Chapter 1

Implementing Matrix Multiplication on the Cell B. E.

Wesley Alvaro

Department of Electrical Engineering and Computer Science, University of Tennessee

Jakub Kurzak

Department of Electrical Engineering and Computer Science, University of Tennessee

Jack Dongarra

Department of Electrical Engineering and Computer Science, University of Tennessee
Computer Science and Mathematics Division, Oak Ridge National Laboratory
School of Mathematics & School of Computer Science, Manchester University

1.1 Introduction

Dense matrix multiplication is one of the most common numerical operations, especially in the area of dense linear algebra, where it forms the core of many important algorithms, including solvers of linear systems of equations, least square problems, and singular and eigenvalue problems. The Cell B. E. excells in its capabilities to process compute-intensive workloads, like matrix multiplication, in single precision, through its powerful SIMD capabilities. This chapter disects implementations of two single precision matrix multiplication kernels for the SIMD cores of the Cell B. E. (the SPEs), one implementing the $C = C - A \times B^T$ operation and the other implementing the $C = C - A \times B$ operation, for fixed size matrices of 64×64 elements. The unique dual-issue architecture of the SPEs provides for a great balance of the floating-point operations and the memory and permutation operations, leading to the utilization of the floating-point pipeline in excess of 99 % in both cases.

1.1.1 Performance Considerations

State of the art numerical linear algebra software utilizes *block algorithms* in order to exploit the memory hierarchy of traditional cache-based systems [8, 9]. Public domain libraries such as LAPACK [3] and ScaLAPACK [5] are good examples. These implementations work on square or rectangular submatrices in their inner loops, where operations are encapsulated in calls to *Basic Linear Algebra Subroutines* (BLAS) [4], with emphasis on expressing the computation as Level 3 BLAS, *matrix-matrix* type, operations. Frequently, the call is made directly to the matrix multiplication routine _GEMM. At the same time, all the other Level 3 BLAS can be defined in terms of _GEMM as well as a small amount of Level 1 and Level 2 BLAS [17]. Any improvement to the _GEMM routine immediately benefits the entire algorithm, which makes the optimization of the _GEMM routine yet more important. As a result, a lot of effort has been invested in optimized BLAS by hardware vendors as well as academic institutions through projects such as ATLAS [1] and GotoBLAS [2].

1.1.2 Code Size Considerations

In the current implementation of the Cell B. E. architecture, the SPEs are equipped with local memories (Local Stores) of 256 KB. It is a common practice to use tiles of 64×64 elements for dense matrix operations in single precision [6, 11, 12, 18, 19]. Such tiles occupy a 16 KB buffer in the Local Store. Between six and eight buffers are necessary to efficiently implement even such a simple operation as matrix multiplication [6, 11, 12]. Also, more complex operations, such as matrix factorizations, commonly allocate eight buffers [18, 19], which consume 128 KB of Local Store. In general, it is reasonable to assume that half of the Local Store is devoted to application data buffers. At the same time, the program may rely on library frameworks like ALF [14] or MCF [23], and utilize numerical libraries such as SAL [20], SIMD Math [15], or MASS [7], which consume extra space for the code. In the development stage, it may also be desirable to generate execution traces for analysis with tools like TATL^{TM} [21] or Paraver [10], which require additional storage for event buffers. Finally, the Local Store also houses the SPE stack, starting at the highest LS address and growing towards lower addresses with no overflow protection. As a result, the Local Store is a scarce resource and any *real-world* application is facing the problem of fitting tightly coupled components together in the limited space.

1.2 Implementation

1.2.1 Loop Construction

The main tool in loop construction is the technique of loop unrolling [13]. In general, the purpose of loop unrolling is to avoid pipeline stalls by separating dependent instructions by a distance in clock cycles equal to the corresponding pipeline latencies. It also decreases the overhead associated with advancing the loop index and branching. On the SPE it serves the additional purpose of balancing the ratio of instructions in the odd and even pipeline, owing to register reuse between iterations.

In the canonical form, matrix multiplication $C_{m \times n} = A_{m \times k} \times B_{k \times n}$ consists of three nested loops iterating over the three dimensions m, n, and k. Loop tiling [22] is applied to improve the locality of reference and to take advantage of the $O(n^3)/O(n^2)$ ratio of arithmetic operations to memory accesses. This way register reuse is maximized and the number of loads and stores is minimized.

Conceptually, tiling of the three loops creates three more inner loops, which calculate a product of a submatrix of A and a submatrix of B and updates a submatrix of C with the partial result. Practically, the body of these three inner loops is subject to complete unrolling to a single block of a straight-line code. The tile size is picked such that the cross-over point between arithmetic and memory operations is reached, which means that there is more FMA or FNMS operations to fill the even pipeline than there is load, store, and shuffle operations to fill the odd pipeline.

The resulting structure consists of three outer loops iterating over tiles of A, B, and C. Inevitably, nested loops induce mispredicted branches, which can be alleviated by further unrolling. Aggressive unrolling, however, leads quickly to undesired code bloat. Instead, the three-dimensional problem can be linearized by replacing the loops with a single loop performing the same traversal of the iteration space. This is accomplished by traversing tiles of A, B, and C in a predefined order derived as a function of the loop index. A straightforward row/column ordering can be used and tile pointers for each iteration can be constructed by simple transformations of the bits of the loop index.

At this point, the loop body still contains *auxiliary* operations that cannot be overlapped with arithmetic operations. These include initial loads, stores of final results, necessary data rearrangement with splats (copy of one element across a vector) and shuffles (permutations of elements within a vector), and

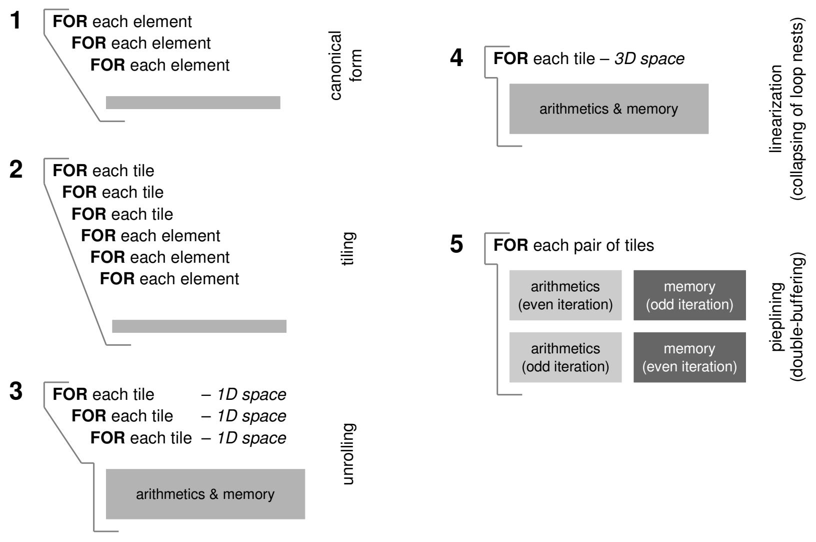

FIGURE 1.1: Basic steps of _GEMM loop optimization.

pointer advancing operations. This problem is addressed by *double-buffering*, on the register level, between two loop iterations. The existing loop body is duplicated and two separate blocks take care of the even and odd iteration, respectively. Auxiliary operations of the even iteration are hidden behind arithmetic instructions of the odd iteration and vice versa, and disjoint sets of registers are used where necessary. The resulting loop is preceded by a small body of *prologue* code loading data for the first iteration, and then followed by a small body of *epilogue* code, which stores results of the last iteration. Figure 1.1 shows the optimization steps leading to a high-performance implementation of the _GEMM inner kernel.

1.2.2 C = C − A × B trans

Before going into details, it should be noted that matrix storage follows C-style row-major format. It is not as much a careful design decision, as compliance with the common practice on the Cell B. E. It can be attributed to C compilers being the only ones allowed to exploit short-vector capabilities of the SPEs through C language SIMD extensions. If compliance with libraries relying on legacy FORTRAN API is required, a translation operation is necessary.

An easy way to picture the $C = C - A \times B^T$ operation is to represent it as the standard matrix vector product $C = C - A \times B$, where A is stored using

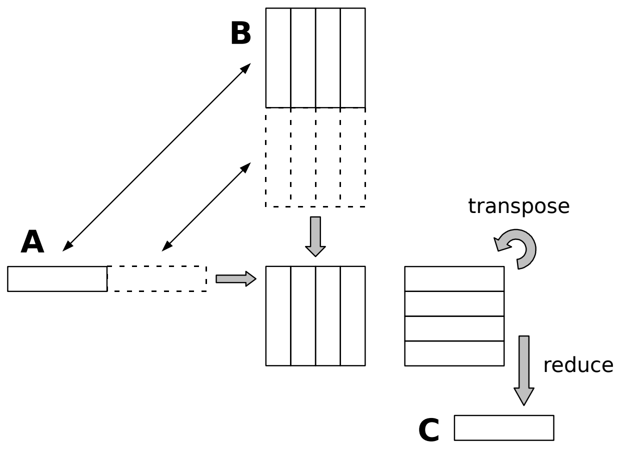

FIGURE 1.2: Basic operation of the $C = C - A \times B^T$ matrix multiplication micro-kernel.

row-major order and B is stored using column-major order. It can be observed that in this case a row of A can readily be multiplied with a column of B to yield a vector containing four partial results, which need to be summed up to produce one element of C. The vector reduction step introduces superfluous multiply-add operations. In order to minimize their number, four row-column products are computed, resulting in four vectors, which need to be internally reduced. The reduction is performed by first transposing the 4×4 element matrix represented by the four vectors and then applying four vector multiply-add operations to produce a result vector containing four elements of C. The basic scheme is depicted in Figure 1.2.

The crucial design choice to be made is the right amount of unrolling, which is equivalent to deciding the right tile size in terms of the triplet $\{m, n, k\}$ (here sizes express numbers of individual floating-point values, not vectors). Unrolling is mainly used to minimize the overhead of jumping and advancing the index variable and associated pointer arithmetic. However, both the jump and the jump hint instructions belong to the odd pipeline and, for compute intensive loops, can be completely hidden behind even pipeline instructions and thus introduce no overhead. In terms of the overhead of advancing the index variable and related pointer arithmetic, it will be shown in §1.2.4 that all of these operations can be placed in the odd pipeline as well. In this situation, the only concern is balancing even pipeline, arithmetic instructions with odd pipeline, data manipulation instructions.

Simple analysis can be done by looking at the number of floating-point operations versus the number of loads, stores, and shuffles, under the assumption that the size of the register file is not a constraint. The search space for the $\{m, n, k\}$ triplet is further truncated by the following criteria: only powers of two are considered in order to simplify the loop construction; the maximum possible number of 64 is chosen for k in order to minimize the number of extraneous floating-point instructions performing the reduction of partial results; only multiplies of four are selected for n to allow for efficient reduction of partial results with eight shuffles per one output vector of C. Under these constraints, the entire search space can be easily analyzed.

Table 1.1 shows how the number of each type of operation is calculated. Table 1.2 shows the number of even pipeline, floating-point instructions including the reductions of partial results. Table 1.3 shows the number of even pipeline instructions minus the number of odd pipeline instructions including loads, stores, and shuffles (not including jumps and pointer arithmetic). In other words, Table 1.3 shows the number of spare odd pipeline slots before

TABLE 1.1: Numbers of different types of operations in the computation of one tile of the $C = C - A \times B^T$ micro-kernel, as a function of tile size ({m, n, 64} triplet).

Type of Operation	Pipeline		Number of Operations
	Even	Odd	
Floating point	✘		(m × n × 64) ⁄ 4 + m × n
Load A		✘	m × 64 ⁄ 4
Load B		✘	64 × n ⁄ 4
Load C		✘	m × n ⁄ 4
Store C		✘	m × n ⁄ 4
Shuffle		✘	m × n ⁄ 4 × 8

TABLE 1.2: Number of even pipeline, floating-point operations in the computation of one tile of the micro-kernel $C = C - A \times B^T$, as a function of tile size ({m, n, 64} triplet).

M/N	4	8	16	32	64
1	68	136	272	544	1088
2	136	272	544	1088	2176
4	272	544	1088	2176	4352
8	544	1088	2176	4352	8704
16	1088	2176	4352	8704	17408
32	2176	4352	8704	17408	34816
64	4352	8704	17408	34816	69632

TABLE 1.3: Number of spare odd pipeline slots in the computation of one tile of the $C = C - A \times B^T$ micro-kernel, as a function of tile size ({m, n, 64} triplet).

M/N	4	8	16	32	64
1	-22	-28	-40	-64	-112
2	20	72	176	384	800
4	104	272	608	1280	2624
8	272	672	1472	3072	6272
16	608	1472	3200	6656	13568
32	1280	3072	6656	13824	28160
64	2624	6272	13568	28160	57344

TABLE 1.4: The size of code for the computation of one tile of the $C = C - A \times B^T$ micro-kernel, as a function of tile size ({m, n, 64} triplet).

M/N	4	8	16	32	64
1	1.2	1.2	2.3	4.5	8.9
2	1.0	1.8	3.6	7.0	13.9
4	1.7	3.2	6.1	12.0	23.8
8	3.2	5.9	11.3	22.0	43.5
16	6.1	11.3	21.5	42.0	83.0
32	12.0	22.0	42.0	82.0	162.0
64	23.8	43.5	83.0	162.0	320.0

jumps and pointer arithmetic are implemented. Finally, Table 1.4 shows the size of code involved in calculations for a single tile. It is important to note here that the double-buffered loop is twice the size.

It can be seen that the smallest unrolling with a positive number of spare odd pipeline slots is represented by the triplet $\{2, 4, 64\}$ and produces a loop with 136 floating-point operations. However, this unrolling results in only 20 spare slots, which would barely fit pointer arithmetic and jump operations. Another aspect is that the odd pipeline is also used for instruction fetch, and near complete filling of the odd pipeline may cause instruction depletion, which in rare situations can even result in an indefinite stall [16].

The next larger candidates are triplets $\{4, 4, 64\}$ and $\{2, 8, 64\}$, which produce loops with 272 floating-point operations, and 104 or 72 spare odd pipeline slots, respectively. The first one is an obvious choice, giving more room in the odd pipeline and smaller code. It turns out that the $\{4, 4, 64\}$ unrolling is actually the most optimal of all, in terms of the overall routine footprint, when the implementation of pointer arithmetic is taken into account, as further explained in §1.2.4.

It can be observed that the maximum performance of the routine is ultimately limited by the extra floating-point operations, which introduce an overhead not accounted for in the formula for operation count in matrix mul-

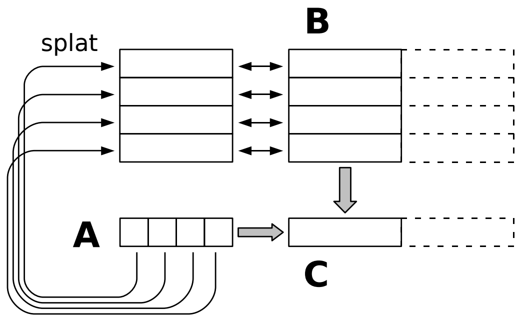

FIGURE 1.3: Basic operation of the $C = C - A \times B$ matrix multiplication micro-kernel.

tiplication: $2 \times m \times n \times k$. For matrices of size 64×64, every 64 multiply-add operations require four more operations to perform the intra-vector reduction. This sets a hard limit on the maximum achievable performance to $64/(64 + 4) \times 25.6 = 24.09$ $[Gflop/s]$, which is roughly 94 % of the peak.

1.2.3 $\mathbf{C = C - A \times B}$

Here, same as before, row-major storage is assumed. The key observation is that multiplication of one element of A with one row of B contributes to one row of C. As a result, the elementary operation splats an element of A over a vector, multiplies this vector with a vector of B, and accumulates the result in a vector of C (Figure 1.3). Unlike for the other kernel, in this case no extra floating-point operations are involved.

Same as before, the size of unrolling has to be decided in terms of the triplet $\{m, n, k\}$. This time, however, there is no reason to fix any dimension. Nevertheless, similar constraints to the search space apply: all dimensions have to be powers of two, and additionally only multiples of four are allowed for n and k to facilitate efficient vectorization and simple loop construction. Table 1.5 shows how the number of each type of operation is calculated. Table 1.6 shows the number of even pipeline, floating-point instructions. Table 1.7 shows the number of even pipeline instructions minus the number of odd pipeline instructions including loads, stores, and splats (not including jumps and pointer arithmetic). In other words, Table 1.7 shows the number of spare odd pipeline slots before jumps and pointer arithmetic are implemented. Finally, Table 1.8 shows the size of code involved in calculations for a single tile. It should be noted again that the double-buffered loop is twice the size.

It can be seen that the smallest unrolling with a positive number of spare odd pipeline slots produces a loop with 128 floating-point operations. Five

TABLE 1.5: Numbers of different types of operations in the computation of one tile of the $C = C - A \times B$ micro-kernel, as a function of tile size ({m, n, k}).

Type of Operation	Pipeline		Number of Operations
	Even	Odd	
Floating point	✘		(m × n × k) ⁄ 4
Load A		✘	m × k ⁄ 4
Load B		✘	k × n ⁄ 4
Load C		✘	m × n ⁄ 4
Store C		✘	m × n ⁄ 4
Splat		✘	m × k

TABLE 1.6: Number of even pipeline operations in the computation of one tile of the micro-kernel $C = C - A \times B$, as a function of tile size ({m, n, k}).

K	M/N	4	8	16	32	64
4	1	4	8	16	32	64
4	2	8	16	32	64	128
4	4	16	32	64	128	256
4	8	32	64	128	256	512
4	16	64	128	256	512	1024
4	32	128	256	512	1024	2048
4	64	256	512	1024	2048	4096
8	1	8	16	32	64	128
8	2	16	32	64	128	256
8	4	32	64	128	256	512
8	8	64	128	256	512	1024
8	16	128	256	512	1024	2048
8	32	256	512	1024	2048	4096
8	64	512	1024	2048	4096	8192
16	1	16	32	64	128	256
16	2	32	64	128	256	512
16	4	64	128	256	512	1024
16	8	128	256	512	1024	2048
16	16	256	512	1024	2048	4096
16	32	512	1024	2048	4096	8192
16	64	1024	2048	4096	8192	16384

TABLE 1.7: Number of spare odd pipeline slots in the computation of one tile of the $C = C - A \times B$ micro-kernel, as a function of tile size ({m, n, k}).

K	M/N	4	8	16	32	64
4	1	-7	-9	-13	-21	-37
4	2	-10	-10	-10	-10	-10
4	4	-16	-12	-4	12	44
4	8	-28	-16	8	56	152
4	16	-52	-24	32	144	368
4	32	-100	-40	80	320	800
4	64	-196	-72	176	672	1664
8	1	-12	-14	-18	-26	-42
8	2	-16	-12	-4	12	44
8	4	-24	-8	24	88	216
8	8	-40	0	80	240	560
8	16	-72	16	192	544	1248
4	32	-136	48	416	1152	2624
4	64	-264	112	864	2368	5376
16	1	-22	-24	-28	-36	-52
16	2	-28	-16	8	56	152
16	4	-40	0	80	240	560
16	8	-64	32	224	608	1376
16	16	-112	96	512	1344	3008
16	32	-208	224	1088	2816	6272
16	64	-400	480	2240	5760	12800

possibilities exist, with the triplet $\{4, 16, 8\}$ providing the highest number of 24 spare odd pipeline slots. Again, such unrolling would both barely fit pointer arithmetic and jump operations and be a likely cause of instruction depletion.

The next larger candidates are unrollings that produce loops with 256 floating-point operations. There are 10 such cases, with the triplet $\{4, 32, 8\}$ being the obvious choice for the highest number of 88 spare odd pipeline slots and the smallest code size. It also turns out that this unrolling is actually the most optimal in terms of the overall routine footprint, when the implementation of pointer arithmetic is taken into account, as further explained in §1.2.4.

Unlike for the other routine, the maximum performance is not limited by any extra floating-point operations, and performance close to the peak of 25.6 $Gflop/s$ should be expected.

1.2.4 Advancing Tile Pointers

The remaining issue is the one of implementing the arithmetic calculating the tile pointers for each loop iteration. Due to the size of the input matrices and the tile sizes being powers of two, this is a straightforward task. The tile offsets can be calculated from the tile index and the base addresses of the input matrices using integer arithmetic and bit manipulation instructions (bitwise logical instructions and shifts). Figure 1.4 shows a sample implementation of

```
int tile;

vector float *Abase;
vector float *Bbase;
vector float *Cbase;

vector float *Aoffs;
vector float *Boffs;
vector float *Coffs;

Aoffs = Abase + ((tile & ~0x0F) << 2);
Boffs = Bbase + ((tile &  0x0F) << 6);
Coffs = Cbase +  (tile &  0x0F)
              + ((tile & ~0x0F) << 2);
```

FIGURE 1.4: Sample C language implementation of pointer arithmetic for the kernel $C = C - A \times B^T$ with unrolling corresponding to the triplet $\{4, 4, 64\}$.

TABLE 1.8: The size of code for the computation of one tile of the $C = C - A \times B$ micro-kernel, as a function of tile size ({m, n, k}).

K	M/N	4	8	16	32	64
4	1	0.1	0.1	0.2	0.3	0.6
4	2	0.1	0.2	0.3	0.5	1.0
4	4	0.2	0.3	0.5	1.0	1.8
4	8	0.4	0.6	1.0	1.8	3.4
4	16	0.7	1.1	1.9	3.4	6.6
4	32	1.4	2.2	3.7	6.8	12.9
4	64	2.8	4.3	7.3	13.4	25.5
8	1	0.1	0.2	0.3	0.6	1.2
8	2	0.2	0.3	0.5	1.0	1.8
8	4	0.3	0.5	0.9	1.7	3.2
8	8	0.7	1.0	1.7	3.1	5.8
8	16	1.3	1.9	3.3	5.9	11.1
4	32	2.5	3.8	6.4	11.5	21.8
4	64	5.0	7.6	12.6	22.8	43.0
16	1	0.2	0.3	0.6	1.1	2.2
16	2	0.4	0.6	1.0	1.8	3.4
16	4	0.7	1.0	1.7	3.1	5.8
16	8	1.3	1.9	3.1	5.6	10.6
16	16	2.4	3.6	6.0	10.8	20.3
16	32	4.8	7.1	11.8	21.0	39.5
16	64	9.6	14.1	23.3	41.5	78.0

```
lqa  $2,tile
lqa  $3,Abase
andi $4,$2,-16
andi $2,$2,15
shli $6,$4,2
shli $4,$4,6
shli $5,$2,10
a    $2,$2,$6
a    $4,$4,$3
shli $2,$2,4
lqa  $3,Bbase
stqa $4,Aoffs
a    $5,$5,$3
lqa  $3,Cbase
stqa $5,Boffs
a    $2,$2,$3
stqa $2,Coffs
```

FIGURE 1.5: The result of compiling the code from Figure 1.4 to assembly language, with even pipeline instructions in bold.

pointer arithmetic for the kernel $C = C - A \times B^T$ with unrolling corresponding to the triplet $\{4, 4, 64\}$. *Abase*, *Bbase*, and *Cbase* are base addresses of the input matrices, and the variable *tile* is the tile index running from 0 to 255; *Aoffs*, *Boffs*, and *Coffs* are the calculated tile offsets.

Figure 1.5 shows the result of compiling the sample C code from Figure 1.4 to assembly code. Although a few variations are possible, the resulting assembly code will always involve a similar combined number of integer and bit manipulation operations. Unfortunately, all these instructions belong to the even pipeline and will introduce an overhead, which cannot be hidden behind floating point operations, like it is done with loads, stores, splats, and shuffles.

One way of minimizing this overhead is extensive unrolling, which creates a loop big enough to make the pointer arithmetic negligible. An alternative is to eliminate the pointer arithmetic operations from the even pipeline and replace them with odd pipeline operations. With the unrolling chosen in §1.2.2 and §1.2.3, the odd pipeline offers empty slots in abundance. It can be observed that, since the loop boundaries are fixed, all tile offsets can be calculated in advance. At the same time, the operations available in the odd pipeline include loads, which makes it a logical solution to precalculate and tabulate tile offsets for all iterations. It still remains necessary to combine the offsets with the base addresses, which are not known beforehand. However, under additional alignment constraints, offsets can be combined with bases using shuffle instructions, which are also available in the odd pipeline. As will be further shown, all instructions that are not floating point arithmetic can be removed from the even pipeline.

The precalculated offsets have to be compactly packed in order to preserve

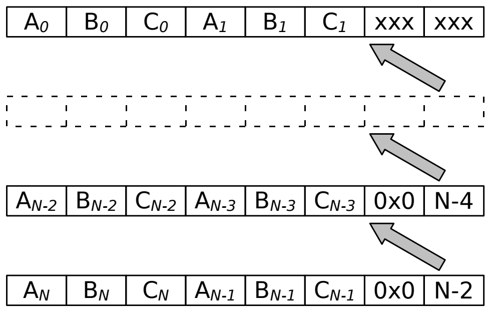
A_0 B_0 C_0 A_1 B_1 C_1 xxx xxx
A_{N-2} B_{N-2} C_{N-2} A_{N-3} B_{N-3} C_{N-3} 0x0 N-4
A_N B_N C_N A_{N-1} B_{N-1} C_{N-1} 0x0 N-2

FIGURE 1.6: Organization of the tile offset lookup table. N is the number of tiles.

space consumed by the lookup table. Since tiles are 16 KB in size, offsets consume 14 bits and can be stored in a 16-bit halfword. Three offsets are required for each loop iteration. With eight halfwords in a quadword, each quadword can store offsets for two loop iterations or a single interation of the pipelined, double-buffered loop. Figure 1.6 shows the organization of the offset lookup table.

The last arithmetic operation remaining is the advancement of the iteration variable. It is typical to decrement the iteration variable instead of incrementing it, and branch on non-zero, in order to eliminate the comparison operation, which is also the case here. This still leaves the decrement operation, which would have to occupy the even pipeline. In order to annihilate the decrement, each quadword containing six offsets for one iteration of the double-buffered loop also contains a seventh entry, which stores the index of the quadword to be processed next (preceding in memory). In other words, the iteration variable, which also serves as the index to the lookup table, is tabulated along with the offsets and loaded instead of being decremented.

Normally, the tile pointers would have to be calculated as a sum of an 18-bit base address and a 14-bit offset, which would require the use of integer addition residing in the even pipeline. With the additional constraint of 16 KB alignment of the base addresses, 14 less significant bits of the base are zero and can be simply replaced with the bits of the offset. The replacement could be implemented with the logical *AND* operation. This would however, again, involve an even pipeline instruction. Instead, both the base addresses and the offsets are initially shifted left by two bits, which puts the borderline between offsets and bases on a byte boundary. At this point the odd pipeline shuffle instruction operating at byte granularity can be used to combine the base with the offset. Finally, the result has to be shifted right by two bits, which can be accomplished by a combination of bit and byte quadword rotations,

TABLE 1.9: The overall footprint of the micro-kernel $C = C - A \times B^T$, including the code and the offset lookup table, as a function of tile size ({m, n, 64} triplet).

M/N	4	8	16	32	64
1	9.2	6.3	6.6	10.0	18.4
2	6.0	5.7	8.1	14.5	28.0
4	5.4	7.4	12.8	24.3	47.6
8	7.4	12.3	22.8	44.1	87.1
16	12.8	22.8	43.1	84.1	166.0
32	24.3	44.1	84.1	164.0	324.0
64	47.6	87.1	166.0	324.0	640.0

which also belong to the odd pipeline. Overall, all the operations involved in advancing the double-buffered loop consume 29 extra odd pipeline slots, which is small, given that 208 are available in the case of the first kernel and 176 in the case of the second.

This way, all operations involved in advancing from tile to tile are implemented in the odd pipeline. At the same time, both the branch instruction and the branch hint belong to the odd pipeline. Also, a correctly hinted branch does not cause any stall. As a result, such an implementation produces a continuous stream of floating-point operations in the even pipeline, without a single cycle devoted to any other activity.

The last issue to be discussed is the storage overhead of the lookup table. This size is proportional to the number of iterations of the unrolled loop and reciprocal to the size of the loop body. Using the presented scheme (Figure 1.6), the size of the lookup table in bytes equals $N^3/(m \times n \times k) \times 8$. Table 1.9 presents the overall footprint of the $C = C - A \times B^T$ micro-kernel as a function of the tile size. Table 1.10 presents the overall footprint of the $C = C - A \times B$ micro-kernel as a function of the tile size. As can be clearly seen, the chosen tile sizes result in the lowest possible storage requirements for the routines.

1.3 Results

Both presented SGEMM kernel implementations produce a continuous stream of floating-point instructions for the duration of the pipelined loop. In both cases, the loop iterates 128 times, processing two tiles in each iteration. The $C = C - A \times B^T$ kernel contains 544 floating-point operations in the loop body and, on a 3.2 GHz processor, delivers 25.54 Gflop/s (99.77 % of peak) if actual operations are counted, and 24.04 Gflop/s (93.90 % of peak) if the

TABLE 1.10: The overall footprint of the micro-kernel $C = C - A \times B$, including the code and the offset lookup table, as a function of tile size ({m, n, 64} triplet).

K	M/N	4	8	16	32	64
4	1	128.1	64.2	32.4	16.7	9.3
4	2	64.2	32.3	16.6	9.1	6.1
4	4	32.4	16.6	9.0	5.9	5.7
4	8	16.7	9.1	5.9	5.6	7.8
4	16	9.4	6.2	5.8	7.9	13.6
4	32	6.8	6.3	8.4	14.0	26.0
4	64	7.5	9.6	15.1	27.0	51.1
8	1	64.2	32.4	16.6	9.2	6.3
8	2	32.4	16.6	9.0	5.9	5.7
8	4	16.7	9.1	5.8	5.3	7.3
8	8	9.3	6.0	5.4	7.1	12.1
8	16	6.6	5.9	7.5	12.3	22.5
4	32	9.1	9.6	13.8	23.5	43.8
4	64	12.1	16.1	25.8	45.8	86.1
16	1	32.4	16.7	9.2	6.3	6.4
16	2	16.7	9.1	5.9	5.6	7.8
16	4	9.3	6.0	5.4	7.1	12.1
16	8	6.5	5.8	7.3	11.8	21.5
16	16	6.9	8.3	12.5	21.8	40.6
16	32	10.6	14.8	23.8	42.1	79.1

standard formula, $2N^3$, is used for operation count. The $C = C - A \times B$ kernel contains 512 floating-point operations in the loop body and delivers 25.55 Gflop/s (99.80 % of peak). Here, the actual operation count equals $2N^3$. At the same time, neither implementation overfills the odd pipeline, which is 31 % empty for the first case and 17 % empty for the second case. This guarantees no contention between loads and stores and DMA operations, and no danger of instruction fetch starvation. Table 1.11 shows the summary of the kernels' properties.

1.4 Summary

Computational micro-kernels are architecture specific codes, where no portability is sought. It has been shown that systematic analysis of the problem combined with exploitation of low-level features of the Synergistic Processing Unit of the Cell B. E. leads to dense matrix multiplication kernels achieving peak performance without code bloat.

This proves that great performance can be achieved on SIMD architecture by optimizing code manually. The question remains, whether similar results

TABLE 1.11: Summary of the properties of the SPE SIMD SGEMM micro-kernels.

Characteristic	C=C-A×B^T	C=C-A×B
Performance	24.04 Gflop/s	25.55 Gflop/s
Execution time	21.80 μs	20.52 μs
Fraction of peak *USING THE 2xMxNxK FORMULA*	93.90 %	99.80 %
Fraction of peak *USING ACTUAL NUMBER OF FLOATING–POINT INSTRUCTIONS*	99.77 %	99.80%
Dual issue rate *ODD PIPELINE WORKLOAD*	68.75 %	82.81 %
Register usage	69	69
Code segment size	4008	3992
Data segment size	2192	2048
Total memory footprint	6200	6040

can be accomplished by automatic vectorization techniques or a combination of auto-vectorization with heuristic techniques based on searching the parameter space. It is likely that good results could be achieved by a combination of the *Superworld Level Parallelism* technique for auto-vectorization [24] with heuristic search similar to the ATLAS [1] methodology.

1.5 Code

The code is freely available, under the BSD license and can be downloaded from the author's web site `http://icl.cs.utk.edu/~alvaro/`. A few comments can be useful here. In absence of better tools, the code has been developed with the help of a spreadsheet, mainly for easy manipulation of two columns of instructions for the two pipelines of the SPE. Other useful features were taken advantage of as well. Specifically, color coding of blocks of instructions greatly improves the readability of the code. It is the hope of the authors that such visual representation of code considerably helps the reader's understanding of the techniques involved in construction of optimized SIMD assembly code. Also, the authors put forth considerable effort in making the software self-contained, in the sense that all tools involved in construction of the code are distributed alongside. This includes the lookup table generation code and the scripts facilitating translation from spreadsheet format to SPE assembly language.

Bibliography

[1] ATLAS. `http://math-atlas.sourceforge.net/`.

[2] GotoBLAS. `http://www.tacc.utexas.edu/resources/software/`.

[3] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. `http://www.netlib.org/lapack/lug/`.

[4] Basic Linear Algebra Technical Forum. *Basic Linear Algebra Technical Forum Standard*, August 2001. `http://www.netlib.org/blas/blast-forum/blas-report.pdf`.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. `http://www.netlib.org/scalapack/slug/`.

[6] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation — A performance view. *IBM J. Res. & Dev.*, 51(5):559–572, 2007. DOI: 10.1147/rd.515.0559.

[7] IBM Corporation. Mathematical Acceleration Subsystem — Product overview. `http://www-306.ibm.com/software/awdtools/mass/`, March 2007.

[8] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713897.

[9] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, 1998. ISBN: 0898714281.

[10] European Center for Parallelism of Barcelona, Technical University of Catalonia. *Paraver, Parallel Program Visualization and Analysis Tool Reference Manual, Version 3.1*, October 2001.

[11] D. Hackenberg. Einsatz und Leistungsanalyse der Cell Broadband Engine. Institut für Technische Informatik, Fakultät Informatik, Technische Universität Dresden, February 2007. Großer Beleg.

[12] D. Hackenberg. Fast matrix multiplication on CELL systems. `http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/matmul/`, July 2007.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann, 2006.

[14] IBM Corporation. *ALF for Cell BE Programmer's Guide and API Reference*, November 2007.

[15] IBM Corporation. *SIMD Math Library API Reference Manual*, November 2007.

[16] IBM Corporation. *Preventing Synergistic Processor Element Indefinite Stalls Resulting from Instruction Depletion in the Cell Broadband Engine Processor for CMOS SOI 90 nm, Applications Note, Version 1.0*, November 2007.

[17] B. Kågström, P. Ling, and C. van Loan. GEMM-Based Level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[18] J. Kurzak, A. Buttari, and J. J. Dongarra. Solving systems of linear equation on the CELL processor using Cholesky factorization. *Trans. Parallel Distrib. Syst.*, 19(9):1175–1186, 2008. DOI: TPDS.2007.70813.

[19] J. Kurzak and J. J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the CELL processor. *Concurrency Computat.: Pract. Exper.*, 19(10):1371–1385, 2007. DOI: 10.1002/cpe.1164.

[20] Mercury Computer Systems, Inc. *Scientific Algorithm Library (SAL) Data Sheet*, 2006. `http://www.mc.com/uploadedfiles/SAL-ds.pdf`.

[21] Mercury Computer Systems, Inc. *Trace Analysis Tool and Library (TATL^{TM}) Data Sheet*, 2006. `http://www.mc.com/uploadedfiles/tatl-ds.pdf`.

[22] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[23] M. Pepe. *Multi-Core Framework (MCF), Version 0.4.4.* Mercury Computer Systems, October 2006.

[24] J. Shin, J. Chame, and M. W. Hall. Exploiting superword-level locality in multimedia extension architectures. *J. Instr. Level Parallel.*, 5:1–28, 2003.