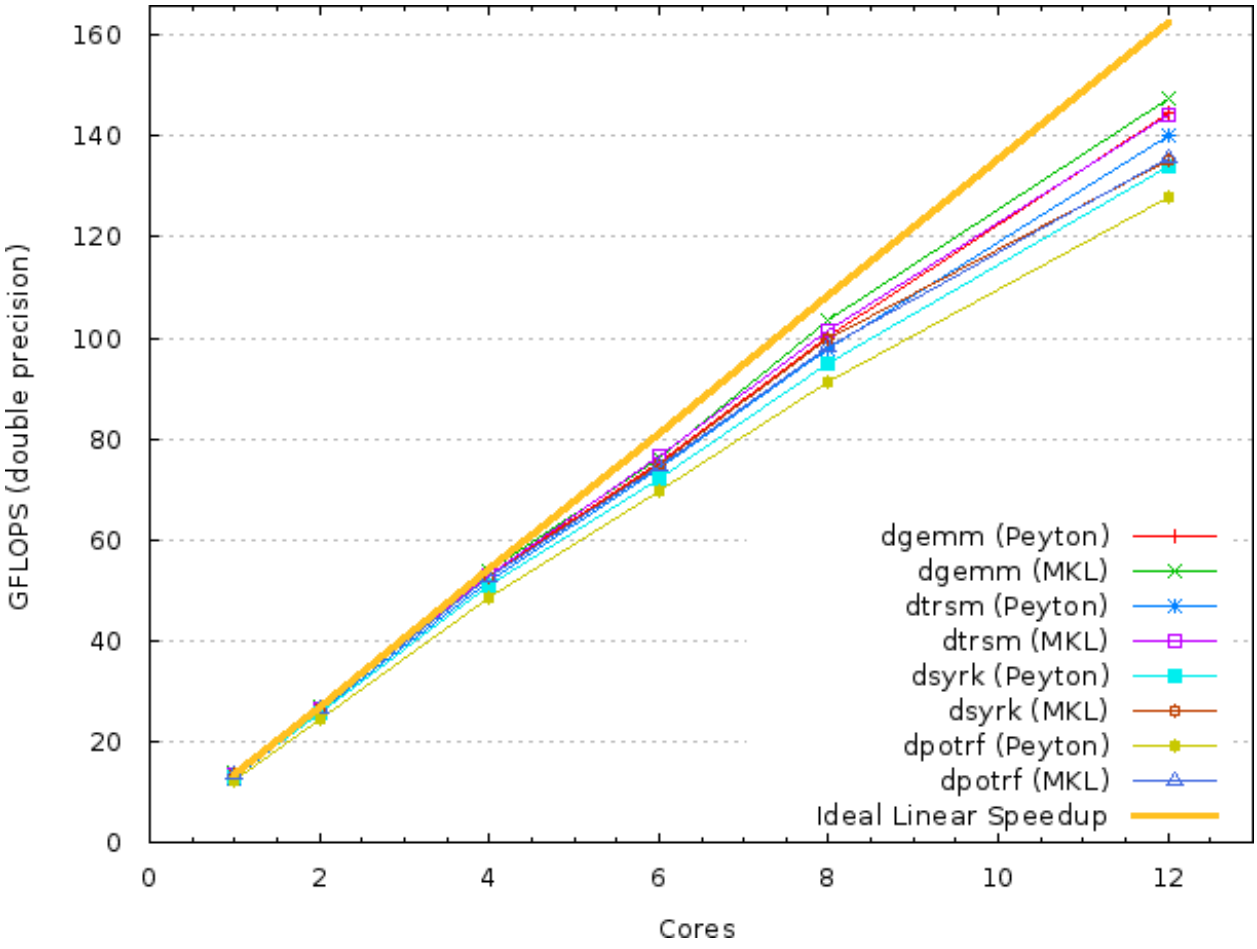

Figure 6.15: Double precision scaling for Nehalem up to 12 cores. For each core count, the size $m = n = k = 9100$ was used.

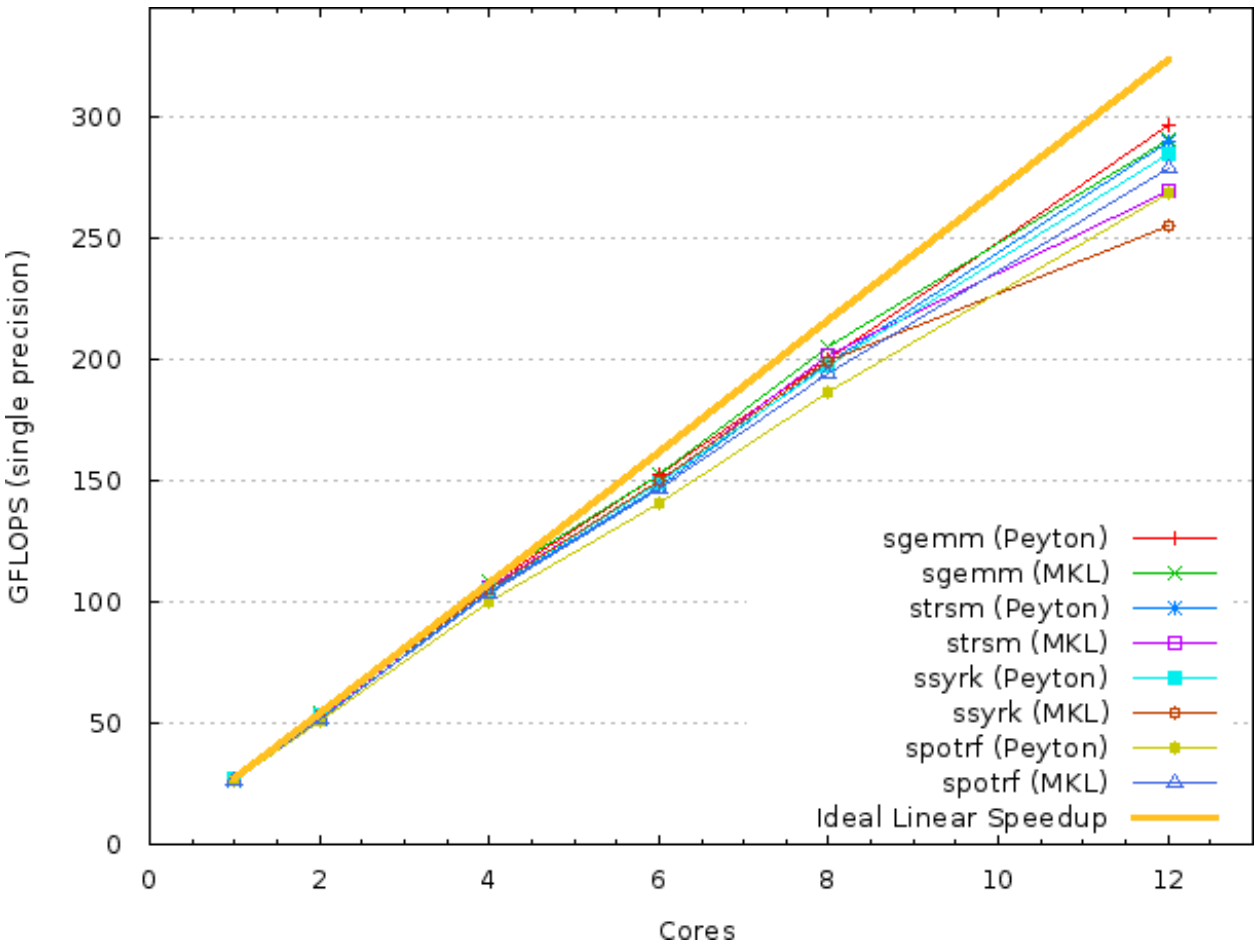

Figure 6.16: Single precision scaling for Nehalem up to 12 cores. For each core count, the size $m = n = k = 9100$ was used.

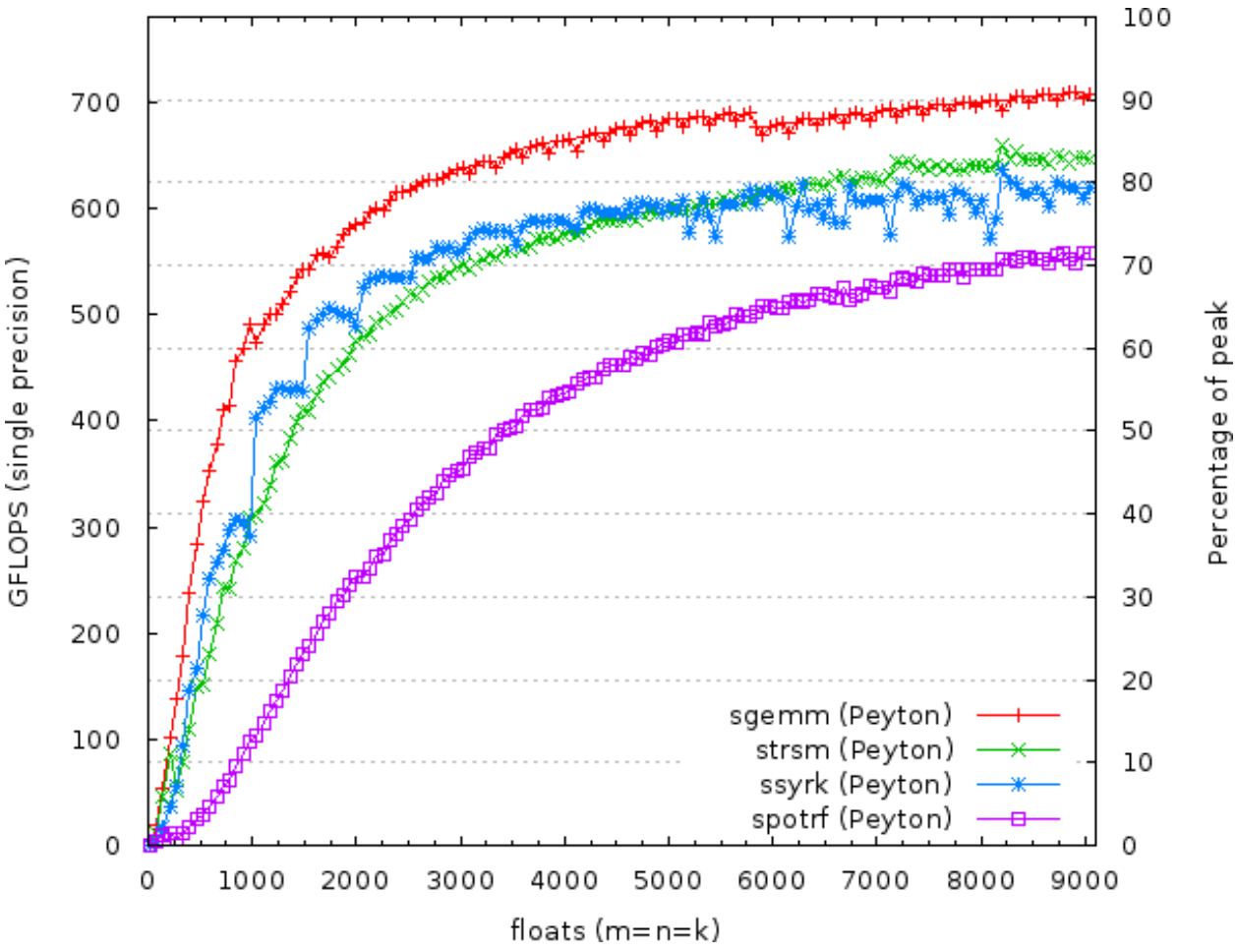

Figure 6.17: Single precision parallel codes using all cores on Sandy Bridge

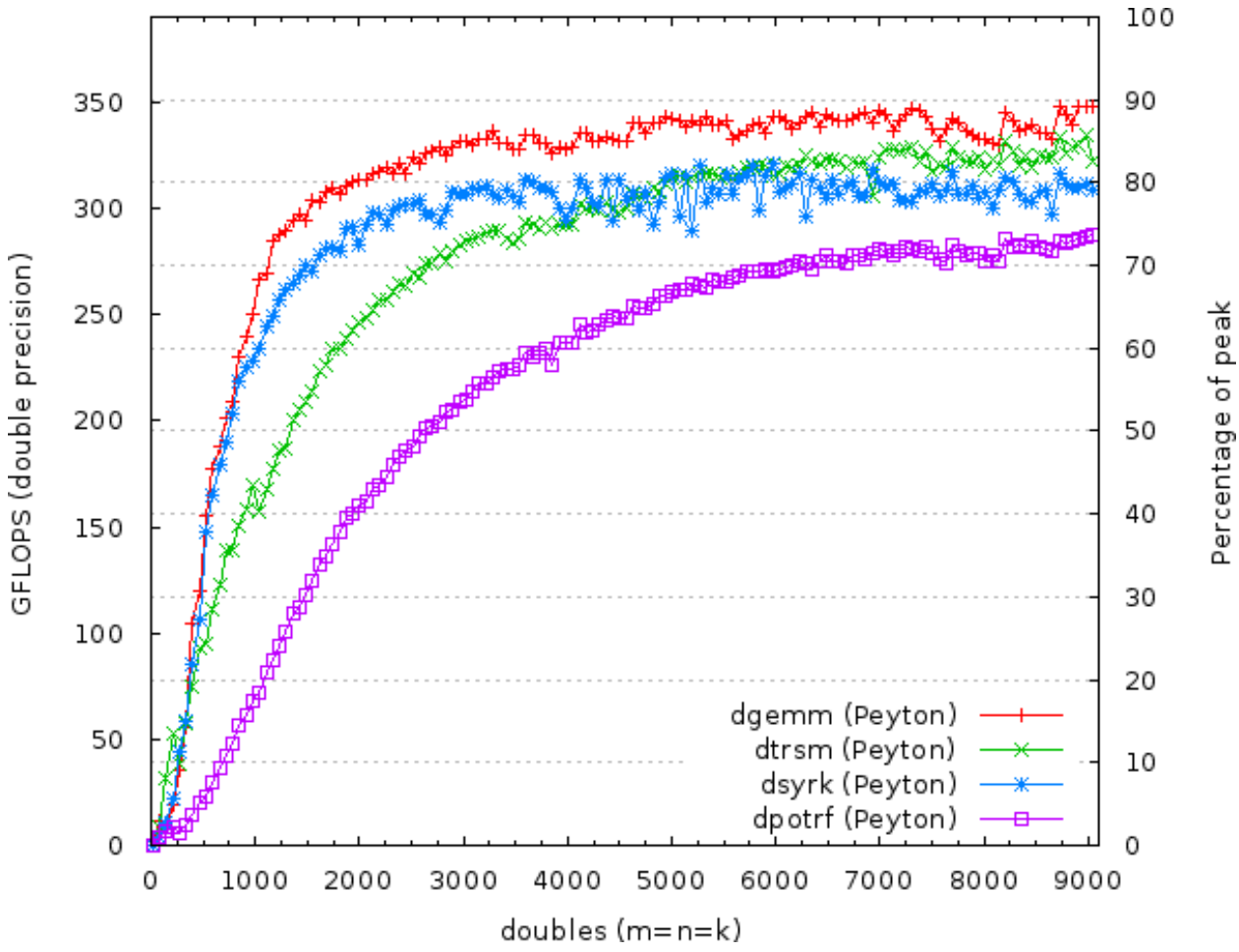

Figure 6.18: Double precision parallel codes using all cores on Sandy Bridge

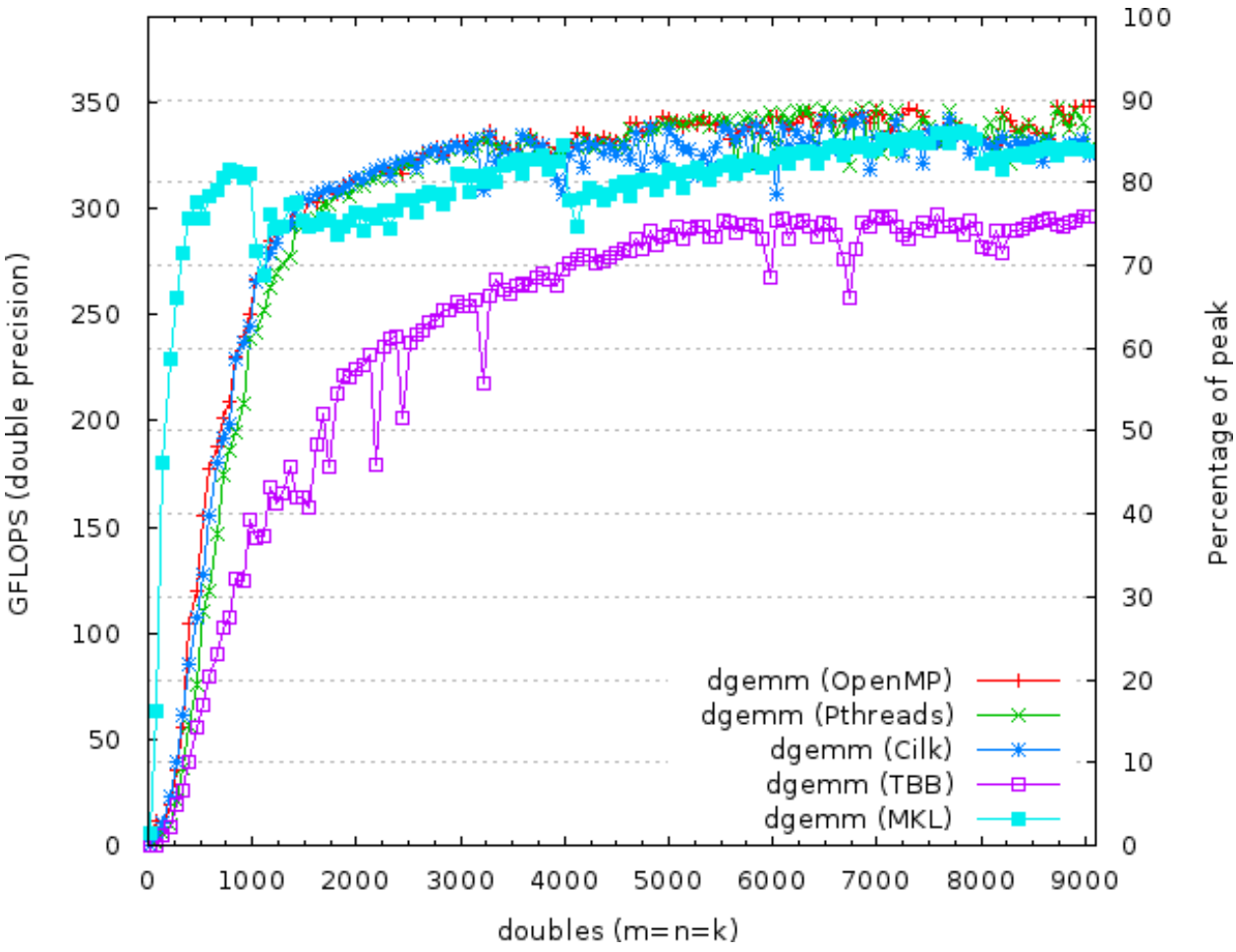

Figure 6.19: Double precision comparison of parallel languages for GEMM routine on Sandy Bridge

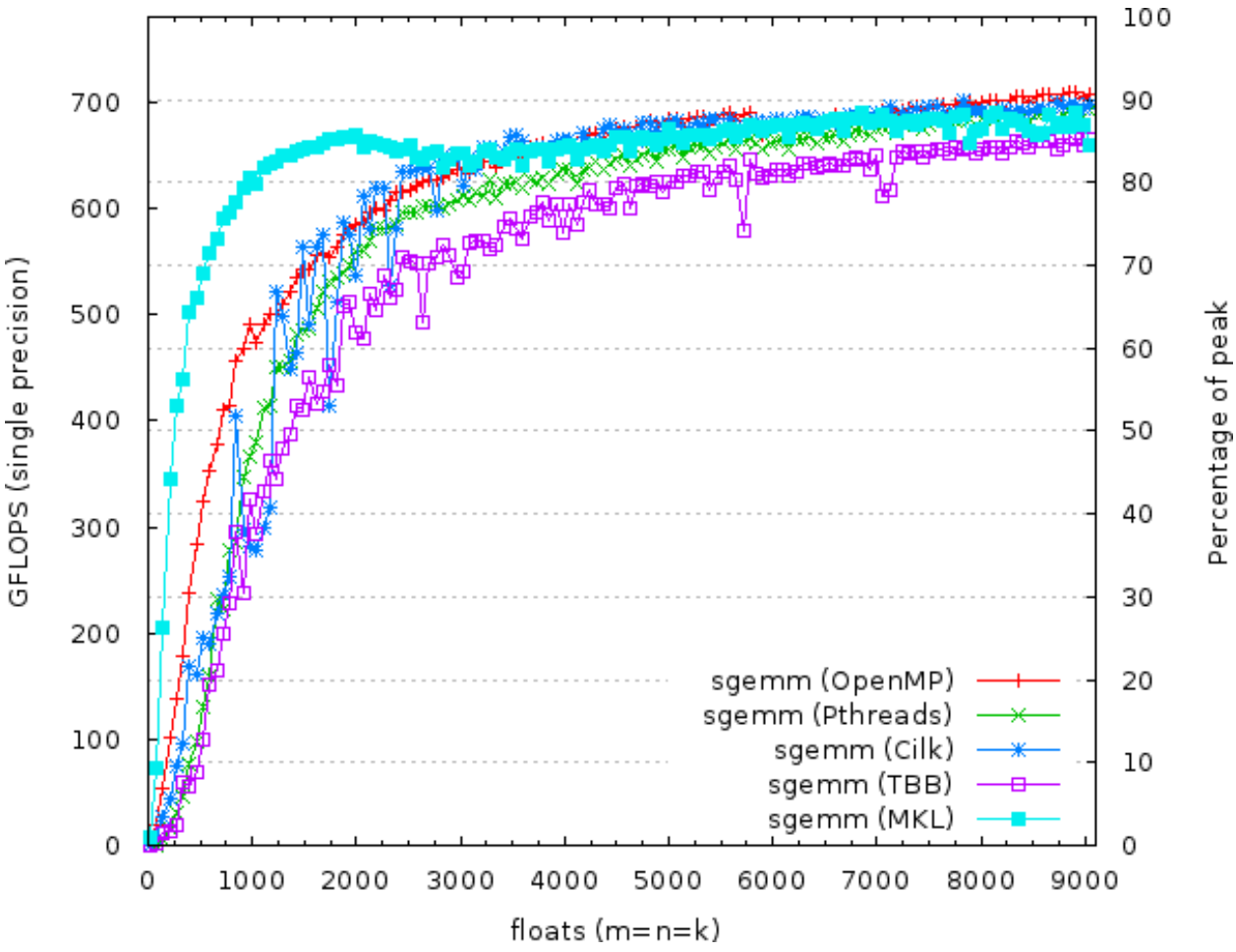

Figure 6.20: Single precision comparison of parallel languages for GEMM routine on Sandy Bridge

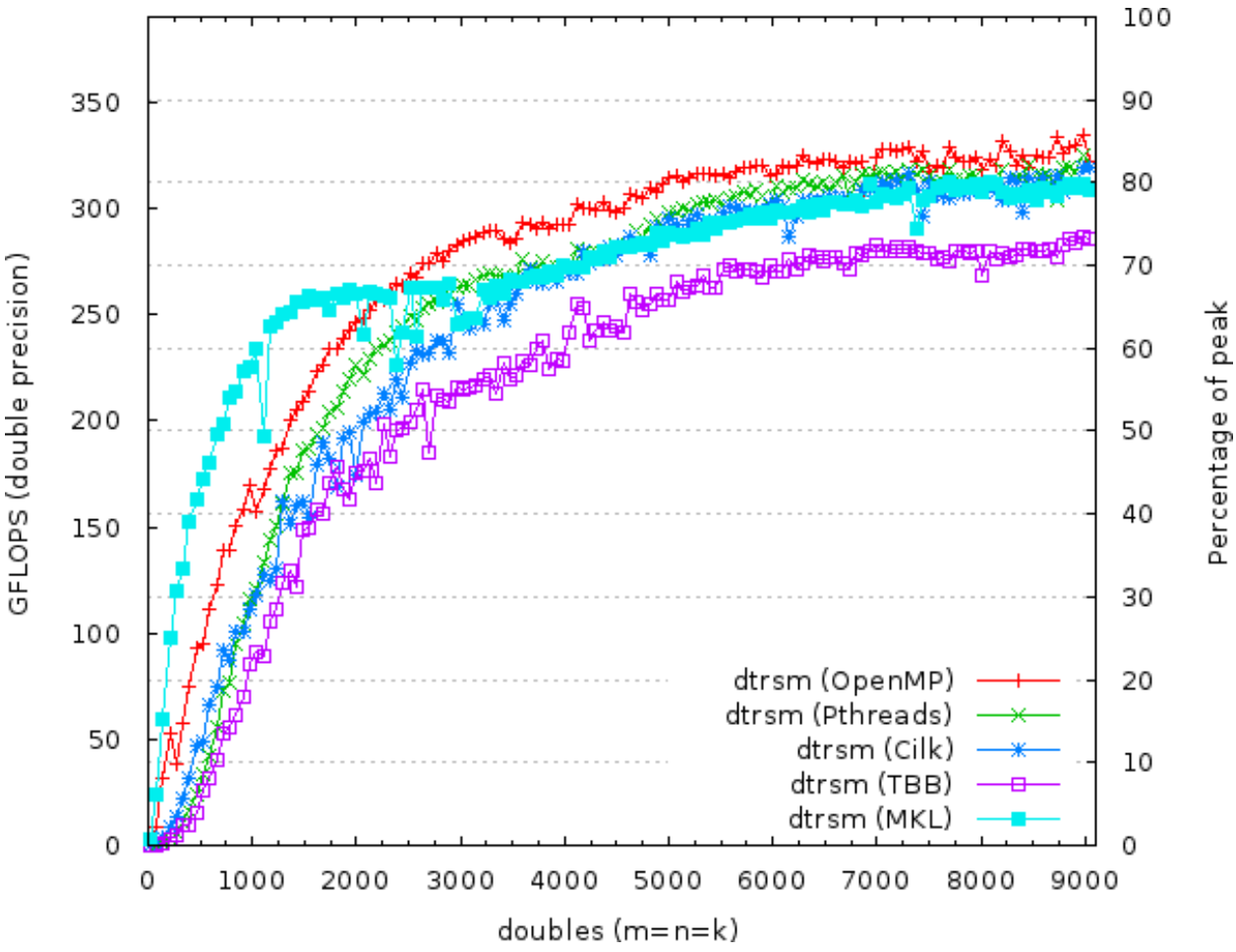

Figure 6.21: Double precision comparison of parallel languages for TRSM routine on Sandy Bridge

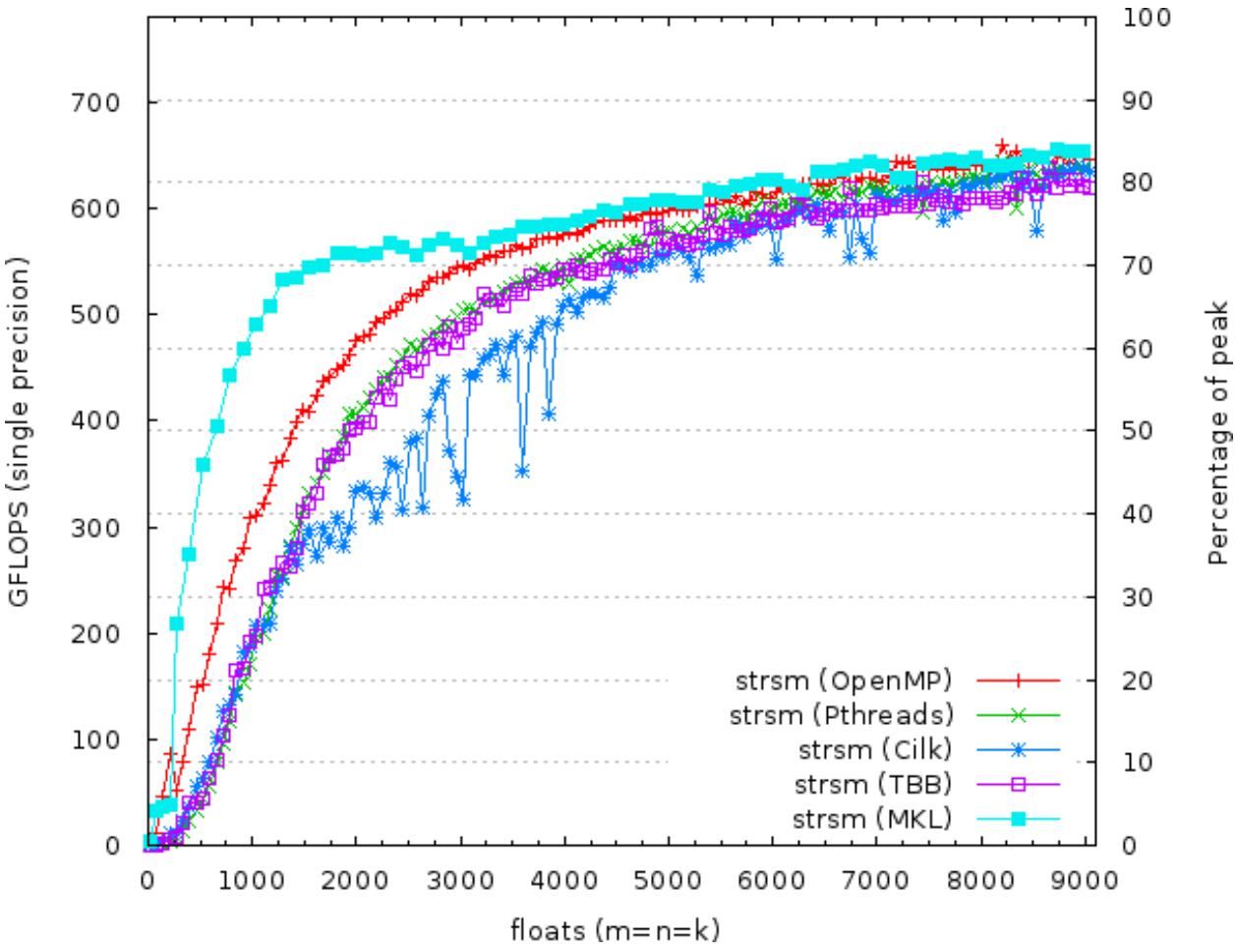

Figure 6.22: Single precision comparison of parallel languages for TRSM routine on Sandy Bridge

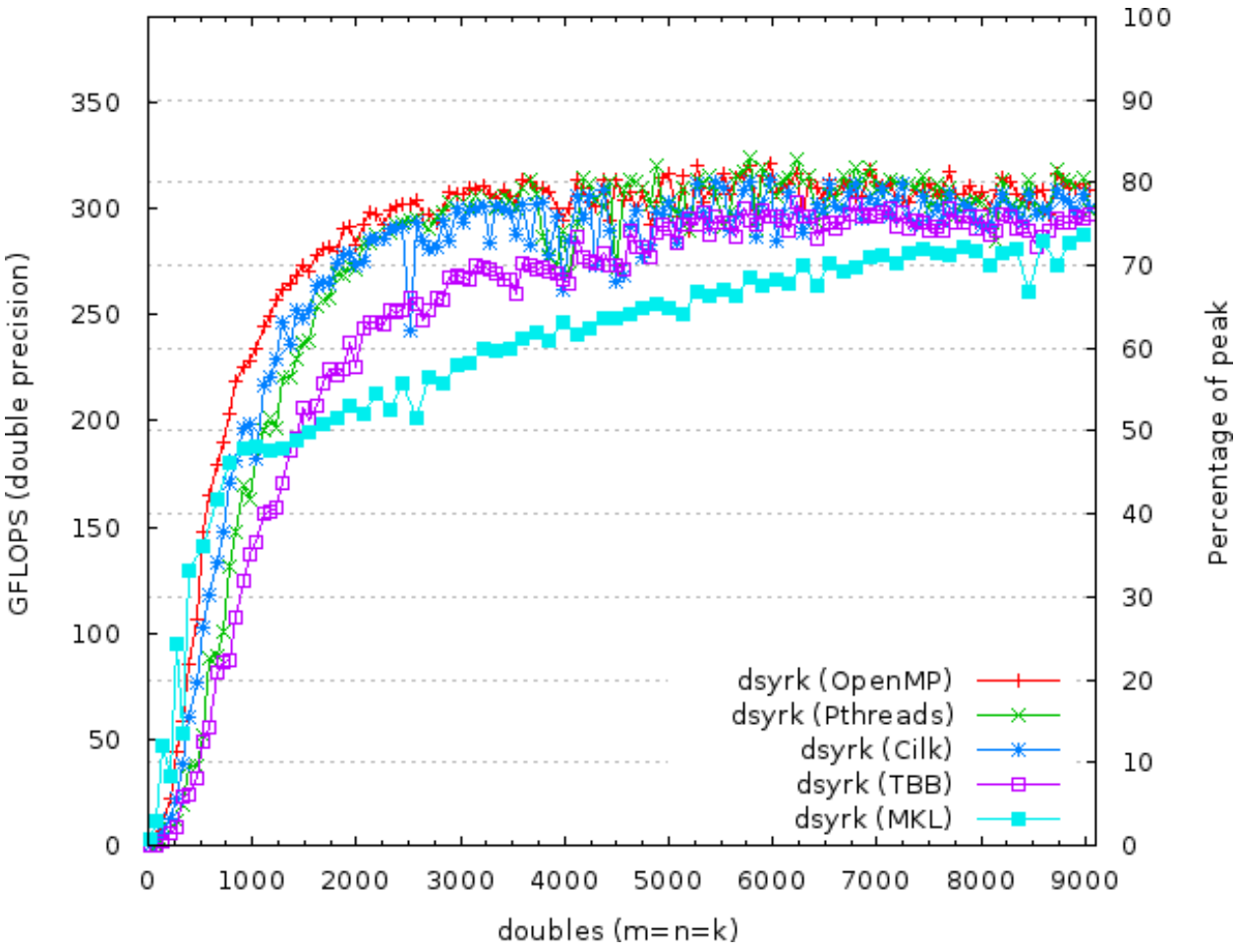

Figure 6.23: Double precision comparison of parallel languages for SYRK routine on Sandy Bridge

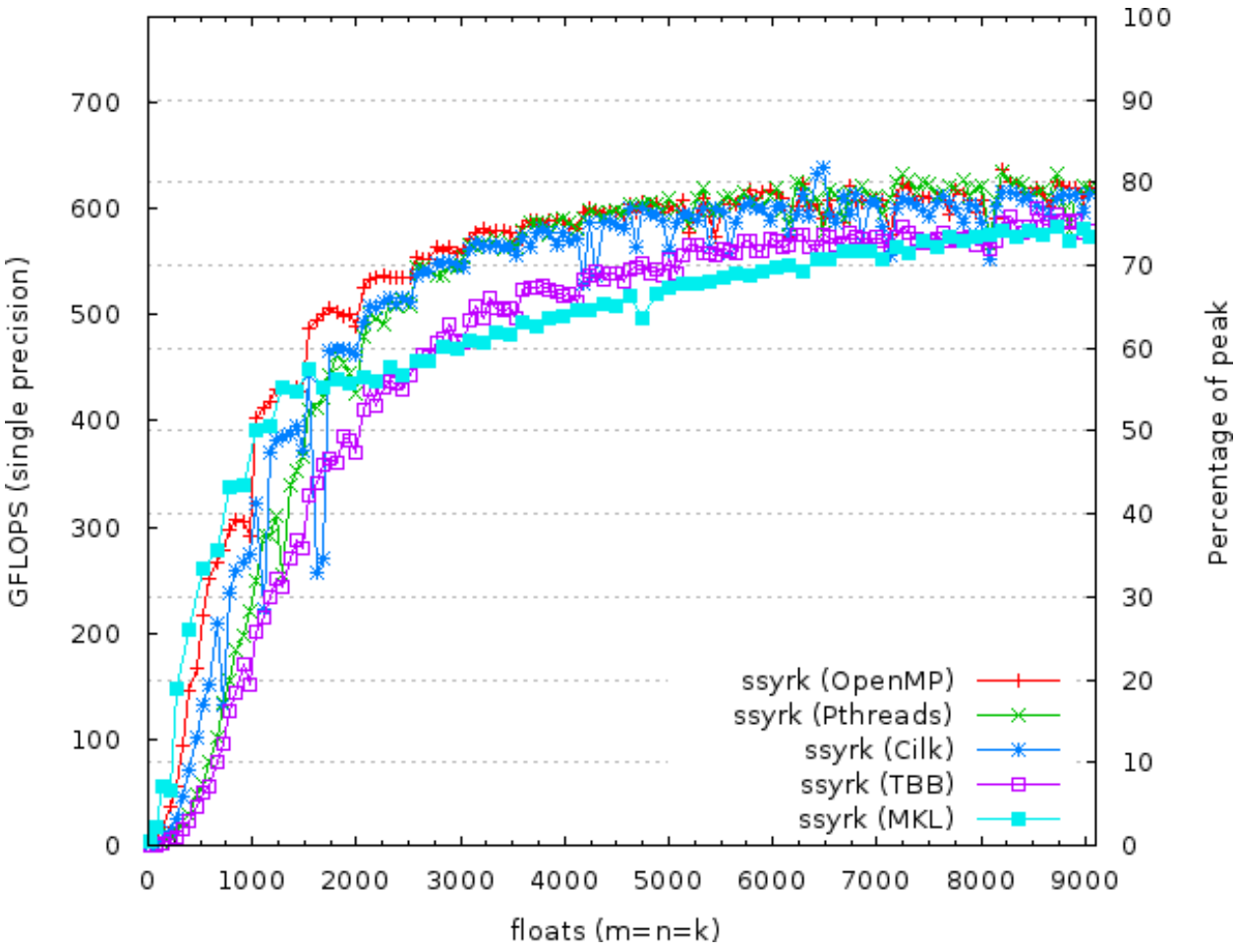

Figure 6.24: Single precision comparison of parallel languages for SYRK routine on Sandy Bridge

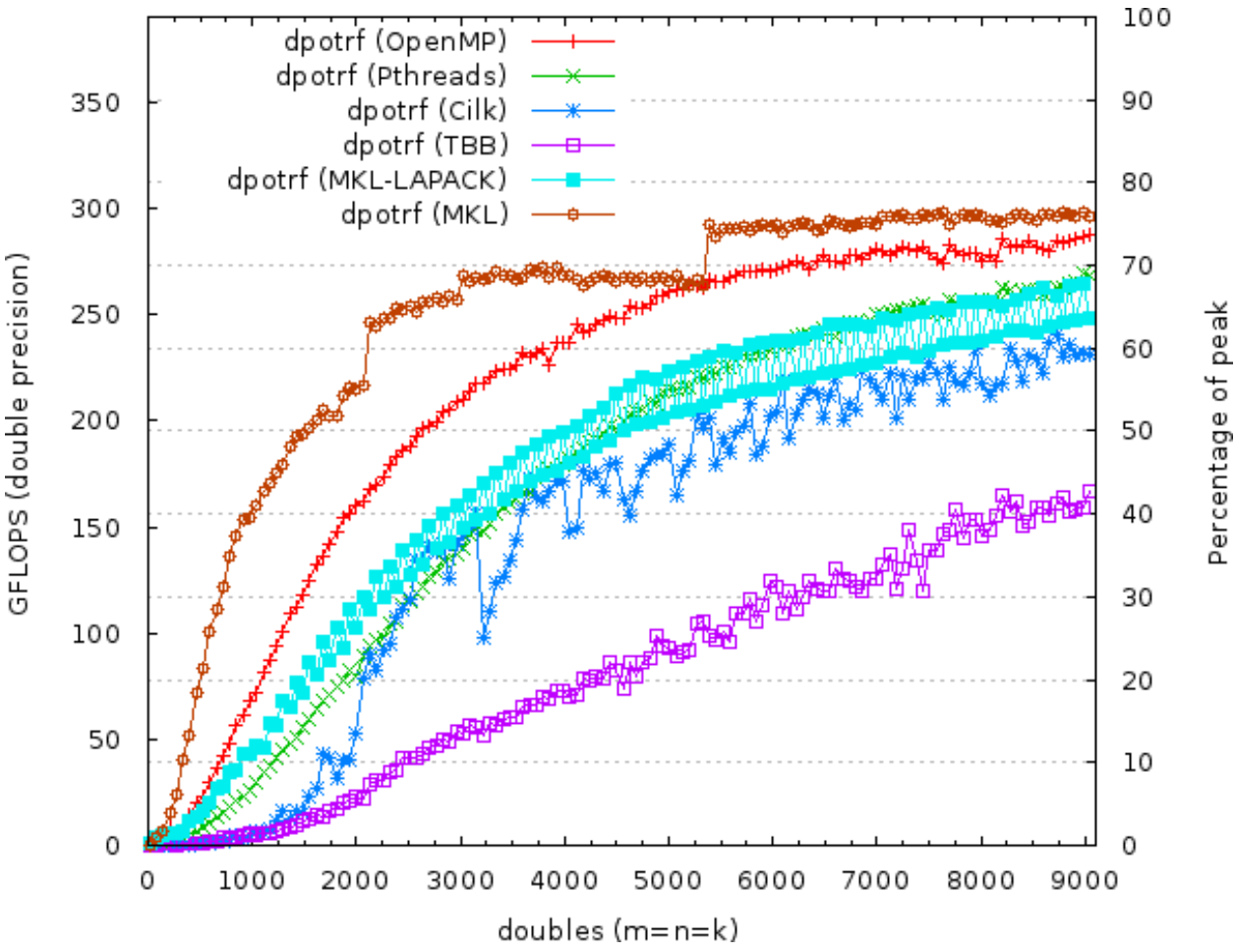

Figure 6.25: Double precision comparison of parallel languages for POTRF routine on Sandy Bridge

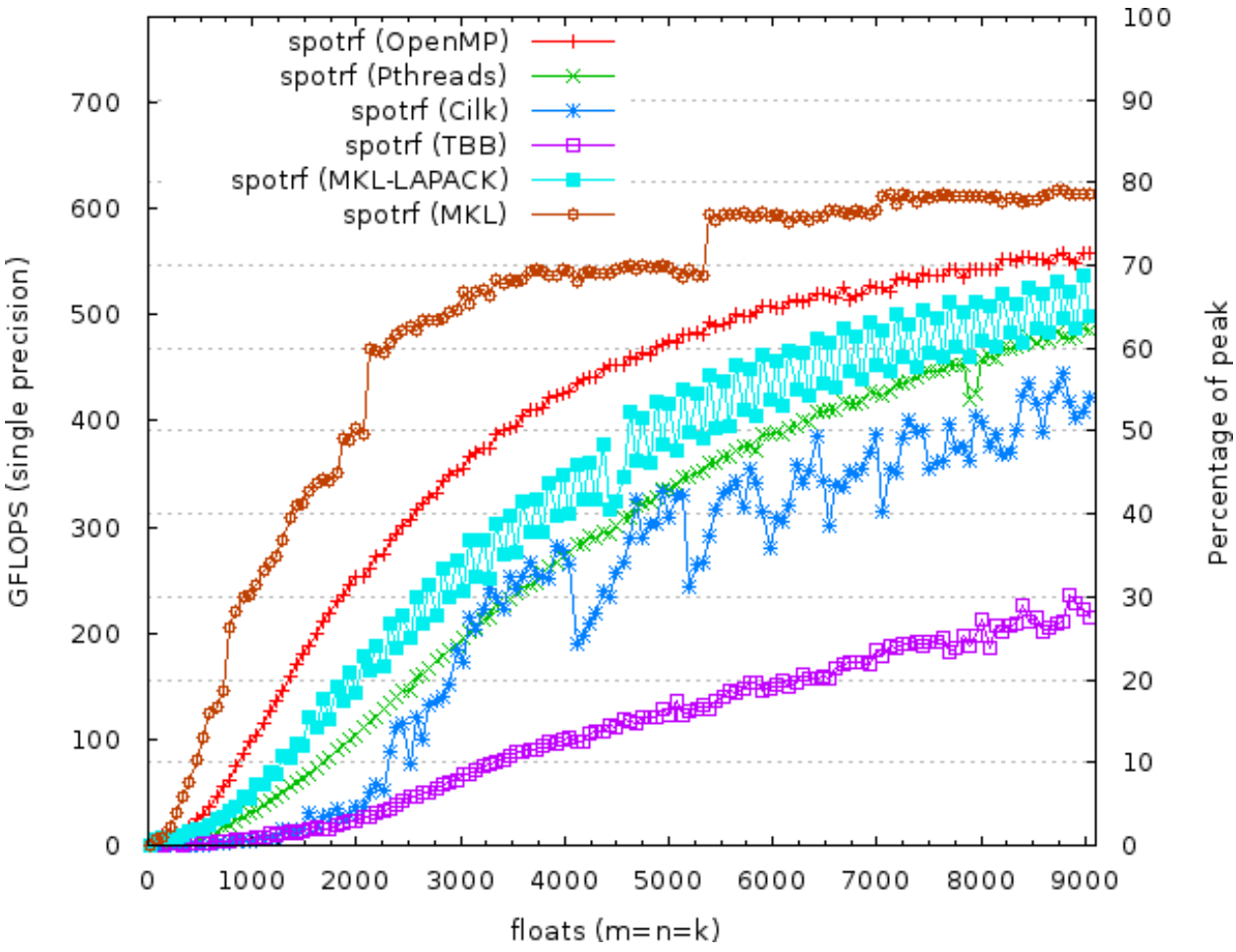

Figure 6.26: Single precision comparison of parallel languages for POTRF routine on Sandy Bridge

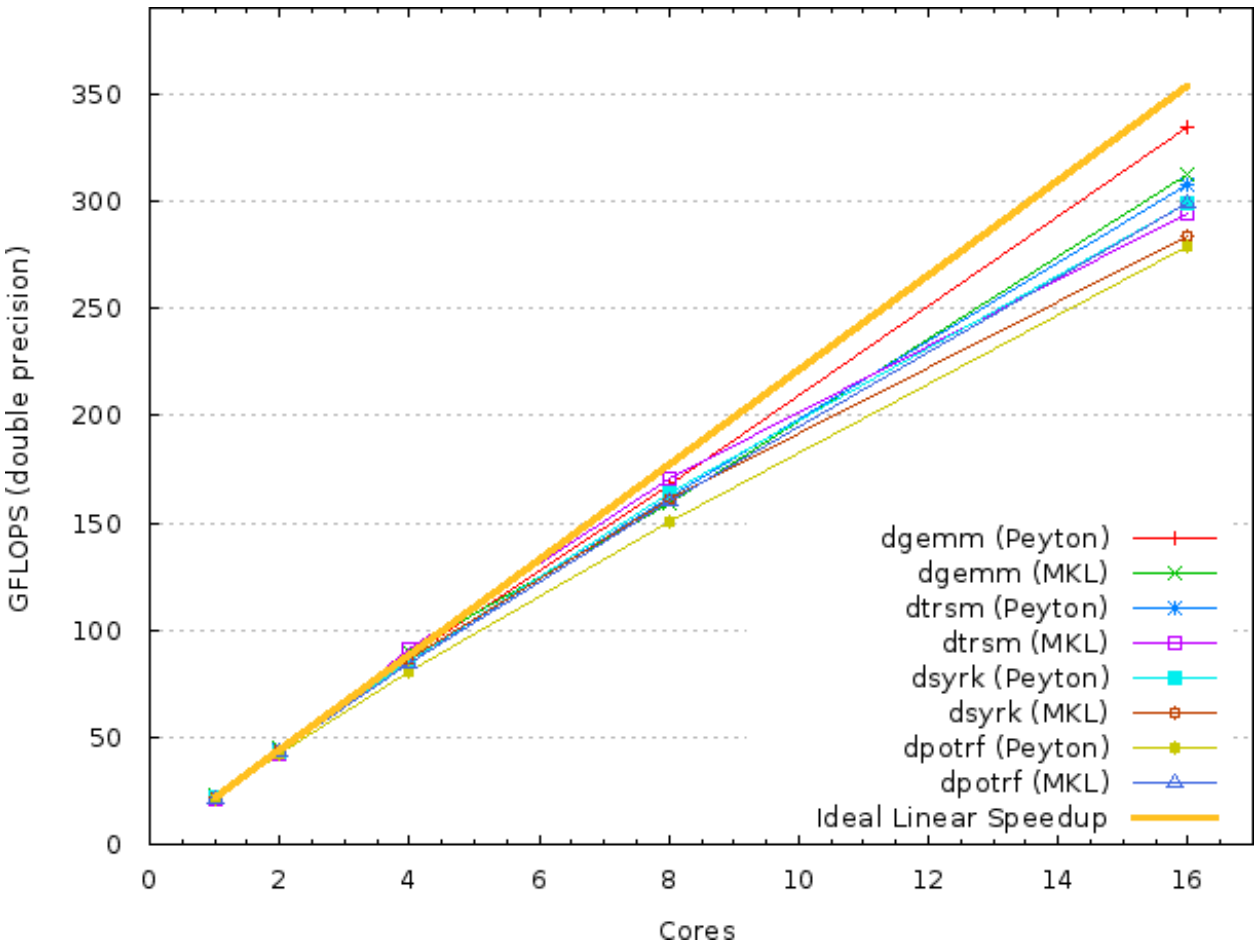

Figure 6.27: Double precision scaling for Sandy Bridge up to 16 cores on Sandy Bridge. For each core count, the size $m = n = k = 9100$ was used.

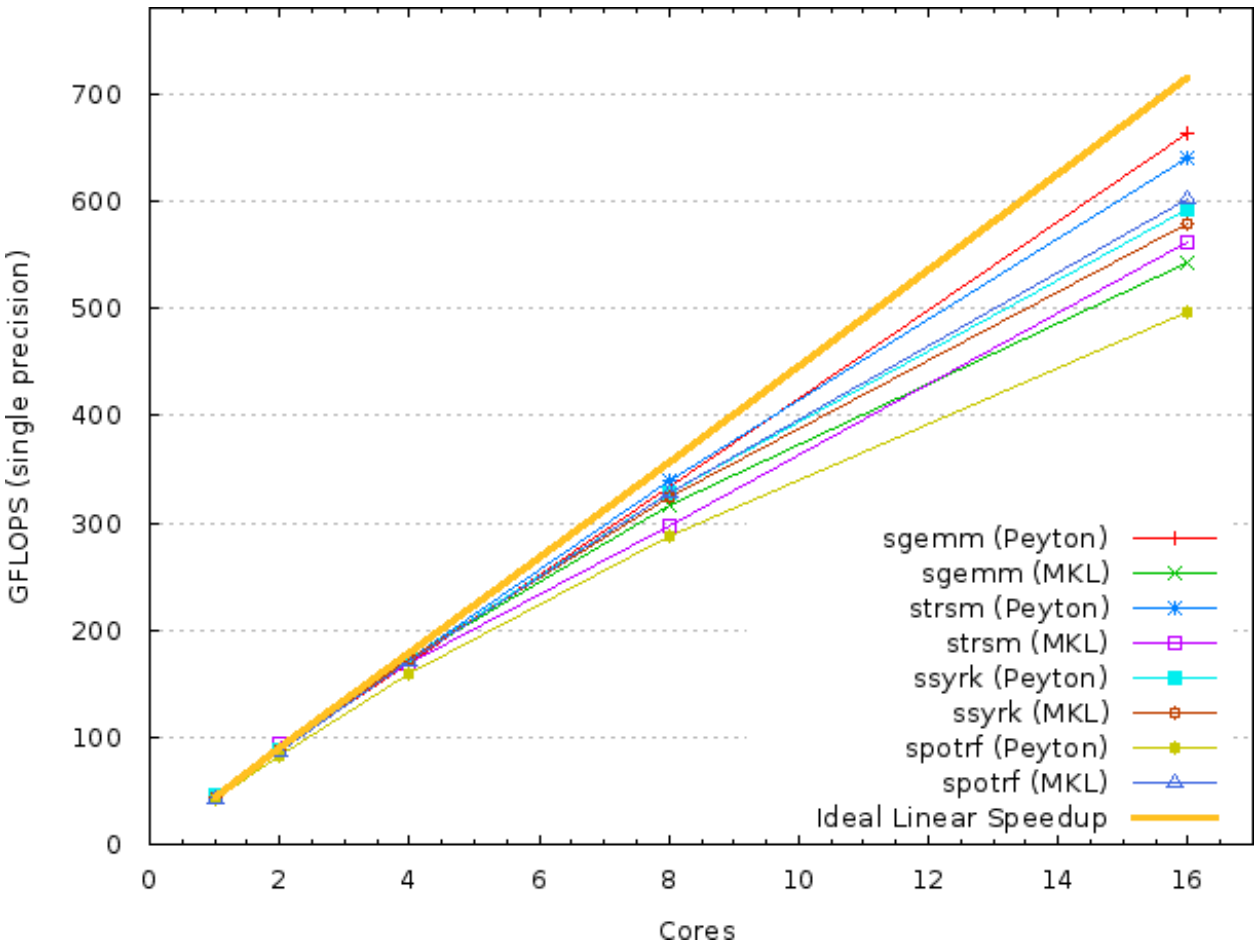

Figure 6.28: Single precision scaling for Sandy Bridge up to 16 cores on Sandy Bridge. For each core count, the size $m = n = k = 9100$ was used.

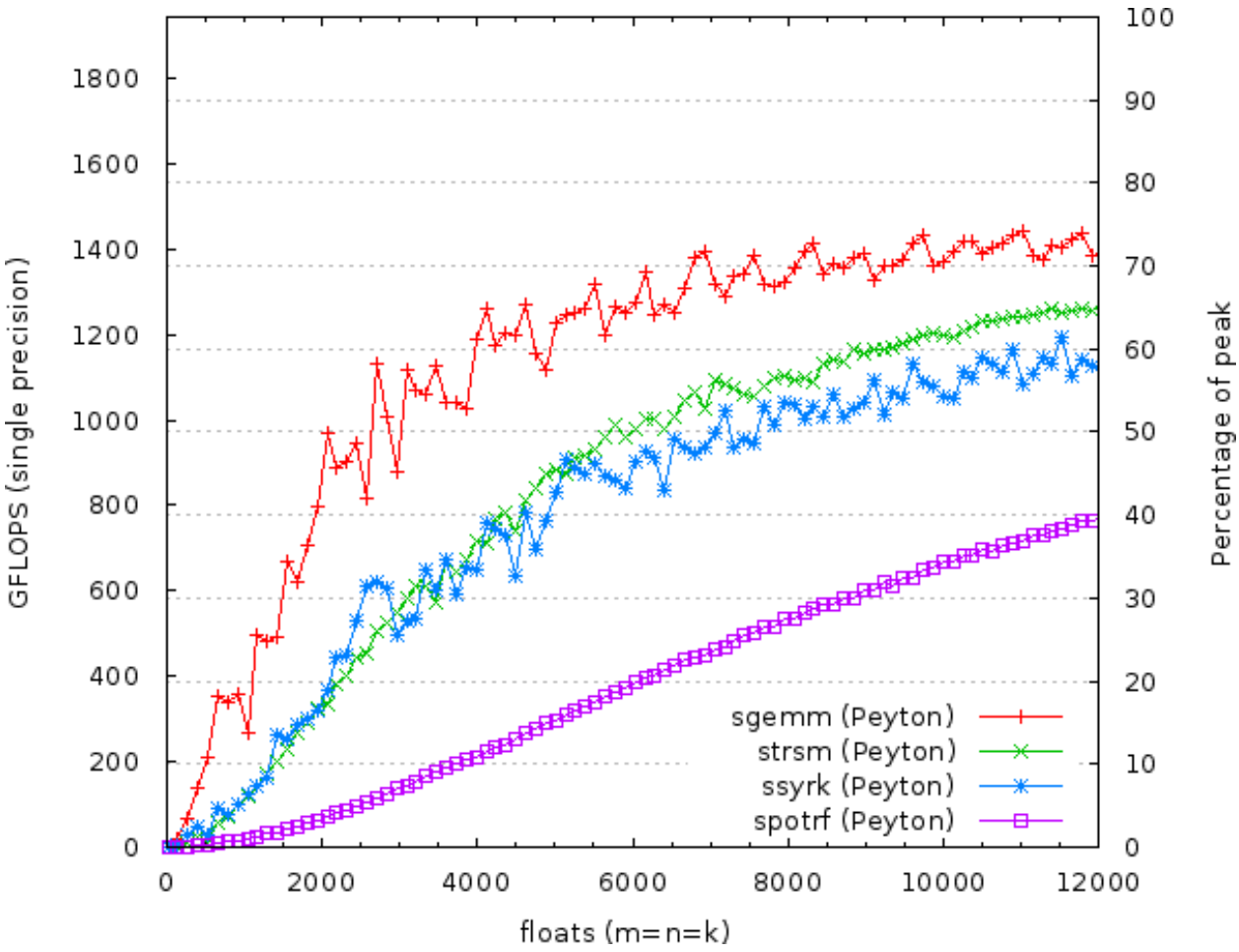

Figure 6.29: Single precision parallel codes using all cores on MIC

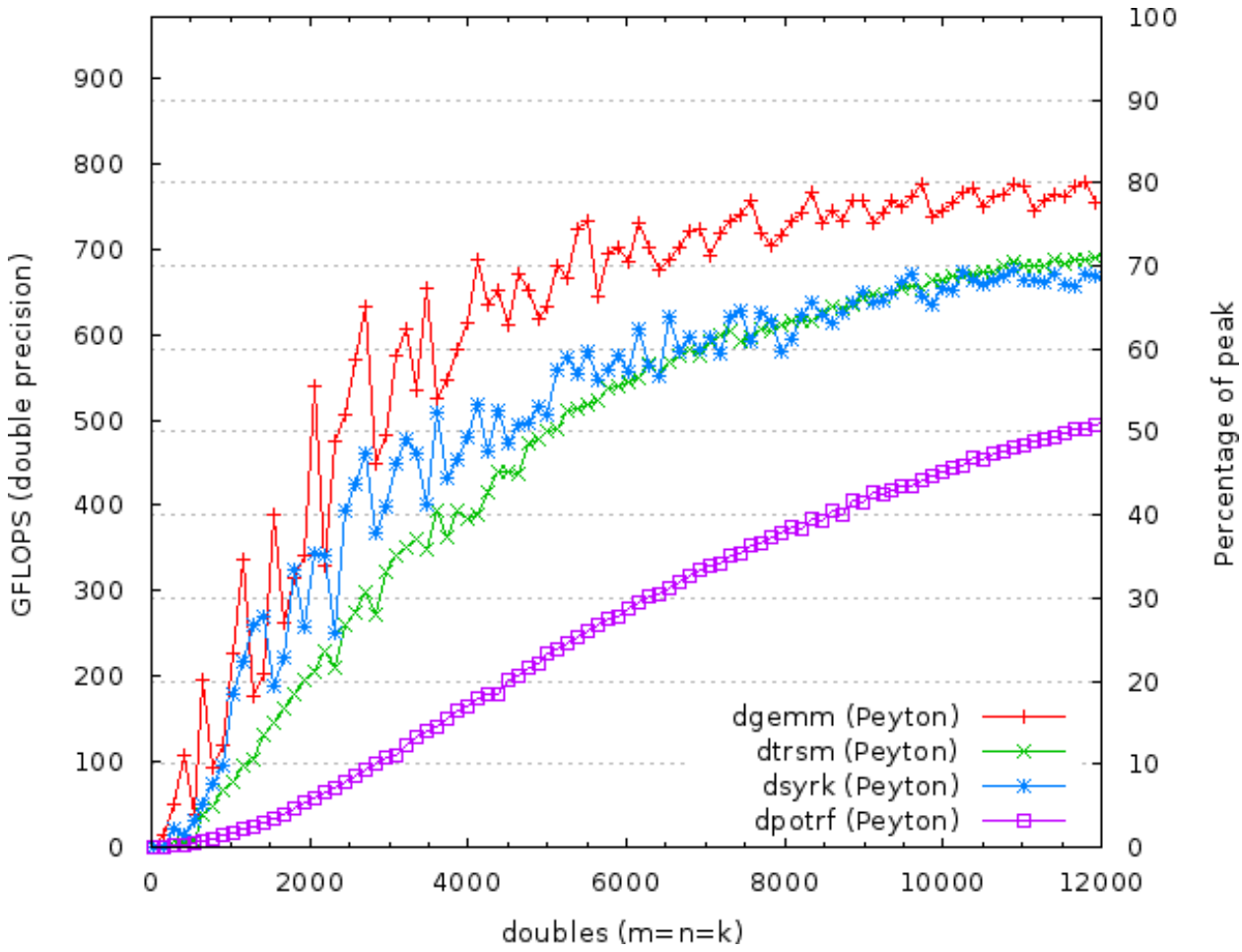

Figure 6.30: Double precision parallel codes using all cores on MIC

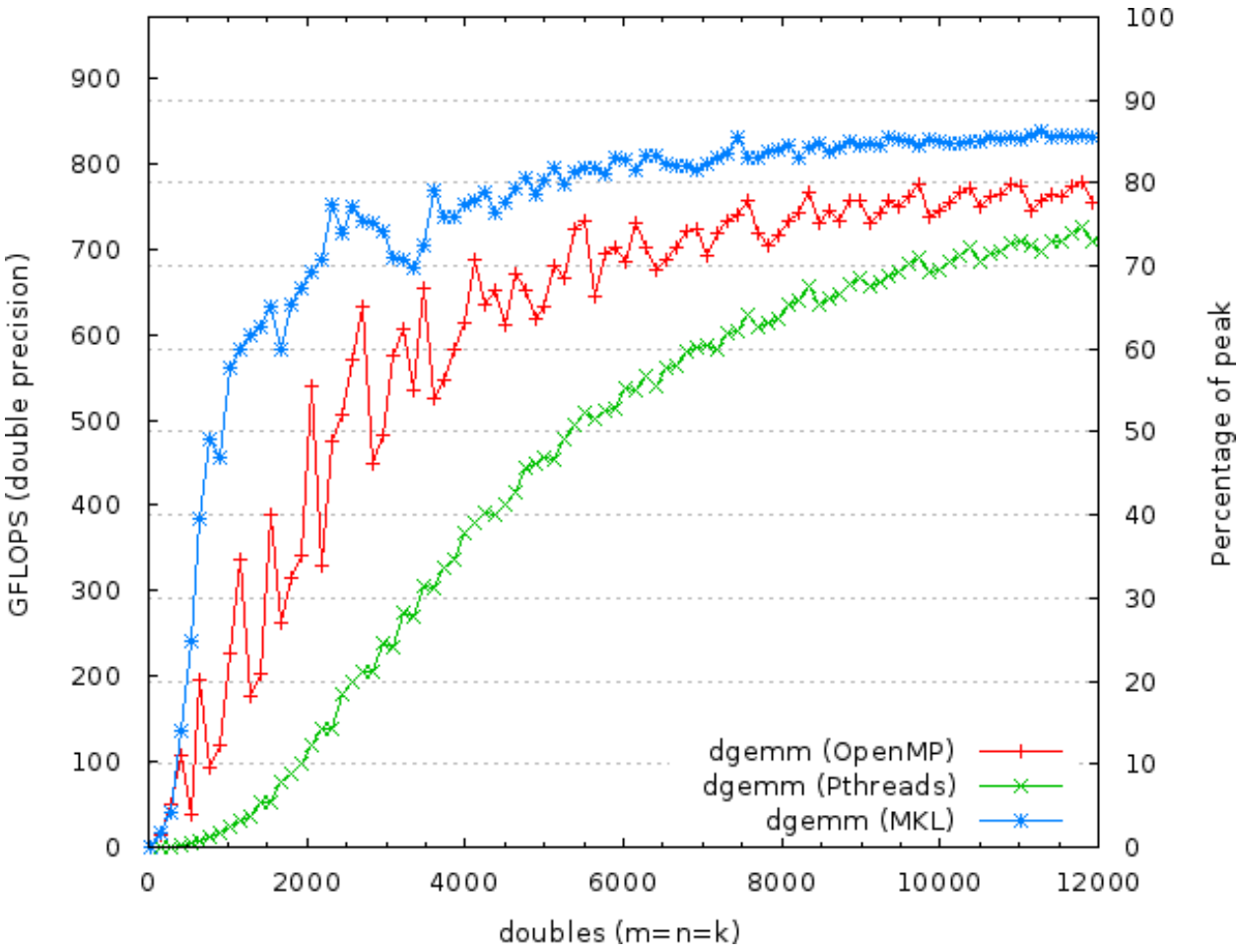

Figure 6.31: Double precision comparison of parallel languages for GEMM routine on MIC

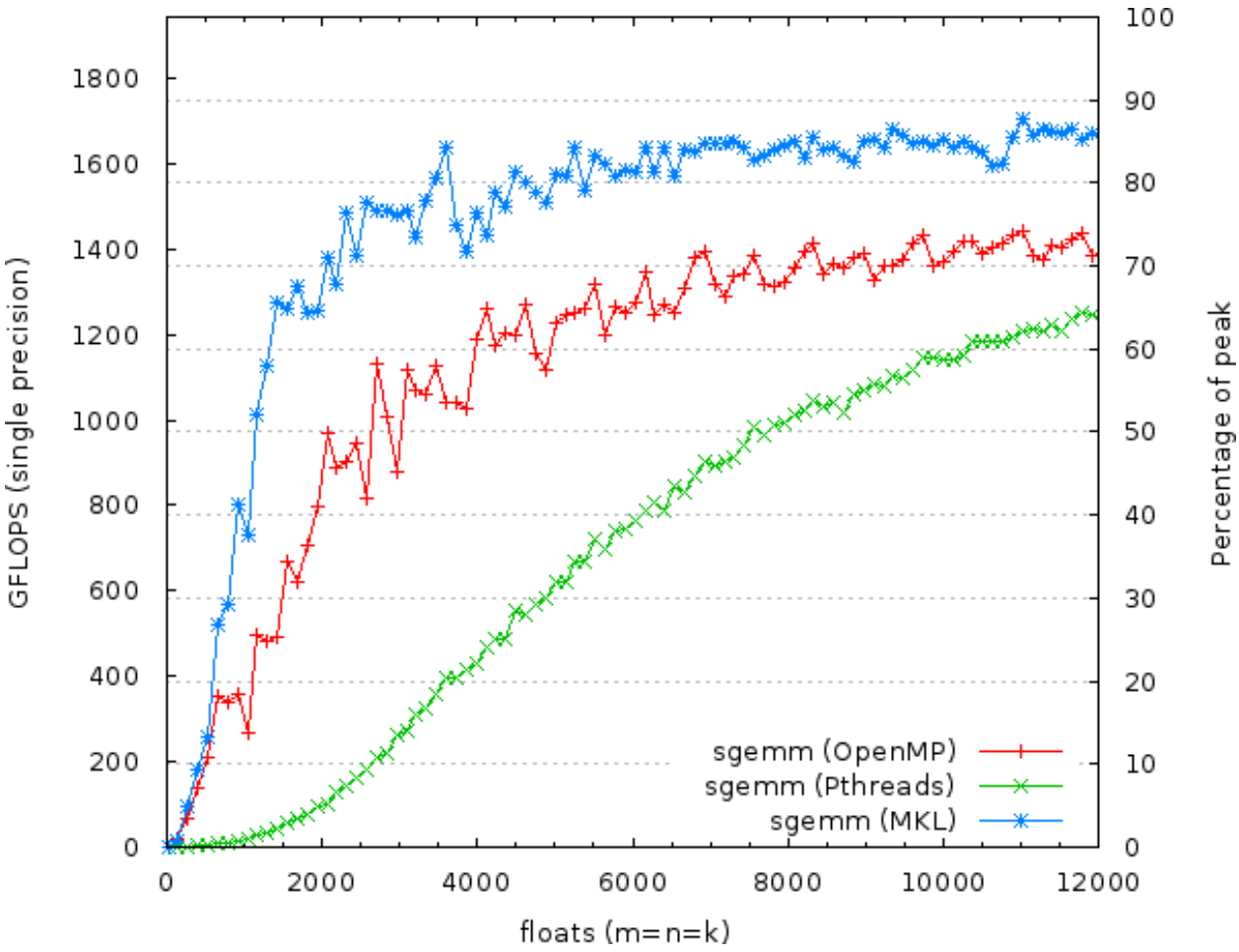

Figure 6.32: Single precision comparison of parallel languages for GEMM routine on MIC

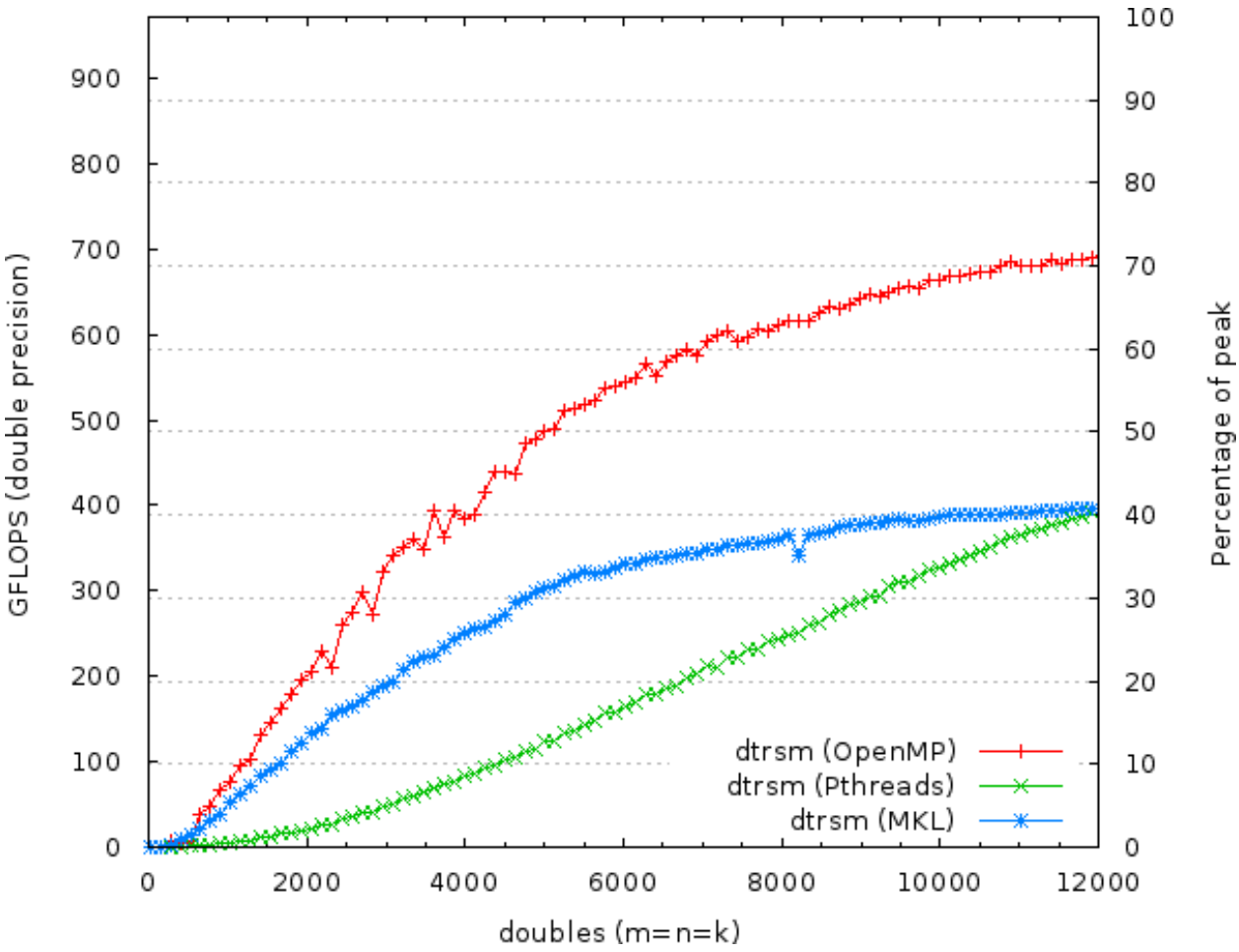

Figure 6.33: Double precision comparison of parallel languages for TRSM routine on MIC

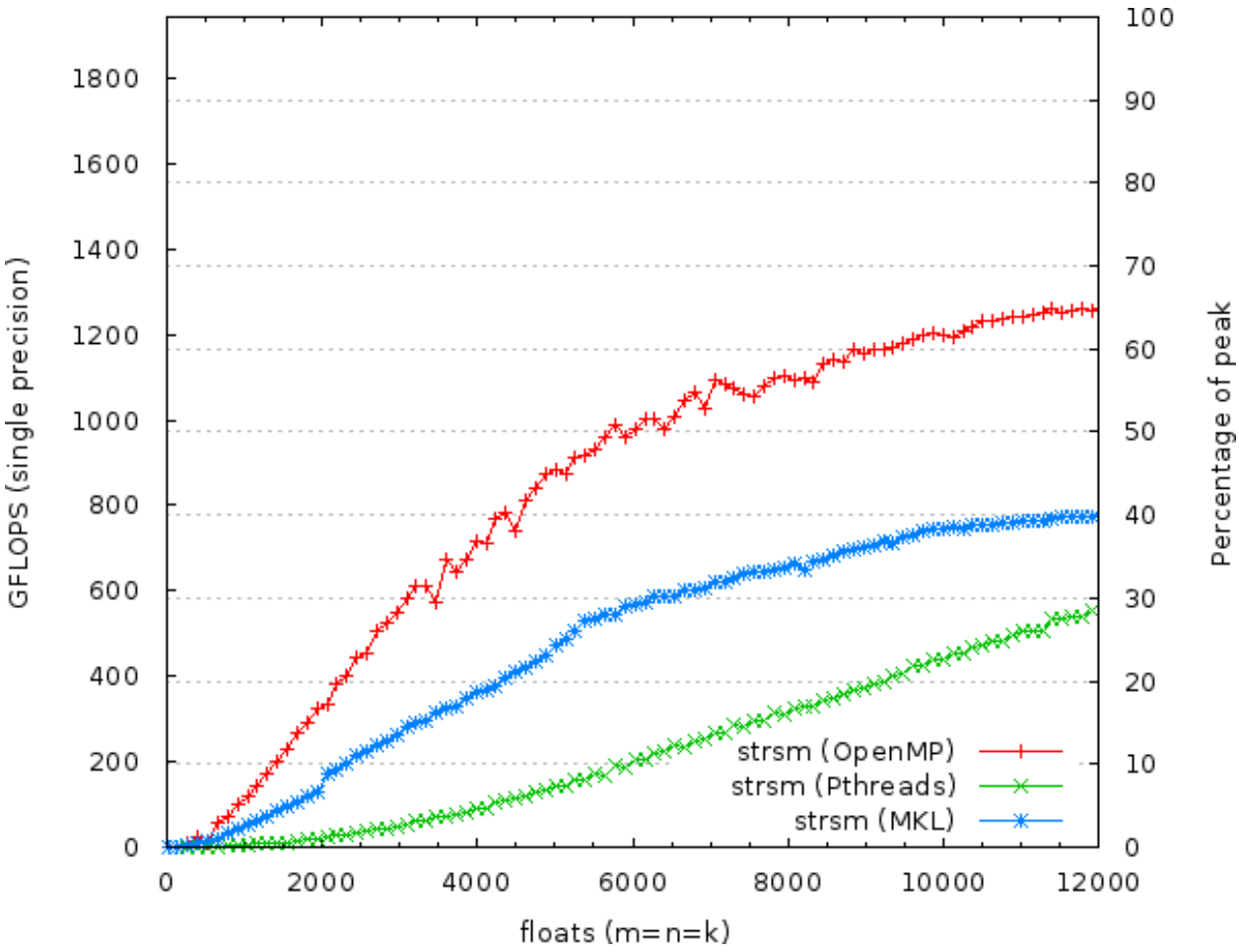

Figure 6.34: Single precision comparison of parallel languages for TRSM routine on MIC

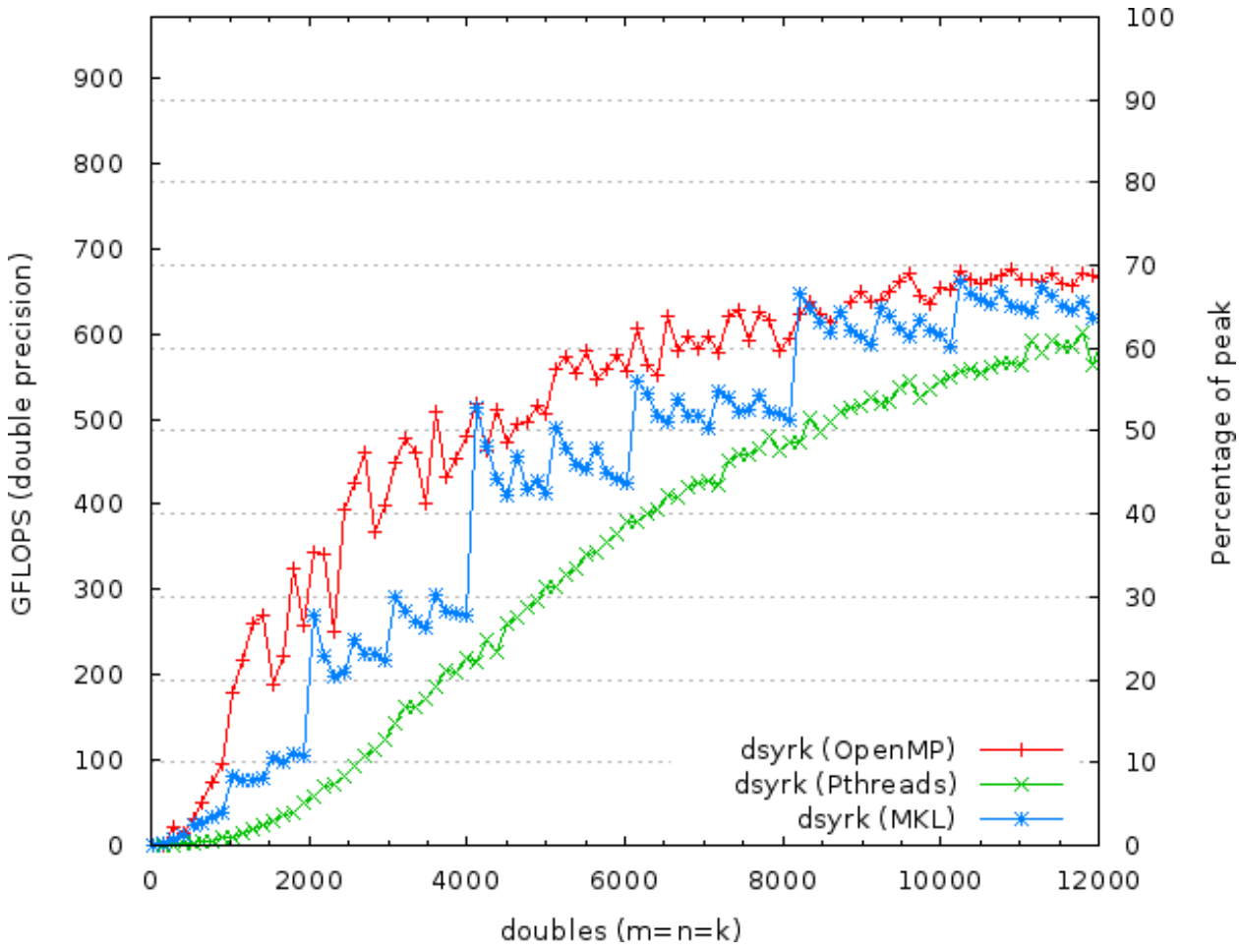

Figure 6.35: Double precision comparison of parallel languages for SYRK routine on MIC

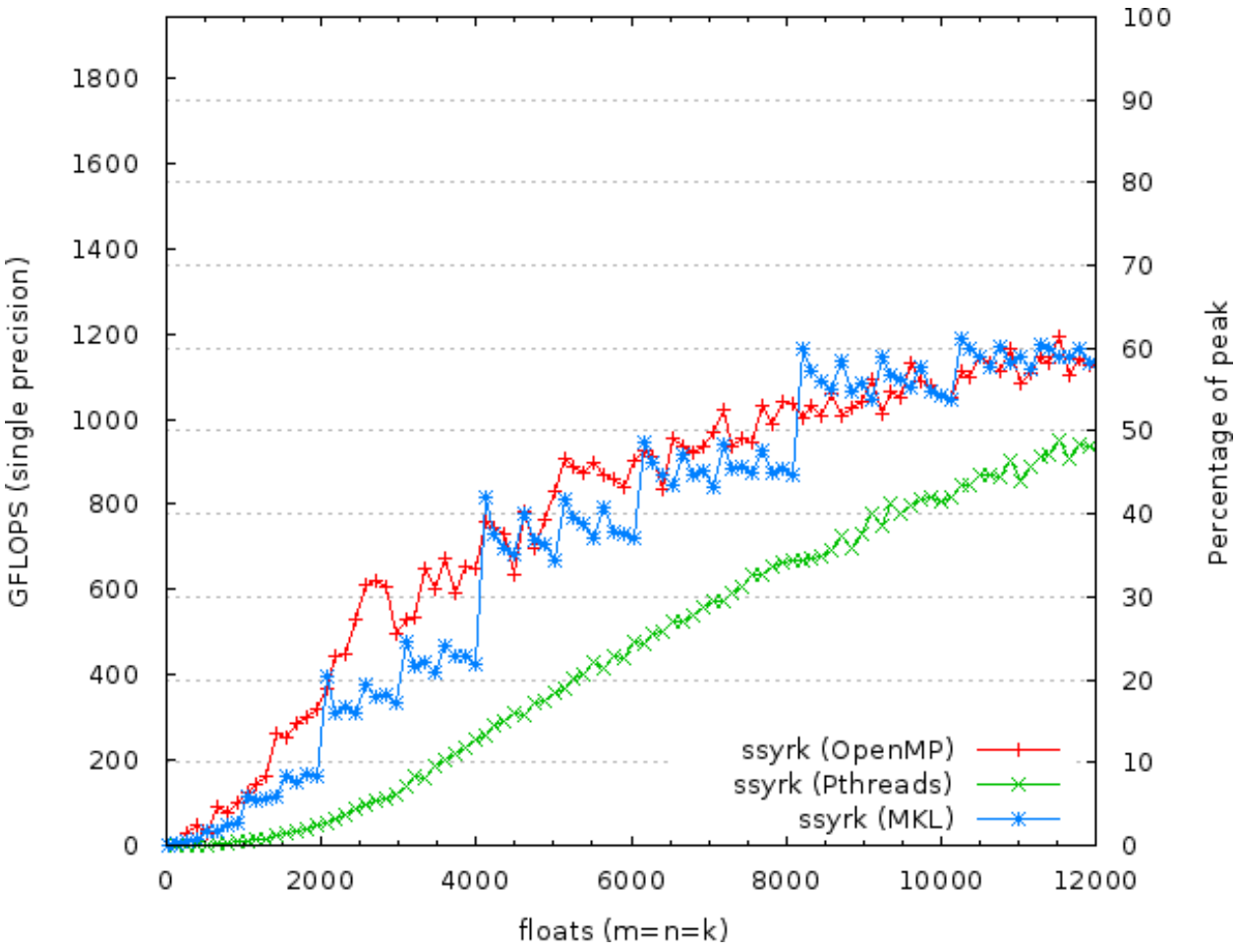

Figure 6.36: Single precision comparison of parallel languages for SYRK routine on MIC

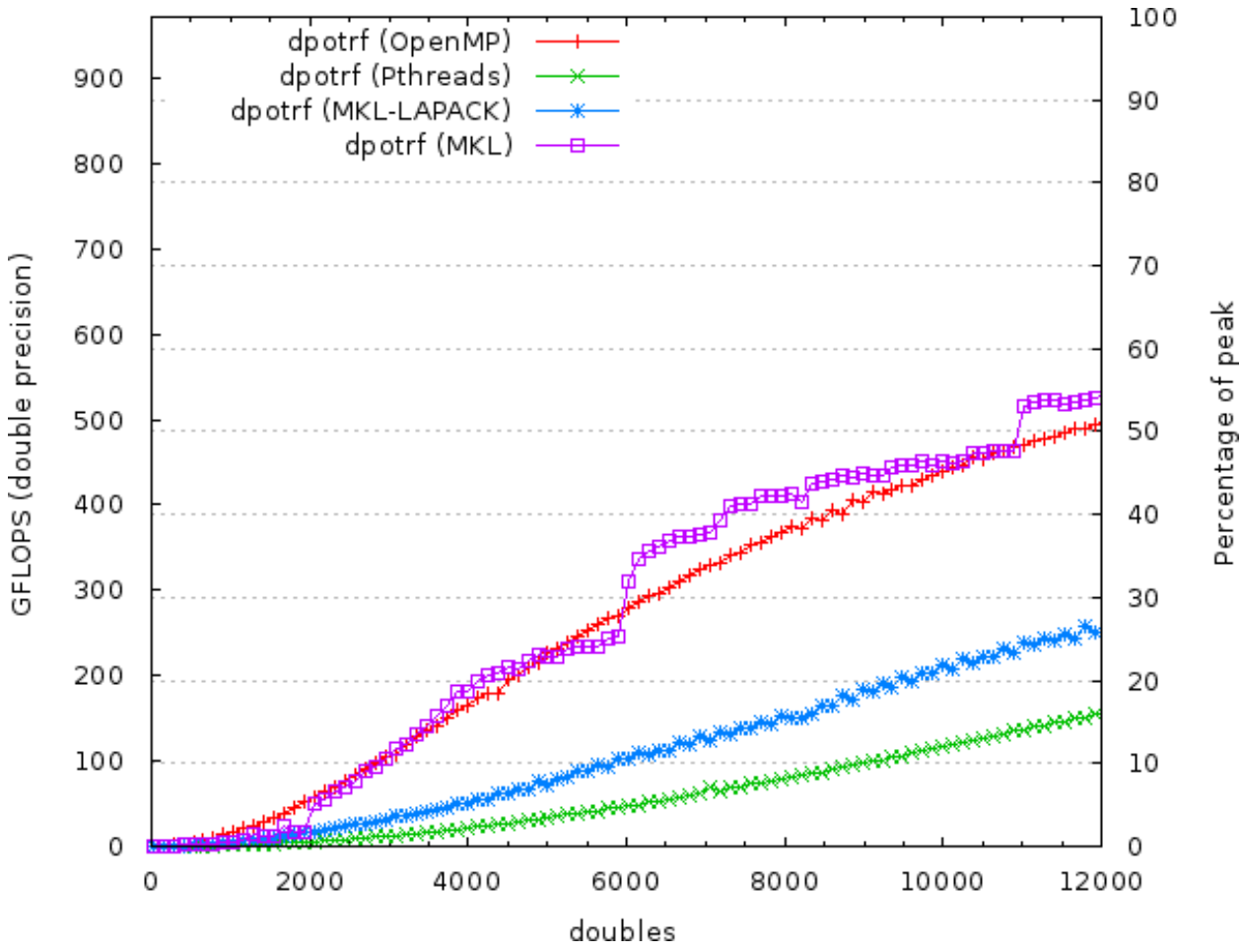

Figure 6.37: Double precision comparison of parallel languages for POTRF routine on MIC

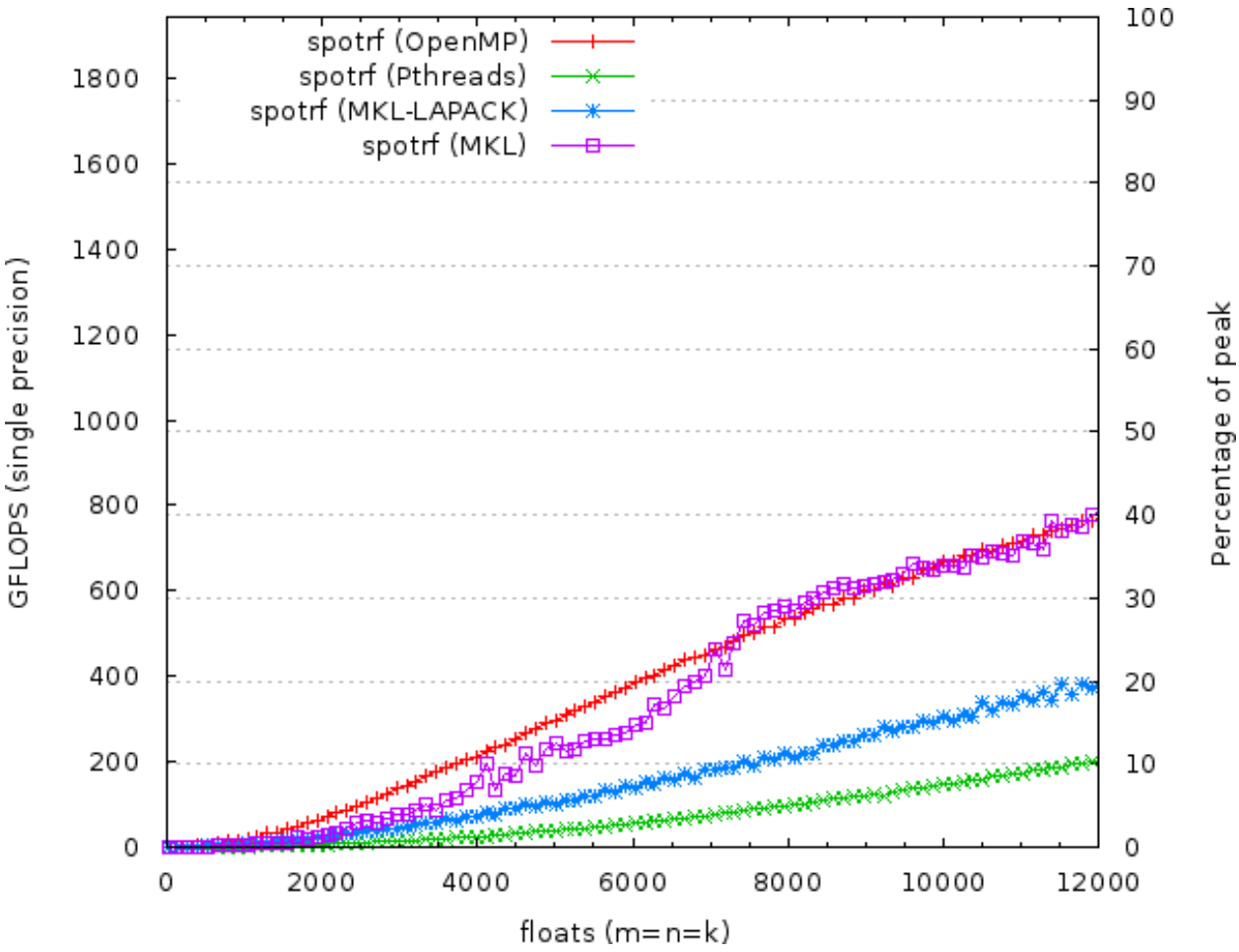

Figure 6.38: Single precision comparison of parallel languages for POTRF routine on MIC

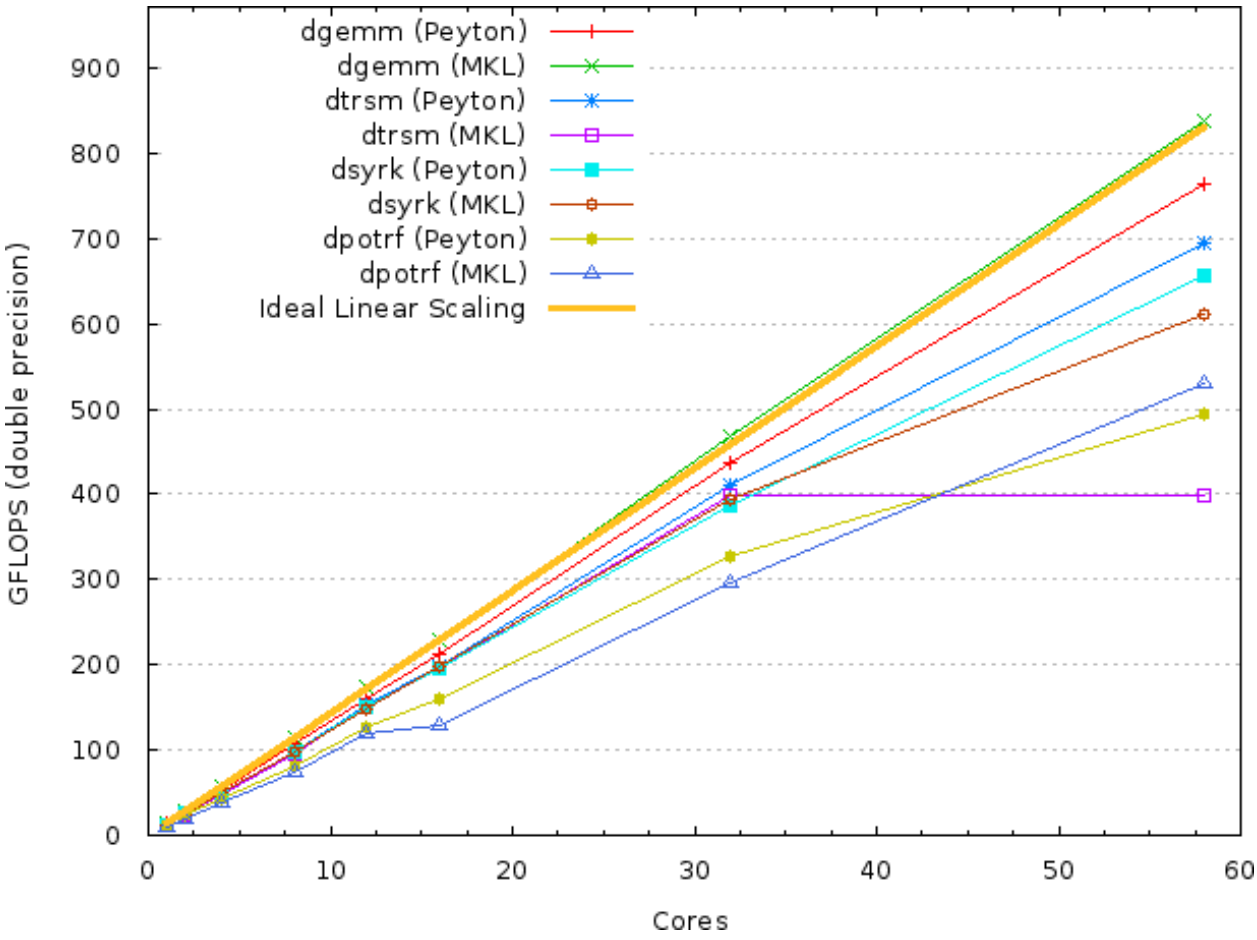

Figure 6.39: Double precision scaling for MIC up to 58 cores on MIC. For each core count, the size $m = n = k = 12000$ was used.

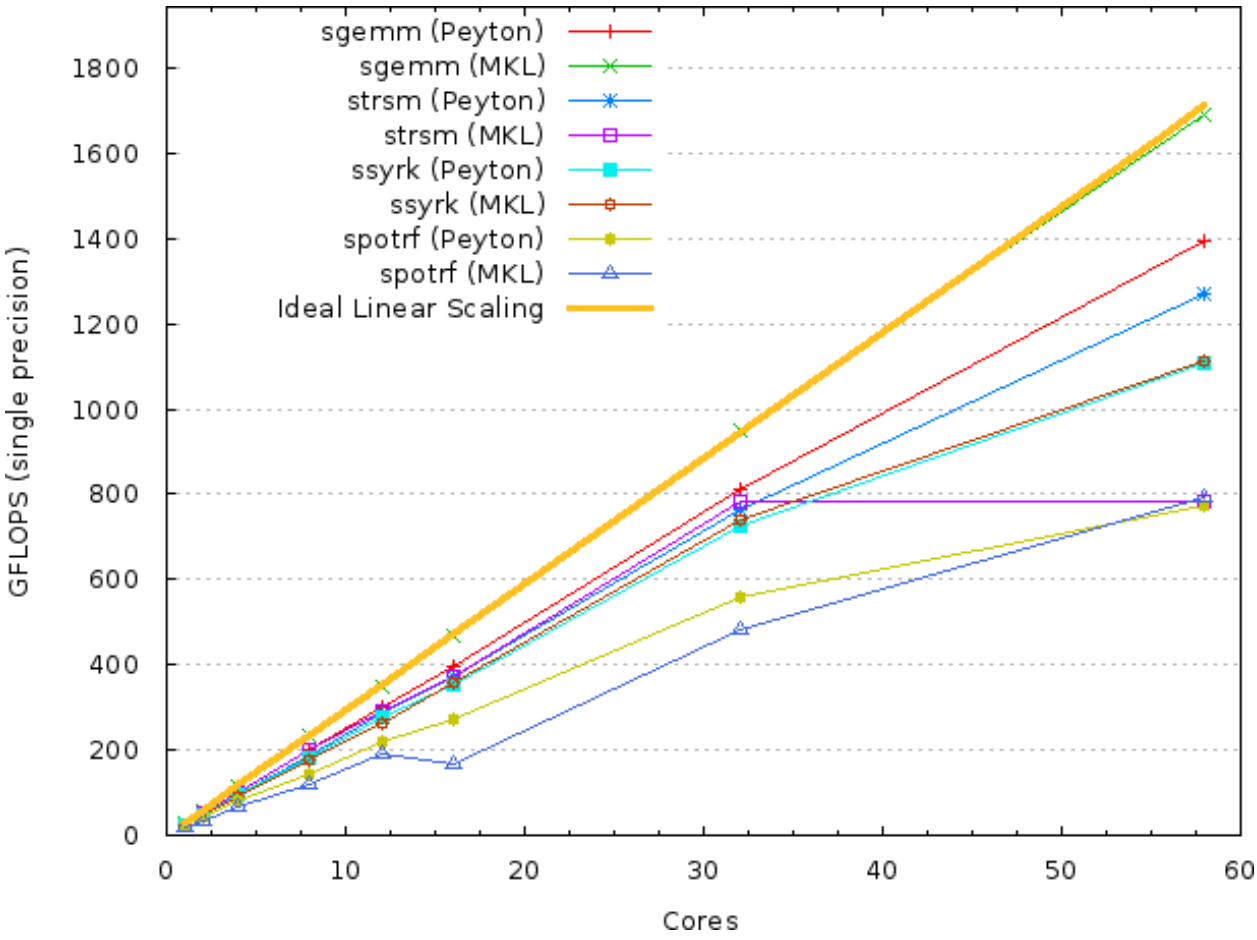

Figure 6.40: Single precision scaling for MIC up to 58 cores on MIC. For each core count, the size $m = n = k = 12000$ was used.

Chapter 7

Conclusions

To overview, this research attempted to map the DGEMM, TRSM, SYRK, and POTRF routines onto Intel architectures as efficiently as possible taking note of the different techniques used to improve performance. The different parallel languages were then examined to see how one should implement this type of application. This development exposed a set of conditions which must be met to achieve high performance. The three conditions are:

1. Taking care of the cache hierarchy (Blocking)
2. Taking care of the TLB (Packing)
3. Inner Kernel, accesses elements of the source matrices so cache latencies are hidden

If even one of these conditions is not met, the performance can suffer a great deal. The blocking scheme chosen for GEMM was explained in great detail with results on par with MKL. The packing was shown to improve TLB references by making all read-only data accessed contiguously. The inner kernel was structured in a way to hide cache latencies and issue as many computational instructions per cycle as possible. Prefetching, which is critical for MIC, was touched on to show how memory latencies could be avoided by software pipelining. The results for all the routines were on par with MKL with some BLAS routines achieving over 90% peak performance.

Next, the parallel implementations were developed and optimized. The languages were judged on programming difficulty and performance. During development, one particular feature was established as the dominating feature that allowed high performance. This is hardware or thread affinity, the ability to map certain threads to cores. With this ability, the user decides how many cores are used. Without it, the runtime system of the parallel language is often the determining factor and can leave cores idle. With all factors considered, OpenMP prevailed as the best option for dense linear algebra kernels, beating out Pthreads, Cilk, and TBB. OpenMP is the easiest to program with simple one line pragmas and also attains the best performance. Another feature separating OpenMP from the others was its ubiquitous compiler support.

Similar applications to the Level 3 BLAS should be written this way as well. Some characteristics that make up the routines in this research are:

1. Highly regular data accesses
2. Vectorizable
3. Simple mapping between tasks and processors
4. High amount of data independence

If an application has these characteristics, it should be written in OpenMP and probably use intrinsics for vectorization. The simple mapping between tasks and processors is important because load balancing is simple to implement for an application like GEMM, it is simple to partition the matrix in to even amounts of work for processors. This typically results in a one-to-one mapping between tasks and processors. This combined with the ability to control hardware affinity allows OpenMP the greatest amount of performance and programmability.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965. 1

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. 1

[3] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st ed., 2010. 1

[4] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010. 1

[5] F. D. Igual, G. Quintana-ort, and R. A. V. D. Geijn, "Level-3 blas on a gpu: Picking the low hanging fruit," 2009. 1

[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley, "An updated set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, pp. 135–151, June 2002. 2, 4, 58

[7] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: a portable linear algebra library for high-performance computers," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, (Los Alamitos, CA, USA), pp. 2–11, IEEE Computer Society Press, 1990. 2

[8] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, pp. 12:1–12:25, May 2008. 2, 15, 18, 19, 21, 25

[9] B. Kgstrm, P. Ling, and C. V. Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark," *ACM*

TRANSACTIONS ON MATHEMATICAL SOFTWARE, vol. 24, no. 3, pp. 268–302, 1998. 2

[10] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 blas," *ACM Trans. Math. Softw.*, vol. 35, pp. 4:1–4:14, July 2008. 2

[11] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst, *Numerical Linear Algebra for High Performance Computers*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1998. 4

[12] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. No. 248966-026, April 2012. 8, 29

[13] S. U. Thiagarajan, C. Congdon, S. Naik, and L. Q. Nguyen, "Intel® Xeon Phi™ Coprocessor Developer's Quick Start." http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide, 2012. 10

[14] N. Park, B. Hong, and V. K. Prasanna, "Tiling, block data layout, and memory hierarchy performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, pp. 640–654, July 2003. 18, 21

[15] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, pp. 101–121, February 2005. 18

[16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 4th ed., 2008. 18

[17] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63–74, 1991. 18

[18] G. H. Golub and C. F. Van Loan, *Matrix computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996. 18

[19] J. A. Gunnels, G. M. Henry, and R. A. v. d. Geijn, "A family of high-performance matrix multiplication algorithms," in *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, (London, UK, UK), pp. 51–60, Springer-Verlag, 2001. 18

[20] K. Goto and R. van de Geijn, "On reducing tlb misses in matrix multiplication. FLAME Working Note #9," Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, Nov. 2002. 21

[21] Intel Corporation, *Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual*. 31

[22] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 47

[23] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. 47

[24] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996. 48

[25] I. Corporation, "Intel® C++ Compiler XE 12.1 User and Reference Guides." http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/index.htm. 49

[26] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. 49

[27] J. Kurzak and J. Dongarra, "Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. lapack working note 178," tech. rep., University of Tennessee, 2006. 58

[28] P. Strazdins, "A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization," 1998. 58

[29] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience," *Concurrency and Computation: Practice and Experience*, vol. 15, p. 2003, 2003. 58

Vita

Jonathan Lawrence Peyton was born in Kirkland WA, on 13 October 1987. Moving to Tennessee one year later, he remained until he graduated from Oak Ridge High School, TN in 2006. From there he went on to the University of Tennessee in Knoxville, TN. He received his Bachelor of Science in Computer Engineering in May 2010. He continued his education at UTK where he is currently pursuing his Master of Science degree in Computer Engineering. His research has focused on optimizing fundamental linear algebra kernels for multiple Intel® Microarchitectures.