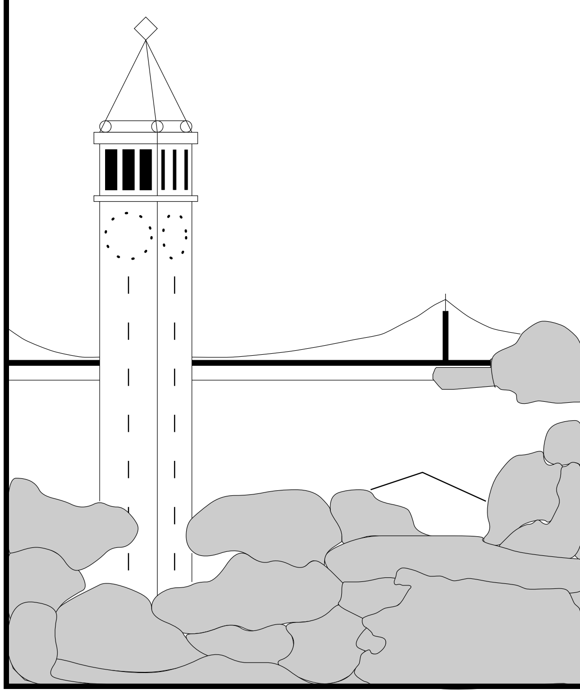
The PHiPAC v1.0 Matrix-Multiply Distribution.

Jeff Bilmes[1], *Krste Asanović*[2], *Chee-Whye Chin*[3], *Jim Demmel*[4]
{bilmes,krste,cheewhye,demmel}@cs.berkeley.edu

CS Division, University of California at Berkeley
Berkeley CA, 94720

International Computer Science Institute
Berkeley CA, 94704



Report No. UCB/CSD-98-1020

October 1998

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The PHiPAC v1.0 Matrix-Multiply Distribution.

Jeff Bilmes[¶], Krste Asanović[‖], Chee-Whye Chin[**], Jim Demmel[††]
{bilmes,krste,cheewhye,demmel}@cs.berkeley.edu

CS Division, University of California at Berkeley
Berkeley CA, 94720

International Computer Science Institute
Berkeley CA, 94704

October 1998

Abstract

Modern microprocessors can achieve high performance on linear algebra kernels but this currently requires extensive machine-specific hand tuning. We have developed a methodology whereby near-peak performance on a wide range of systems can be achieved automatically for such routines. First, by analyzing current machines and C compilers, we've developed guidelines for writing Portable, High-Performance, ANSI C (PHiPAC, pronounced "fee-pack"). Second, rather than code by hand, we produce parameterized code generators. Third, we write search scripts that find the best parameters for a given system. We report on a BLAS GEMM compatible multi-level cache-blocked matrix multiply generator which produces code that achieves around 90% of peak on the Sparcstation-20/61, IBM RS/6000-590, HP 712/80i, SGI Power Challenge R8k, and SGI Octane R10k, and over 80% of peak on the SGI Indigo R4k. In this paper, we provide a detailed description of the PHiPAC V1.0 matrix multiply distribution. We describe the code generator in detail including the various register and higher level blocking strategies. We also document the organization and parameters of the search scripts. This technical report is an expanded version of [BACD97].

1 Introduction

The use of a standard linear algebra library interface, such as BLAS [LHKK79, DCHH88, DCDH90], enables portable application code to obtain high-performance provided that an optimized library (e.g., [AGZ94, KHM94]) is available and affordable.

[*]CS Division, University of California at Berkeley and the International Computer Science Institute. The author acknowledges the support of JSEP contract F49620-94-C-0038.

[†]CS Division, University of California at Berkeley and the International Computer Science Institute. The author acknowledges the support of ONR URI Grant N00014-92-J-1617.

[‡]Department of Mathematics, Princeton University. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02).

[§]CS Division and Mathematics Dept., University of California at Berkeley. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02), ARPA contract DAAH04-95-1-0077 (University of Tennessee Subcontract ORA7453.02), DOE grant DE-FG03-94ER25219, DOE contract W-31-109-Eng-38, NSF grant ASC-9313958, and DOE grant DE-FG03-94ER25206.

[¶]CS Division, University of California at Berkeley and the International Computer Science Institute. The author acknowledges the support of JSEP contract F49620-94-C-0038.

[‖]CS Division, University of California at Berkeley and the International Computer Science Institute. The author acknowledges the support of ONR URI Grant N00014-92-J-1617.

[**]Department of Mathematics, Princeton University. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02).

[††]CS Division and Mathematics Dept., University of California at Berkeley. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02), ARPA contract DAAH04-95-1-0077 (University of Tennessee Subcontract ORA7453.02), DOE grant DE-FG03-94ER25219, DOE contract W-31-109-Eng-38, NSF grant ASC-9313958, and DOE grant DE-FG03-94ER25206.

Developing an optimized library, however, is a difficult and time-consuming task. Even excluding algorithmic variants such as Strassen's method [BLS91] for matrix multiplication, these routines have a large design space with many parameters such as blocking sizes, loop nesting permutations, loop unrolling depths, software pipelining strategies, register allocations, and instruction schedules. Furthermore, these parameters have complicated interactions with the increasingly sophisticated microarchitectures of new microprocessors.

Various strategies can be used to produce optimized routines for a given platform. For example, the routines could be manually written in assembly code, but fully exploring the design space might then be infeasible, and the resulting code might be unusable or sub-optimal on a different system.

Another commonly used method is to code in a high level language but with manual tuning to match the underlying architecture [AGZ94, KHM94]. While less tedious than coding in assembler, this approach still requires writing machine specific code which is not performance-portable across a range of systems.

Ideally, the routines would be written once in a high-level language and fed to an optimizing compiler for each machine. There is a large literature on relevant compiler techniques, many of which use matrix multiplication as a test case [WL91, LRW91, MS95, ACF95, CFH95, SMP+96][1]. While these compiler heuristics generate reasonably good code in general, they tend not to generate near-peak code for any one operation. Also, a high-level language's semantics might obstruct aggressive compiler optimizations. Moreover, it takes significant time and investment before compiler research appears in production compilers, so these capabilities are often unavailable. While both microarchitectures and compilers will improve over time, we expect it will be many years before a single version of a library routine can be compiled to give near-peak performance across a wide range of machines.

We have developed a methodology, named PHiPAC [BAD+96, BACD97], for developing Portable High-Performance linear algebra libraries in ANSI C. Our goal is to produce, with minimal effort, high-performance linear algebra libraries for a wide range of systems. The PHiPAC methodology has three components. First, we have developed a generic model of current C compilers and microprocessors that provides guidelines for producing portable high-performance ANSI C code. Second, rather than hand code particular routines, we write parameterized generators [ACF95, MS95] that produce code according to our guidelines. Third, we write scripts that automatically tune code for a particular system by varying the generators' parameters and benchmarking the resulting routines.

We have found that writing a parameterized generator and search scripts for a routine takes less effort than hand-tuning a single version for a single system. Furthermore, with the PHiPAC approach, development effort can be amortized over a large number of platforms. And by automatically searching a large design space, we can discover winning yet unanticipated parameter combinations.

Using the PHiPAC methodology, we have produced a portable, BLAS-compatible matrix multiply generator. The resulting code can achieve over 90% of peak performance on a variety of current workstations, and is sometimes faster than the vendor-optimized libraries. We focus on matrix multiplication in this paper, but we have produced other generators including dot-product, AXPY, and convolution, which have similarly demonstrated portable high performance.

We concentrate on producing high quality uniprocessor libraries for microprocessor-based systems because multiprocessor libraries, such as [CDD+96], can be readily built from uniprocessor libraries. For vector and other architectures, however, our machine model would likely need substantial modification.

Section 2 describes our generic C compiler and microprocessor model, and develops the resulting guidelines for writing portable high-performance C code. Section 3 describes our C-code generator and the resulting code variants for a BLAS-compatible matrix multiply. Section 4 describes our strategy for searching the matrix multiply parameter space and the structure of the resulting GEMM library. This section also provides a detailed description of the various options that can be used to control the search. Section 5 presents performance results on several architectures comparing against vendor-supplied BLAS GEMM. Section 6 describes the availability of the distribution, and discusses future work. This technical report is an expanded version of [BACD97].

2 PHiPAC

By analyzing the microarchitectures of a range of machines, such as workstations and microprocessor-based SMP and MPP nodes, and the output of their ANSI C compilers, we derived a set of guidelines that help us attain high performance across a range of machine and compiler combinations [BAD+96].

[1] A longer list appears in [Wol96].

From our analysis of various ANSI C compilers, we determined we could usually rely on reasonable register allocation, instruction selection, and instruction scheduling. More sophisticated compiler optimizations, however, including pointer alias disambiguation, register and cache blocking, loop unrolling, and software pipelining, were either not performed or not very effective at producing the highest quality code.

Although it would be possible to use another target language, we chose ANSI C because it provides a low-level, yet portable, interface to machine resources, and compilers are widely available. One problem with our use of C is that we must explicitly work around pointer aliasing as described below. In practice, this has not limited our ability to extract near-peak performance.

We emphasize that for both microarchitectures and compilers we are determining a *lowest common denominator*. Some microarchitectures or compilers will have superior characteristics in certain attributes, but, if we code assuming these exist, performance will suffer on systems where they do not. Conversely, coding for the lowest common denominator should not adversely affect performance on more capable platforms. For example, some machines can fold a pointer update into a load instruction while others require a separate add. Coding for the lowest common denominator dictates replacing pointer updates with base plus constant offset addressing where possible. In addition, while some production compilers have sophisticated loop unrolling and software pipelining algorithms, many do not. Our search strategy (Section 4) empirically evaluates several levels of explicit loop unrolling and depths of software pipelining. While a naive compiler might benefit from code with explicit loop unrolling or software pipelining, a more sophisticated compiler might perform better without either.

2.1 PHiPAC Coding Guidelines

The following paragraphs exemplify PHiPAC C code generation guidelines. Programmers can use these coding guidelines directly to improve performance in critical routines while retaining portability, but this does come at the cost of less maintainable code. This problem is mitigated in the PHiPAC approach, however, by the use of parameterized code generators.

Using local variables, reorder operations to explicitly remove false dependencies.

Casually written C code often over-specifies operation order, particularly where pointer aliasing is possible. C compilers, constrained by C semantics, must obey these over-specifications thereby reducing optimization potential. We therefore remove these extraneous dependencies.

For example, the following code fragment contains a false Read-After-Write hazard:

```
a[i] = b[i] + c;
a[i+1] = b[i+1]*d;
```

The compiler may not assume `&a[i] != &b[i+1]` and is forced to first store `a[i]` to memory before loading `b[i+1]`. We may re-write this with explicit loads to local variables:

```
float f1,f2;
f1 = b[i]; f2 = b[i+1];
a[i] = f1 + c; a[i+1] = f2*d;
```

The compiler can now interleave execution of both original statements thereby increasing parallelism.

Exploit multiple integer and floating-point registers.

We explicitly keep values in local variables to reduce memory bandwidth demands. For example, consider the following 3-point FIR filter code:

```
while (...) {
   *res++ = filter[0]*signal[0] +
            filter[1]*signal[1] +
            filter[2]*signal[2];
   signal++;
}
```

The compiler will usually reload the filter values every loop iteration because of potential aliasing with `res`. We can remove the alias by preloading the filter into local variables that may be mapped into registers:

```
float f0,f1,f2;
f0=filter[0];
f1=filter[1];
f2=filter[2];
while ( ... ) {
  *res++ = f0*signal[0]
         + f1*signal[1] + f2*signal[2];
  signal++;
}
```

Minimize pointer updates by striding with constant offsets.

We replace pointer updates for strided memory addressing with constant array offsets. For example:

```
f0 = *r8; r8 += 4;
f1 = *r8; r8 += 4;
f2 = *r8; r8 += 4;
```

should be converted to:

```
f0 = r8[0];
f1 = r8[4];
f2 = r8[8];
r8 += 12;
```

Compilers can fold the constant index into a register plus offset addressing mode.

Hide multiple instruction FPU latency with independent operations.

We use local variables to expose independent operations so they can be executed independently in a pipelined or superscalar processor. For example:

```
f1=f5*f9;
f2=f6+f10;
f3=f7*f11;
f4=f8+f12;
```

Balance the instruction mix.

A balanced instruction mix has a floating-point multiply, a floating-point add, and 1–2 floating-point loads or stores interleaved. It is not worth decreasing the number of multiplies at the expense of additions if the total floating-point operation count increases.

Increase locality to improve cache performance.

Cached machines benefit from increases in spatial and temporal locality. Whenever possible, we arrange our code to have predominantly unit-stride memory accesses and try to reuse data once it is in cache. See Section 3.1 for our blocked matrix multiply example.

Convert integer multiplies to adds.

Integer multiplies and divides are slow relative to integer addition. Therefore, we use pointer updates rather than subscript expressions. For example, rather than:

```
for (i=...)
  { row_ptr = &p[i*col_stride]; ... }
```

we would produce:

```
for (i=...)
  { ... row_ptr += col_stride; }
```

Minimize branches, avoid magnitude compares.

Branches are costly, especially on modern superscalar processors. Therefore, we unroll loops to amortize branch cost and use C `do {} while ();` loop control whenever possible to avoid any unnecessary compiler-produced loop head branches.

Also, on many microarchitectures, it is cheaper to perform equality or inequality loop termination tests than magnitude comparisons. For example, instead of:

```
for (i=0,a=start_ptr;i<ARRAY_SIZE;i++,a++}
    { ... }
```

we produce:

```
end_ptr = &a[ARRAY_SIZE]; a = start_ptr;
do {
   ... a++;
} while (a != end_ptr);
```

This also removes one loop control variable.

Explicitly unroll loops to expose optimization opportunities.

We unroll loops explicitly to increase opportunities for other performance optimizations. For example, our 3-point FIR filter example above may be further optimized as follows:

```
float f0,f1,f2,s0,s1,s2;
f0=filter[0];
f1=filter[1];
f2=filter[2];
s0=signal[0];
s1=signal[1];
s2=signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
do {
   signal += 3;
   s0 = signal[0];
   res[0] = f0*s1 + f1*s2 + f2*s0;
   s1 = signal[1];
   res[1] = f0*s2 + f1*s0 + f2*s1;
   s2 = signal[2];
   res[2] = f0*s0 + f1*s1 + f2*s2;
   res += 3;
} while ( ... );
```

In the inner loop, there are now only two memory access per five floating point operations whereas in our unoptimized code, there were seven memory accesses per five floating point operations.

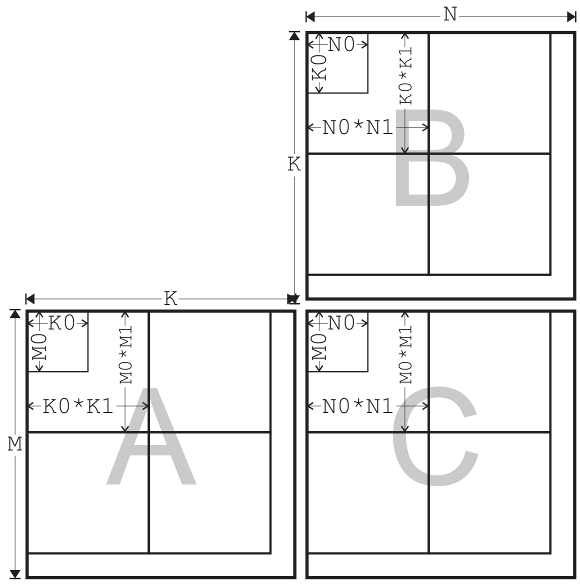


Figure 1: Definition of the matrix blocking parameters. M_0, K_0, and N_0 are the register blocking parameters – an $M_0 \times K_0$ block of A and a $K_0 \times N_0$ block of B is multiplied and accumulated into an $M_0 \times N_0$ block of C. M_1, K_1, and N_1 are the register blocking parameters – an $M_1 M_0 \times K_1 K_0$ block of A and a $K_1 K_0 \times N_1 N_0$ block of B is multiplied and accumulated into an $M_1 M_0 \times N_1 N_0$ block of C. Higher level cache blocking parameters (such as M_2, K_2, and N_2) are defined similarly.

3 Matrix Multiply C-Code Generators

`mm_cgen` and `mm_lgen` are generators that produce C code following the PHiPAC coding guidelines for one variant of the matrix multiply operation $C = \alpha op(A)op(B) + \beta C$ where $op(A)$, $op(B)$, and C, are respectively M×K, K×N, and M×N matrices, α and β are scalar parameters, and $op(X)$ is either $transpose(X)$ or just X. Our individual procedures have a lower level interface then a BLAS GEMM and have no error checking. For optimal efficiency, error checking should be performed by the caller when necessary rather than unnecessarily by the callee on every invocation. We create a full BLAS-compatible GEMM by generating all required matrix multiply variants and linking with our GEMM-compatible interface that includes error checking.

The code generators can produce a cache-blocked matrix multiply [GL89, LRW91, MS95], restructuring the algorithm for unit stride, and reducing the number of cache misses and unnecessary loads and stores. Under control of command line parameters, the generators can produce blocking code for any number of levels of memory hierarchy, including register, L1 cache, TLB, L2 cache, and so on. The generators can produce code using different accumulator precisions, and can also generate various flavors of software pipelining.

`mm_cgen` produces "core" code, that is code that blocks for the machine registers. `mm_lgen`, on the other hand, produces "level" code, or code that blocks for some higher level of the cache hierarchy. Typically, code produced by `mm_lgen` calls the routines produced by `mm_cgen`.

A typical invocation of a code generator is:

```
mm_cgen -l0 M0 K0 N0 [ -l1 M1 K1 N1 ] ...
```

where the register blocking is M_0, K_0, N_0, the L1-cache blocking is M_1, K_1, N_1, etc. The parameters M_0, K_0, and N_0 are specified in units of matrix elements, i.e., single, double, or extended precision floating-point numbers, M_1, K_1, N_1 are specified in units of register blocks, M_2, K_2, and K_2 are in units of L1 cache blocks, and so on. For a particular cache level, say i, the code accumulates into a C destination block of size $M_i \times N_i$ units and uses A source blocks of size $M_i \times K_i$ units and B source blocks of size $K_i \times N_i$ units (see Figure 1).

The next few sections describe the code generators and their resulting code in detail.

3.1 Matrix Multiply Core Code

mm_cgen is our core code generator. The generator produces code simliar to a six nested loop matrix multiply but where the three inner loops are completely unrolled. The unrolled code does a matrix-mutiply with A and B source matrices of size $M_0 \times K_0$ and $K_0 \times N_0$ respectively accumulating into a sized $M_0 \times N_0$ C matrix. The outer three loops perform an i-j-k-order blocked matrix multiply with block sizes determined either by matrix size arguments or by the register blocking parameters M_1, K_1, and N_1.

The generator can also optionally generate code for "fringes", i.e., portions of the matrix not multiples of M_0, K_0, and N_0 that are what we respectively call the m-fringe, k-fringe, and n-fringe. The fringe code is described below.

mm_cgen supports the following command line options:

```
Usage: mm_cgen [OPTIONS]
where [OPTIONS] include:
Semantics options:
  -opA [N|T]                 : A matrix op. Normal|Transpose
  -opB [N|T]                 : B matrix op. Normal|Transpose
  -no_m_fringe               : don't generate an M reg block fringe
  -no_k_fringe               : don't generate a K reg block fringe
  -no_n_fringe               : don't generate an N reg block fringe
  -no_fringes                : don't generate an M,K, or N reg block fringes
  -no_loop_head_branch       : Use do-while loops where possible, requiring M>=M0,K>=K0,N>=K0.
  -alpha [<val>|c]           : fix alpha at value <val> or arbitrary
  -beta [<val>|c]            : fix beta at value <val> or arbitrary
Optimization options:
  -l0 M0 K0 N0               : register (L0) blocking parameters
  -l1 M1 K1 N1               : L1 blocking parameters
  -sp [1|2lm|2ma|3]          : software pipelining options
  -holdstripe [A|B|m]        : hold regs. stripe A or B (default), or min(M0,N0) (-sp 1 only)
  -lin_m_fringe               : generate linear rather than log m-fringe code stripes
  -lin_k_fringe               : generate linear rather than log k-fringe code stripes
  -lin_n_fringe               : generate linear rather than log n-fringe code stripes
  -lin_fringe                 : generate linear rather than log fringe code stripes
Precision options:
  -prec [single|double|ldouble]  : Precision
  -sprec [single|double|ldouble] : Source Precision
  -aprec [single|double|ldouble] : Accumulator Precision
  -dprec [single|double|ldouble] : Destination Precision
Code generation & Misc. options:
  -help                      : Print this message
  -file name                 : Write to file 'name'
  -routine_name name         : Name of routines
  -spacechar c               : char to use as space
  -numspaces i               : spaces per nest
  -version                   : print version and exit.
```

The semantic options determine the set of matrices that may be multipled (correctly) by the resulting routine.

- `-opA [N|T]` controls if the A-matrix source operand should be treated as A or A^T.
- `-opB [N|T]` controls if the B-matrix source operand should be treated as B or B^T.
- `-no_m_fringe` means do not produce code for the M-fringe (so the resulting code is valid only for matrices whose M-dimension is a multiple of M_0.)
- `-no_k_fringe` means do not produce code for the K-fringe (so K must be a multiple of K_0 plus a constant depending on the software pipelining option, see below).

- `-no_n_fringe` means do not produce code for the N-fringe (so N must be a multiple of N_0.)
- `-no_fringe` means do not produce any fringe code at all.
- `-no_loop_head_branch` use C `do {} while` constructs rather than `while {}` in the three outer loops. This therefore causes the code to assume that M, K, and N are positive.
- `-alpha [<val>|c]` either hard-code α to be a particular value `val` or allow it to be an arbitrary value passed in as an argument. This is a semantic option but it can also effect performance, especially for $\alpha = \pm 1$ since, in this case, the generator produces code without the extra α multiplies.
- `-beta [<val>|c]` the same for β.

The optimization options change the code primarily in how it effects performance. As seen below, however, these options can also effect the resulting routine's behavior.

- `-l0 M0 K0 N0` controls the register blocking parameters.
- `-l1 M1 K1 N1` if given, produces code for a fixed matrix multiply where $M = M_0 \times M_1$, $K = K_0 \times K_1 + \ell$, and $N = N_0 \times N_1$, where ℓ is either 0, 1, or 2 depending on the software pipelining option being set respectively to either `-sp 1`, `-sp [2lm|2ma]`, or `-sp 3`.
- `-sp [1|2lm|2ma|3]` controls core-code sofware pipelining. `1` means no software pipelining, `2lm` and `2ma` means a two stage pipe, and `3` means a three stage pipe. Sofware pipelining will be described in detail below.
- `-holdstripe [A|B|m]` effects the code generated only when `-sp 1` is active. This will also be described below.
- `-lin_m_fringe` produces code for linearly spaced rather than logarithmically spaced fringe strips in the M dimension.
- `-lin_k_fringe` produces code for linearly spaced rather than logarithmically spaced fringe strips in the K dimension.
- `-lin_n_fringe` produces code for linearly spaced rather than logarithmically spaced fringe strips in the N dimension.
- `-lin_fringe` produces code for linearly spaced rather than logarithmically spaced fringe strips in all dimensions.

Note that for any set of options, the generated code will have comments stating for which matrices that particular routine will be valid. For example, the command

```
mm_cgen -l0 4 4 4 -sp 2ma -no_k_fringe -no_m_fringe
```

will produce the comment

```
General (M,K,N) = (m*4:m>=0, k*4+1:k>=1, N) matrix multiply
```

stating that this routine is valid for N non-negative, for M a non-negative multiple of 4, and for K one greater than a positive multiple of 4.

The precision options change the precision of the various operands and/or of the local variables used as temporaries internal to the routine. The other options are obvious from their command-line help description.

3.1.1 Simple Code example

In this section, we examine the code produced by `mm_cgen` for the operation `C = C + A*B` where A (respectively B, C) is an M×K (respectively K×N, M×N) matrix. Figure 2 lists the L1 cache blocking core code comprising the 3 nested loops, M, N, and K. This code was produced with the command:

```
mm_cgen -l0 2 2 2 -no_fringes -no_loop_head_branch
```

Because of the `no_loop_head_branch` parameter, this routine is valid only for matrices with $(M, K, N) = (2m : m >= 1, 2k : k >= 1, 2n : n >= 1)$.

The outer M loop in Figure 2 maintains pointers `c0` and `a0` to rows of register blocks in the A and C matrices. It also maintains end pointers (`ap0_endp` and `cp0_endp`) used for loop termination. The middle N loop maintains a pointer `b0` to columns of register blocks in the B matrix, and maintains a pointer `cp0` to the current C destination register block. The N loop also maintains separate pointers (`ap0_0` through `ap0_1`) to successive rows of the current A source block. It also initializes a pointer `bp0` to the current B source block. We assume local variables can be held in registers, so our code uses many pointers to minimize both memory references and integer multiplies.

The K loop iterates over source matrix blocks and accumulates into the same $M_0 \times N_0$ destination block. We assume that the floating-point registers can hold a $M_0 \times N_0$ accumulator block, so this block is loaded into local variables once before the K loop begins and stored after it ends. The K loop updates the set of pointers to the A source block, one of which is used for loop termination.

Currently, `mm_cgen` does not vary the loop permutation [MS95, LRW91] because the resulting gains in locality are subsumed by the method described below, at least for non-outer-product shaped matrices.

The parameter K_0 controls the extent of inner loop unrolling as can be seen in Figure 2. The unrolled core loop performs K_0 outer products accumulating into the C destination block. We code the outer products by loading one row of the B block, one element of the A block, then performing N_0 multiply-accumulates. The C code uses $N_0 + M_0$ memory references per $2N_0M_0$ floating-point operations in the inner K loop, while holding $M_0N_0 + N_0 + 1$ values in local variables. While the intent is that these local variables map to registers, the compiler is free to reorder all of the independent loads and multiply-accumulates to trade increased memory references for reduced register usage. The compiler also requires additional registers to name intermediate results propagating through machine pipelines.

The code we have so far described is valid only when M, K, and N are integer multiples of M_0, K_0, and N_0 respectively. In the general case, `mm_cgen` also includes code that operates on power-of-two sized fringe strips, i.e., 2^0 through $2^{\lfloor \log_2 L \rfloor}$ where L is M_0, K_0, or N_0. We can therefore manage any fringe size from 1 to L$-$1 by executing an appropriate combination of fringe code. The resulting code size growth is logarithmic in the register blocking (i.e., $O(\log(M_0)\log(K_0)\log(N_0))$) yet maintains good performance. `mm_cgen` also has the option to produce linear sized fringe strips (i.e., it will produce separate code for each possible fringe size) and is controlled in each dimension individually (see the `-lin_m_fringe`, `-lin_k_fringe`, `-lin_n_fringe`, and `-lin_fringe` options). This can be advantageous if the matrix workload has many matrices with a small M, K, or N (i.e., they are less than the corresponding L0 blocking numbers). It can therefore also be advantageous for use in LU decomposition.

To reduce the demands on the instruction cache, we arrange the code into several independent sections, the first handling the matrix core and the remainder handling the fringes. The code is structured not dissimilar to that shown in Figure 8 except that the fringes are managed, as described above, by power-of-2 fringe strips.

3.1.2 Core Code Options/Software Pipelining

As mentioned in the previous section, the fully unrolled core code consists of a series of K_0 outer products where each outer product uses a column vector from the A matrix and a row vector from the B matrix as operands and accumulates into a $M_0 \times N_0$ block of the C matrix (see Figure 3). The code always uses local variables sufficient to hold the entire $M_0 \times N_0$ block. It also declares additional local variables depending on the the `-sp` optimization option. Assuming that each local variable maps to a machine register, we structure the core code to achieve $2N_0M_0$ floating-point operations per $N_0 + M_0$ memory operations.

With no software pipelining `-sp 1`, there are two core code generation options. `mm_cgen` can either: 1) generate code to hold a column vector strip of the A matrix and a single element of the B matrix while accumulating into each column of the C block resulting in the additional use of $M_0 + 1$ local variables or can 2) generate code to hold a row vector strip of the B matrix and a single element of the A matrix while accumulating into each row of the C block resulting in the additional use of $N_0 + 1$ local variables. This choice is determined by the `-holdstripe` option.

```
mul_mfmf_mf(const int M,const int K,const int N,
const float*const A,const float*const B,float*const C,
const int Astride,const int Bstride,const int Cstride)
{
   const float *a,*b; float *c;
   const float *ap_0,*ap_1; const float *bp; float *cp;
   const int A_sbs_stride = Astride*2;
   const int C_sbs_stride = Cstride*2;
   const int k_marg_el = K & 1;
   const int k_norm = K - k_marg_el;
   const int m_marg_el = M & 1;
   const int m_norm = M - m_marg_el;
   const int n_marg_el = N & 1;
   const int n_norm = N - n_marg_el;
   float *const c_endp = C+m_norm*Cstride;
   register float c0_0,c0_1,c1_0,c1_1;
   c=C;a=A;
   do { /* M loop */
      const float* const ap_endp = a + k_norm;
      float* const cp_endp = c + n_norm;
      const float* const apc_1 = a + Astride;
      b=B;cp=c;
      do {  /* N loop */
         register float _b0,_b1;
         register float _a0,_a1;
         float *_cp;
         ap_0=a;ap_1=apc_1;bp=b;
         _cp=cp;c0_0=_cp[0];c0_1=_cp[1];
         _cp+=Cstride;c1_0=_cp[0];c1_1=_cp[1];
         do { /* K loop */
            _b0 = bp[0]; _b1 = bp[1];
            bp += Bstride; _a0 = ap_0[0];
            c0_0 += _a0*_b0; c0_1 += _a0*_b1;
            _a1 = ap_1[0];
            c1_0 += _a1*_b0; c1_1 += _a1*_b1;

            _b0 = bp[0]; _b1 = bp[1];
            bp += Bstride; _a0 = ap_0[1];
            c0_0 += _a0*_b0; c0_1 += _a0*_b1;
            _a1 = ap_1[1];
            c1_0 += _a1*_b0; c1_1 += _a1*_b1;

            ap_0+=2;ap_1+=2;
         } while (ap_0!=ap_endp);
         _cp=cp;_cp[0]=c0_0;_cp[1]=c0_1;
         _cp+=Cstride;_cp[0]=c1_0;_cp[1]=c1_1;
         b+=2;cp+=2;
      } while (cp!=cp_endp);
      c+=C_sbs_stride;a+=A_sbs_stride;
   } while (c!=c_endp);
}
```

Figure 2: $M_0 = 2$, $K_0 = 2$, $N_0 = 2$ matrix multiply L1 routine for $M \in \{2m : m \geq 1\}, K \in \{2k : k \geq 1\}, N \in \{2n : n \geq 1\}$. Within the K-loop is our fully-unrolled $2 \times 2 \times 2$ core matrix multiply. The code is not unlike the register code in [CFH95]. In our terminology, the leading dimensions LDA, LDB, and LDC are called `Astride`, `Bstride`, and `Cstride` respectively. The four local variables `c0_0` through `c1_1` hold a complete C destination block. Variables `ap_0` and `ap_1` point to successive rows of the A source matrix block, and variable `bp` points to the first row of the B source matrix block. Elements in A and B are accessed using constant offsets from the appropriate pointers.

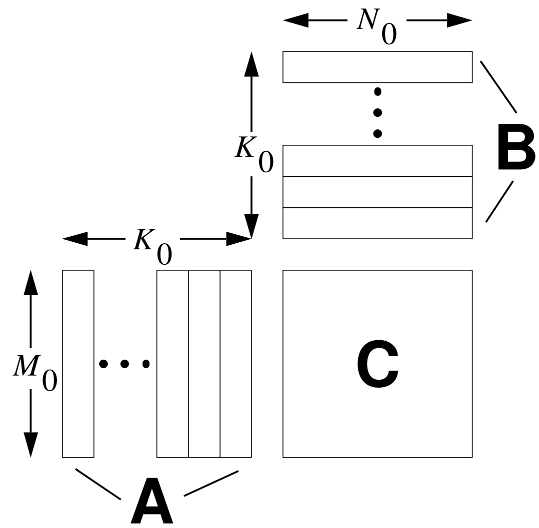


Figure 3: The three inner loops are fully unrolled to perform K_0 outer products accumulating into the destination matrix. If no software pipelining has been selected, the multiply-accumulates for each outer product are performed after either one of the following: 1) for each column of C, a length M_0 column of A and an element of B is loaded into local variables (the `-holdstrip A` option), or 2) for each row of C, a length N_0 row of B and an element of A is loaded into local variables (the `-holdstrip B` option).

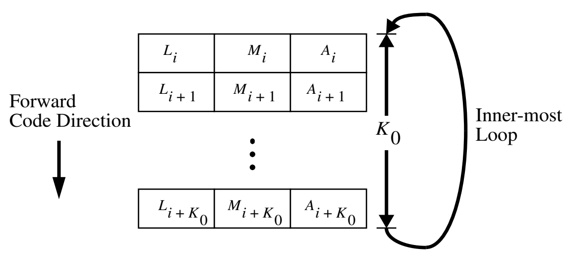


Figure 4: With no software pipelining, the loads, multiples, and accumulates for the K_0 outer product are placed within the inner-most loop. This leads to a startup cost due to the multiply and add units waiting for the loads to come in, and an "endup" cost when the load/store unit has nothing to do while the multiply and add units finish up.

`-holdstrip m` automatically choose either `-holdstripe A` or `-holdstripe B` depending on the minimum of M_0 and N_0 thereby minimizing the number of local variables used.

The compiler is free to re-order the loads, multiples, and adds as it chooses (while respecting data-dependences). Because of this fact, the `-holdstripe` option should *theoretically* have little or no effect on compiler optimization. In practice, however, we have indeed found non-negligible performance gains by varying this option.

Software pipelining is achieved by observing that each outer product consists of three sets of operations: 1) the loads of the source operands, 2) the multiplies of the source operands, and 3) the accumulates into the destination matrix. With no software pipelining, the code is structured as listed in Figure 4. A potential performance hit, therefore, can occur at the beginning of each loop body where the multiply and add units can sit idle while the operands (via the loads) become available. Similarly, at the end of the loop body, the load/store unit can sit idle while the multiplies and adds complete. While this problem is mitigated by increasing K_0, we ideally want to keep the load/store unit, the adder, and the multiplier occupied as often as possible and we would prefer not to require a huge K_0 since that reduces the set of possible L1 blocks (see Section 3.2).

We solve this problem by grouping together the loads, multiplies, and adds from different loop iterations while placing the starting and ending delays respectively before and after the core loop body. `mm_cgen` can produce code according to one of three styles of software pipelining. The `-sp 3` option uses a 3-stage pipe (Figure 5) where the

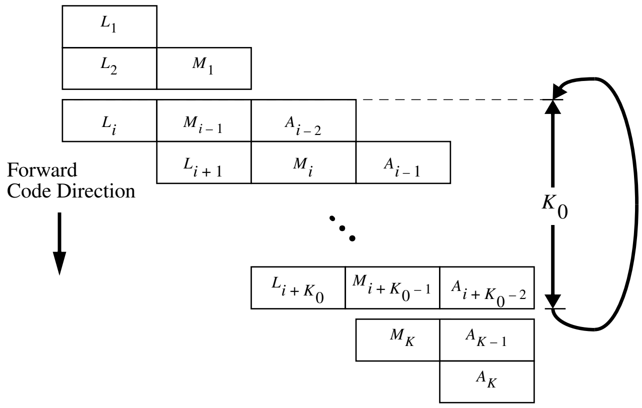


Figure 5: Three stage software pipelinging. The loads for the first two loop iterations and the multiplies for the first iteration are factored out of the unrolled inner-most loop as are the multiplies for the penultimate and the accumulates for the last two loop iterations.

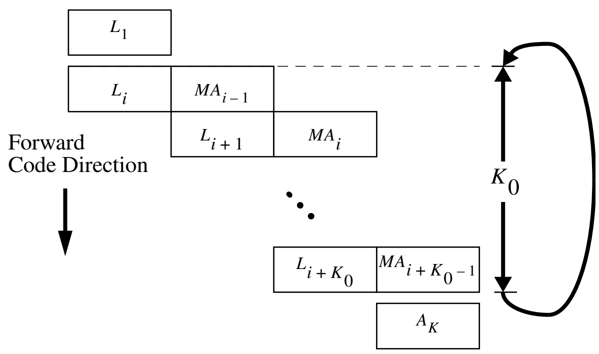


Figure 6: Two stage software pipelining.

loads, multiplies, and adds of an outer product are placed in three separate pipe stages. The core code semantics slightly change, however, as the code now operates on blocks of size $kK_0 + 2$ for non-negative integers k. One additional $M_0 \times N_0$ set of local variables is declared for use as temporaries as data flows down the "pipe". The actual code order is to first perform the accumulates of the $i - 2^{nd}$ iteration, the multiplies of the $i - 1^{th}$ iteration, and then the loads of the i^{th} iteration.

The `-sp 2ma` option uses a 2-stage pipe (Figure 6) where the loads of one iteration and the multiplies and adds of another iteration are placed in two separate pipe stages. The `-sp 2lm` option also uses a 2-stage pipe but groups the loads/multiplies and the adds together. Both these two-stage strategies result in core code that operates on blocks of size $kK_0 + 1$.

The software pipelining options provide the potential for the memory system, multiply unit, and add unit to remain busy throughout the core code block. A good instruction scheduler, however, is still required to take advantage of its independent instructions. While maintaining comperable performance, the extent of loop unrolling (i.e., the value of K_0) can theoretically decrease leading to more and potentially better sizes for the L1 cache-block (described below). In practice, results show that software pipelining (in particular, the `2ma` strategy) results in an appreciable core-code speedup.

3.2 Matrix Multiply L1 and/or L2 Cache Blocking Code

Because of the separation between matrix dimension and matrix stride, we can implement higher levels of cache blocking as calls to lower level routines with appropriately sized sub-matrices. `mm_lgen`, our cache-blocking code generator, achives this by producing code that uses the sub-matrix mutliply code produced either by `mm_cgen` or by `mm_lgen` itself.

`mm_lgen` supports the following command line options:

```
Usage: mm_lgen [OPTIONS]
where [OPTIONS] include:
Semantics options:
  -opA [N|T]                  : A matrix op. Normal|Transpose
  -opB [N|T]                  : B matrix op. Normal|Transpose
  -no_m_fringe                : don't generate an M reg block fringe
  -no_k_fringe                : don't generate a K reg block fringe
  -no_n_fringe                : don't generate an N reg block fringe
  -no_fringes                 : don't generate an M,K, or N reg block fringes
  -alpha [<val>|c]            : fix alpha at value <val> or arbitrary
  -beta [<val>|c]             : fix beta at value <val> or arbitrary
Optimization options:
  -l0 M0 K0 N0                : register (L0) blocking parameters
  -l1 M1 K1 N1                : L1 blocking parameters
  -calldown                   : check/call down lower routines first
Precision options:
  -prec [single|double|ldouble]  : Precision
  -sprec [single|double|ldouble] : Source Precision
  -aprec [single|double|ldouble] : Accumulator Precision
  -dprec [single|double|ldouble] : Destination Precision
Core code routine names options:
  -gen_rout  name          : General MM routine
  -gen_nf_rout  name       : General nofring (M0,K0,N0) routine
  -fixed_rout  name        : Fixed (M0M1,K0K1,N0N0) routine
  -sp [1|2lm|2ma|3]        : software pipelining option for core routine.
Code generation & Misc. options:
  -help                       : Print this message
  -file name                  : Write to file 'name'
  -routine_name name          : Name of resulting routine
  -spacechar c                : char to use as space
  -numspaces i                : spaces per nest
  -version                    : print version and exit.
```

Many of the options are similar to that of `mm_cgen`. We describe the set of operations performed by the routine produced by `mm_lgen` and in doing so describe the options that do not have an obvious similarity with `mm_cgen`.

The matrixes A, B, and C can be divided into regions depending on the values of the L0 and L1 blocking parameters. For example, figure 7 shows a division of the C matrix into three regions, I, II, and III. Region III is a submatrix whose dimensions are multiples of $M_1 M_0$ and $N_1 N_0$ respectively. Region II corresponds to three matrices whose dimensions are mutliples of M_0 and N_0 but where the multiples are less than M_1 and N_1 respectively. Region I corresponds to three matrices whos dimensions are less than the L0 blocking parameters.

In order to increase performance, `mm_lgen` can use an appropriate sub-matrix routine for different matrix regions. `mm_lgen` therefore takes the names of three routines, resolved at link time, that should be optimized for different conditions. The meanings are as follows:

- `gen_rout` specifies the name of a general matrix multiply routine that can be used on any size matrix. This argument must be present and, if no other routine names are given, this routine will be used for all regions of all the matrices.

- `gen_nf_rout` specifies the name of a "general but no fringe" matrix mutliply routine that only operates on matrices whose sizes satisfies $M = mM_0$, $K = kK_0$, and $N = nN_0$ for some non-negative integers m, k, and n (i.e., the matrices must be multiples of the L0 blocking sizes). If this argument is provided, `mm_lgen` calls this routine whenever sub-matrix sections are appropriately sized.

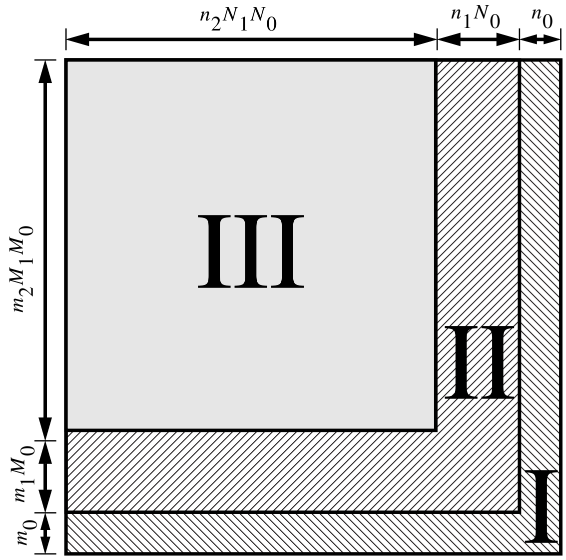


Figure 7: Regions of the C matrix according to the L0 and L1 blocking parameters. We assume that $M_1 > m_1 > 0$, $m_2 > 0$, $N_1 > n_1 > 0$, and $n_2 > 0$.

- `fixed_rout` specifies the name of a matrix mutliply routine that operates only on matrices whose sizes satisfy $M = mM_1M_0$, $K = kK_1K_0$, and $N = nN_1N_0$ (i.e., multiples of the L1 blocking sizes).

Figure 8 shows complete pseudocode for the operation performed by `mm_lgen`'s routine. This strategy was chosen to balance the tradeoff between overall code size and code size within a loop (due to I-cache limitations). Note that `fixed_rout` is only called once, but it will typically correspond to the largest matrix region (i.e., greatest number of FLOPS). The other two routines are called depending on the size of the remaining matrix fringes.

`mm_lgen` also takes a `-sp` option because it must know the blocking size offset in the K dimension that `fixed_rout` or `gen_nf_rout` use, and that depends on the software piplining option used to generate them. The search scripts (see Section 4) always ensure that `fixed_rout` and `gen_nf_rout` use the same software pipelining option, although there is no theoretical reason why this must always be the case.

3.3 Higher-level Cache Blocking

L2 cache blocking can also be performed using code produced by `mm_lgen`. In this case, however, the meaning of the arguments change slightly from the L1 blocking case. The `-l0` option becomes a dummy argument which must be set to `1 1 1`. The `-l1` option then specifies the L2 cache blocking size and is typically set to `-l1` $M_0M_1M_2$ $K_0K_1K_2$ $N_0N_1N_2$ for L2 parameters M_2, K_2, and N_2. Also, the `-sp` option should be set to `1` indicating no additional K-dimension offset. Finally, at least a single `-gen_rout` routine name must be given which specifies a general L1 blocked matrix multiply routine. An analogous strategy can be used to produce L3-blocked and even higher level code.

3.4 Routine Interface

The code generators produce routines which conform to the following interface:

```
void
mul_mpmp_mp(
            const int M,
            const int K,
            const int N,
            const <prec> *const A,
            const <prec> *const B,
```

```
for m ∈ M-Block
        for n ∈ N-Block
                for k ∈ K-block
                        call fixed_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_nf_rout();
                if any K-dimension remains
                        call gen_rout();
        if rest of N-dimension is a multiple of N0
                for k ∈ K-block
                        call gen_nf_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_nf_rout();
                if any K-dimension remains
                        call gen_rout();
        if any N-dimension remains
                for k ∈ K-block
                        call gen_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_rout();
                if any K-dimension remains
                        call gen_rout();
if rest of M-dimension is a multiple of M0
        for n ∈ N-Block
                for k ∈ K-block
                        call gen_nf_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_nf_rout();
                if any K-dimension remains
                        call gen_rout();
        if rest of N-dimension is a multiple of N0
                for k ∈ K-block
                        call gen_nf_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_nf_rout();
                if any K-dimension remains
                        call gen_rout();
        if any N-dimension remains
                for k ∈ K-block
                        call gen_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_rout();
                if any K-dimension remains
                        call gen_rout();
if any M-dimension remains
        for n ∈ N-Block
                for k ∈ K-block
                        call gen_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_rout();
                if any K-dimension remains
                        call gen_rout();
        if rest of N-dimension is a multiple of N0
                for k ∈ K-block
                        call gen_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_rout();
                if any K-dimension remains
                        call gen_rout();
        if any N-dimension remains
                for k ∈ K-block
                        call gen_rout();
                if rest of K-dimension is a multiple of K0
                        call gen_rout();
                if any K-dimension remains
                        call gen_rout();
```

Figure 8: The structure of `mm_lgen`'s L1 and higher level cache blocking matrix multiply code.

```
<dprec> *const C,
const int Astride,
const int Bstride,
const int Cstride
[,const <prec> alpha]
[,const <prec> beta]);
```

Where, assuming alpha and beta are present, the operation corresponds to $C = \alpha AB + \beta C$. mul_mpmp_mp is the default generated routine name but where the character 'p' is normally 'f', 'd', or 'l' for single, double, or extended precision. <prec> is the source matrix precision and <dprec> is the destination matrix precision. A, B, and C are respectively $M \times K$, $K \times N$, and $M \times N$ matrices. Astride (respectively Bstride and Cstride) is the number of elements between successive elements in a column of A (respectively, B and C). In other words, Astride (resp. Bstride, Cstride) is the number of elements in the leading dimension of A (resp. B, C). The alpha and beta parameters are optionally present depending on the values of the -alpha and -beta command line arguments.

The meanings of the above parameters slightly change when transpose operations are specified. For example, to generate code for the operation: $C = \alpha A^T B^T + \beta C$, the command line is

```
mm_cgen -l0 M0 K0 N0 -l1 M1 K1 N1 \
   -prec single -alpha c -beta c -opA T -opB T
```

The parameter semantics are now: transpose(A) (respectively transpose(B)) is a $M \times K$ (resp. $K \times N$) matrix. Astride (resp. Bstride) is, again, the number of elements between successive elements in a column of A (resp. B). The default generated routine name also changes to mul_mptmpt_mp.

For example, suppose we have three single precision matrices A, B, and C that are respectively of size $M \times K$, $K \times N$, and $M \times N$. For the operation $C = 0.1AB + 0.0B$, we would call the generator as:

```
mm_cgen -cb M0 K0 N0 -prec single -alpha c -beta c
```

which would produce a routine called as:

```
mul_mfmf_mf(M,K,N,A,B,C,K,N,N,0.1,0.0);
```

If we, alternatively, knew a priori that α is fixed at 0.1 and β at 0.0, we would generate code using:

```
mm_cgen -cb M0 K0 N0 -prec single -alpha 0.1 -beta 0.0
```

and we would call

```
mul_mfmf_mf(M,K,N,A,B,C,K,N,N);
```

As another example, suppose we have three single precision matrices A, B, and C that are respectively of size $K \times M$, $N \times K$, and $M \times N$. For the operation $C = 3A^T B^T + 10C$, we would generate code using:

```
mm_cgen -l0 M0 K0 N0 -l1 M1 K1 N1 \
    -prec single -alpha c -beta c -opA T -opB T
```

and we would call the transpose-transpose routine as

```
mul_mftmft_mf(M,K,N,A,B,C,M,K,N,3.0,10.0);
```

As yet another example, here is the calling sequence for non- sub-matrix-matrix multiply (i.e., the strides equal the sizes) for all transposition possibilities; normal-normal (NN), normal-transpose (NT), transpose-normal (TN), and transpose-transpose (TT):

```
/* NN, size(A) = MxK, size(B) = KxN, size(C) = MxN */
mul_mfmf_mf(M,K,N,A,B,C,K,N,N);
/* NT, size(A) = MxK, size(B) = NxK, size(C) = MxN */
mul_mfmft_mf(M,K,N,A,B,C,K,K,N);
/* TN, size(A) = KxM, size(B) = KxN, size(C) = MxN */
mul_mftmf_mf(M,K,N,A,B,C,M,N,N);
/* TT, size(A) = KxM, size(B) = NxK, size(C) = MxN */
mul_mftmft_mf(M,K,N,A,B,C,M,K,N);
```

Recall that the C language stores matrices in row-order where the rows are the leading dimension (i.e., consecutive memory locations are successive row elements) but Fortran stores matrices in column-order. It is still possible, however, to use a C-based multiplier for Fortran matrices. For example, suppose you have three Fortran matrices A, B, and C where A's size $M \times K$, B's size is $K \times N$, and C's size is $M \times K$. To do a NN matrix multiply in Fortran using a C routine, you would call the C NN routine as:

```
mul_mfmf_mf(N,K,M,B,A,C,K,M,M);
```

If A's size is $M \times K$, B's size is $N \times K$, and C's size is $M \times K$, to do a NT matrix multiply using a C routine, you would call the C TN routine as:

```
mul_mftmf_mf(N,K,M,B,A,C,N,M,M);
```

The TN and TT cases are performed in a similar way.

4 PHiPAC v1.0 Matrix Multiply Search Engine

The PHiPAC v1.0 matrix-multiply search scripts take parameters describing the machine architecture, including the number of integer and floating-point registers and the sizes of each level of cache. For each combination of generator parameters and compilation options, the matrix multiply search script calls the generator, compiles the resulting routine, links it with timing code, and benchmarks the resulting executable.

To produce a complete BLAS GEMM routine, we find separate parameters for each of the four cases $A \times B$, $A^T \times B$, $A \times B^T$, and $A^T \times B^T$. For each case, the overall code is structured as described in Section 3.2. The search script performs the following top-level set of operations:

1. Find the best register (L0) blocking parameters M_0, K_0, and N_0. These are called the *L0 core* parameters and are used for sections II and III of the matrix in Figure 7.

2. Find the best L1-cache blocking parameters M_1, K_1, and N_1.

3. Optionally, find the best L0 blocking parameters M_0^g, K_0^g, and N_0^g used to generate code to handle the matrix fringes (see the `do_L0_gen_frng` option below). The resulting *L0 general* blocking parameters are separate from the L0 core parameters, and are used for portions of the matrix with fringes less than any of the L0 core parameters in the respective dimension.

4. Find the best L2-cache blocking parameters M_2, K_2, and N_2.

After these steps have completed, code is generated for each of the resulting blocking parameters which is then used to produce a complete BLAS GEMM compatible matrix multiply.

Unlike the code generators, the search script options are specified in parameter files. The following sections describe each of the above procedures in detail by giving descriptions of all relevant search script options. The actual syntax of the parameter files is described in example files that are included with the distribution.

4.1 Search Procedure

The PHiPAC v1.0 distribution contains: (1) the code generator for generating matrix-matrix multiply code in ANSI C, (2) timing libraries to benchmark the performances of the generated matrix-matrix multiply code, and (3) search scripts to find the code generator's parameters that yield the code with the highest performance for a given system.

The basic top level command line call to the search script is:

```
search.pl [-long|-default|-short] -machine machine_specs -prec [single|double] \
         -ccopt compiler_options -level [0|1|2]
```

The `-machine` option gives a file describing the machine specifications and the `-ccopt` option gives a file describing the desired compiler optimization options. The `-level` option says whether to search just for the register blocking parameters, or to also perform an L1 or L2 search.[2] The `-long`, `-default`, and `-short` options say whether to do a long, medium, or short search. These options cause the search script to load different files that specify differing degrees of search thoroughness – the meanings of long, medium, and short are independent of the search scripts. The parameter files can therefore be modified to create a custom search.

The top level directory contains the following subdirectories:

- `mm_gen-2.0/`: This subdirectory contains the C source code for the `mm_cgen` and `mm_lgen` code generators. They can be independently compiled on your system to yield the executable code generator. Alternatively, the search scripts can be run to compile them automatically.

- `ipm-2.0/, rprf-v0_19/` Interval Performance Monitor, version 2.0.[Asaa] and Realization group (at ICSI) Performance measurement library, alpha version 0.19. [Asab].

  These are the timing libraries PHiPAC uses for benchmarking generated code. They provide a uniform interface to a wide variety of machine timers on a variety of platforms. These libraries are not documented in this paper.

- search-2.2/: This subdirectory contains PERL search scripts that find blocking parameters for the code generator to produce optimally performing code on your system.

- runs/: This subdirectory contains a self-explanatory set of example parameter files to be used with the search scripts.

4.1.1 Getting a BLAS compatible GEMM

The PHiPAC search scripts are capable of finding parameters to produce optimal BLAS compatible SGEMM or DGEMM routines. We describe how to do this below.

1. In the `runs/` subdirectory, make a copy of the `template/` subdirectory to one corresponding to the machine on which you plan to to run the search script. For instance, suppose your machine is named `myhost`; then you can do the following:

   ```
   % cd PHiPAC/runs/
   % cp -r template myhost
   ```

   The files in the `template/` subdirectory should not be changed (and will not be used by the search scripts), since you may need to refer to them (or recopy them) when you run the search scripts on other hosts. When you make the copy as above, the files in the `myhost/` subdirectory will still have the prefix tag `template`. In subsequent steps, we shall assume you have renamed these files to remove the prefix tag.

2. Edit the `machine_specs` file in the `runs/myhost` subdirectory to reflect the characteristics of the machine (see Section 4.2). You need to supply the following information:

   - number of single precision registers (if doing single precision search)
   - number of double precision registers (if doing double precision search)
   - L1 cache size in bytes
   - L2 cache size in bytes (if it exists)
   - L3 cache size in bytes (if it exists)
   - ANSI C compiler to be used

   If the machine does not have an L2 (or L3) cache, the corresponding information may be omitted. If the `-level` option of the top level search script indicates a higher level cache than that which size information has been provided, the size information will be guessed using the size given for the previous cache level (see the `L1_cache_size, L2_cache_size, L3_cache_size` options in Section 4.2).

[2] In the machine specifications file, described in Section 4.2, you must provide the sizes of the L1 (and perhaps also the L2 and L3) cache. Just providing those sizes, however, do not ensure that the corresponding cache level is searched. It is the `-level` option that controls the depth of the cache search.

3. Edit the `compiler_options` file in the `runs/myhost` subdirectory to list the compiler optimization options the search script should try (see Section 4.3). It is common to specify just one set of compiler optimization flags since an entire search will be performed for each one.

4. Decide on the cache blocking levels (0, 1 or 2) you wish to search over. A higher level of blocking yields code with a better performance on large matrices, but it also takes a much longer time. Usually, the level of blocking should be the same as the number of levels of cache available on the machine (e.g., a machine with only L1 cache should be search for only level 1 blocking).

5. Decide on the thoroughness of the search (i.e., long, default or short). A long search is fully exhaustive, but takes an extremely long time to finish (say, several weeks). A short search is much quicker (say, within a day or two) but may not yield the best code. The default search is a compromise between these two extremes. The actual search run time, however, depends on characteristics of the specific machine running the search script.

6. To avoid spurious results, **make sure the machine is unloaded before starting any search**. Then start the search as follow:

   ```
   % cd runs/myhost
   % perl5 ../../search-2.2/search.pl [-long|-default|-short]  \
         -machine machine_specs -prec [single|double]          \
         -ccopt compiler_options -level [0|1|2]
   ```

   For instance, to do a long search for level 2 blocked DGEMM, you would run

   ```
   % perl5 ../../search-2.2/search.pl -long \
        -machine machine_specs -prec double \
        -ccopt compiler_options -level 2
   ```

   To do a short search for level 1 blocked SGEMM, you would run

   ```
   % perl5 ../../search-2.2/search.pl -short \
      -machine machine_specs -prec single \
      -ccopt compiler_options -level 1
   ```

   To do a default search for register blocked DGEMM, you would run

   ```
   % perl5 ../../search-2.2/search.pl -default \
       -machine machine_specs -prec double \
       -ccopt compiler_options -level 0
   ```

   or simply

   ```
   % perl5 ../../search-2.2/search.pl \
       -machine machine_specs -prec double \
       -ccopt compiler_options -level 0
   ```

   As the search proceeds, files will be created in a performance directory – for the example above, the name of that directory would be `runs/myhost/perf/`. Each file contains the performance in MFLOPS for various blocking sizes and has names such as:

   L0CORE.{precision}.{matop}.{alphatype}.{softpipe}.{compopt}
   L0CORE_almstsq.{precision}.{matop}.{alphatype}.{softpipe}.{compopt}
   L0GEN.{precision}.{matop}.{alphatype}.{softpipe}
   L1.{precision}.{matop}.{alphatype}.{L1type}
   L2.{precision}.{matop}.{alphatype}

where

- L0CORE, LOCORE_almstsq, L0GEN, L1, L2 correspond respectively to performance numbers for the L0 fat-dot-product core, the L0 almost-square core, the L0 general, the L1, and the L2 blocking case.
- {precision} is the machine precision and is either `single` or `double`.
- {matop} is the matrix operation and is either `NN, NT, TN` or `TT`.
- {alphatype} is the alpha type and is either `c` or `1`.
- {softpipe} is the software pipelining option and is either `1A, 1B, 2ma, 2lm` or `3`.
- {compopt} is the tag from the compiler optimizations file (see Section 4.3).
- {L1type} indicates either `nofringe_general` or `fixed_nofringe_general` (see Section 4.5).

Some example file names include:

```
L0CORE.double.NN.1.1B.CC
L0CORE.double.NN.1.2ma.CC
L0CORE_almstsq.double.NN.1.1B.CC
L0CORE_almstsq.double.NN.1.2ma.CC
L0GEN.double.NT.c.1B
L0GEN.double.NT.c.2ma
L1.double.NT.1.nofringe_general
L1.double.NT.c.fixed_nofringe_general
L1.double.NT.1.nofringe_general
L1.double.NT.c.fixed_nofringe_general
```

The existence of each file indicates the completion of a checkpoint. That is, the search script will produce the performance numbers for the blocking parameters corresponding to one file and only then create that file. If the search script is killed for some reason and then restarted, it will not regenerate the performance numbers if the corresponding file exists, even if changes have been made to a parameter files. Therefore, if you first run, say, a `short` search and then later run a `default` one, it might (depending on the contents of the corresponding parameter files) be necessary to delete the appropriate performance file.

When the search script is finished with an entire step of the procedure listed in Section 4, files with the names L0CORE_top.{precision}, L0GEN_top.{precision}, L1_top.{precision}, and L2_top.{precision} will be created that give the top performers in each category. The blocking numbers in these files are ultimately used to produce the resulting BLAS GEMM.

7. When the search script terminates, a subdirectory called `PHiPAC_sgemm` or `PHiPAC_dgemm` will be created (depending on the chosen precision). You can then compile the corresponding GEMM library as:

```
% cd PHiPAC_sgemm or % cd PHiPAC_dgemm
% make % make
```

This will yield a `libsgemm.a` or `libdgemm.a` in that subdirectory. This is the PHiPAC optimized GEMM library routine for your machine.

8. At this stage, you can restart the whole search process from step 4 for the other GEMM library routine; for instance, if you have just generated the DGEMM library routine, you can repeat steps 4 to 7 for the SGEMM library routine.

9. We encourage you to send us your search results and the performance of the library routine on your machine so we may make it available for others to use. To do so, first go into the `runs/myhost` subdirectory:

```
% cd ..
```

Now benchmark the SGEMM library:

```
% perl5 ../../search-2.2/time_gemm.pl machine_specs single
```

and benchmark the DGEMM library:

```
% perl5 ../../search-2.2/time_gemm.pl machine_specs double
```

These will report the performance of the library routines on square matrix multiplication for a list of sizes such that all three matrices (A, B and C) occupy at most 16MB of memory.

After these benchmarking scripts have completed, do

```
% tar -cvf perf.tar perf
```

to create an archive of all performance readings obtained during the search and the benchmarking. Finally, make the file available to use by sending `phipac@icsi.berkeley.edu` mail containing a URL where we may obtain the resulting tar file.

4.2 Machine Specifications File

The machine specifications file provides all machine specific information to the search engine. This file is separate from the others as it might be used in future releases to search operations besides matrix multiply.

- `num_single_prec_registers, num_double_prec_registers, L1_cache_size, L2_cache_size, L3_cache_size`: The `num_single_prec_registers` and `num_double_prec_registers` parameters specify the number of single precision and double precision floating point registers respectively. The `L1_cache_size`, `L2_cache_size` and `L3_cache_size` parameters specify the sizes (in bytes) of the level-1, level-2 and level-3 data caches of the machine running the search script. If the machine running the search script has only a level-1 cache, then the `L2_cache_size` parameter may be omitted, and correspondingly for the `L3_cache_size`. In such case, when needed (i.e., if the `-level` option of the top level search specifies a higher level than cache size information is specified), these parameters are assumed to be eight times the size of the immediately preceding cache.

- `compiler`: The `compiler` parameter specifies the path name of the ANSI-C compiler to use. Additional options to be passed to the compiler may also be specified. For instance, a compiler that supports different versions of the C language may be given the option that specifies ANSI-C conformance. Options to increase the maximum macro size supported by the C pre-processor should also be provided here. These additional options are given every time the compiler is called. Therefore, code-optimization options should not be specified here. Instead, those options should be listed in the file specified by the `compiler_options_file` parameter.

- `generator_opts`: Any additional options to be passed to the generator (besides those controlled by the search script itself) can be given through the `generator_opts` parameter. These additional options are given every time the code generator is called. For a normal run of the search script, no additional generator options is necessary.

- `timer_args`: The `timer_args` parameter specifies any options to be passed to the underlying IPM/RPRF [Asaa, Asab] timer program for benchmarking.

4.3 Compiler Specifications File

This file simply specifies the optimization options for the compiler that the search script should use to compile the matrix code. Multiple compiler options can be specified and an entire search will be performed for each one. For index purposes, each set of compiler options is preceded by a text tag.

4.4 Register (L0) Parameter Search

The core register block search evaluates all combinations of values of M_0 and N_0 where $1 \leq M_0 N_0 \leq$ NR and where NR is the number of machine floating-point registers. The above is searched for $K_0 \in \mathcal{K}_0$ where $\mathcal{K}_0$ is determined by the `k0_set` parameter (see below). Each blocking parameter triple is selected at random from the satisfying set and is used to generate code which is subsequently benchmarked.

The majority of the computation, especially for larger matrices, is performed by the core $M_0 \times K_0 \times N_0$ register blocked code. Our L0 core search strategy, therefore, produces code containing only this core (i.e., no fringe code), which decreases compile time, and for each L0 parameter set, we benchmark only a single matrix multiply with size $M = M_0$, $N = N_0$, and $K = kK_0$ for some large integer k (we call this a "fat" dot-product). The parameter k is chosen such that the three matrices occupy some percentage of the L1 cache (although this can be set to either the L2 or L3 depending on a search option). While this case ignores the cost of the M- and N-loop overhead, we have so far found that this approach produces good quality code in less time than had we searched larger matrices. We nevertheless also provide the option to search matrices that fit in cache and are *almost square*. See the options below.

- `one_stage_holdstripe_A, one_stage_holdstripe_B, two_stage_load_mult, two_stage_mult_add, three_stage` : These specify the software-pipelining options that are correspondingly passed to the code generator `mm_cgen` (see Section 3.1.2). Any combination of these options is allowed but at least one of them must be specified. All of them will be searched in an attempt to find the best.

- `alpha_equals_one, alpha_arbitrary`: If `alpha_equals_one` or `alpha_arbitrary` is set, separate code for the case when $\alpha = 1$ and $\alpha = c$ where $c \neq 1$ (respectively) is generated and benchmarked. Any combination of these options is allowed, but at least one of them must be set. The $\alpha = 1$ case requires fewer operations than the arbitrary case and might also, therefore, require different blocking parameters for optimal performance. As described in Section 3.1, `mm_cgen` generates different code for these two cases.

- `matop_NN, matop_NT, matop_TN, matop_TT`: If `matop_NN`, `matop_NT`, `matop_TN` or `matop_TT` is set, the code for Normal-Normal, Normal-Transpose, Transpose-Normal or Transpose-Transpose (respectively) matrix multiplication is generated and benchmarked. The differing routines are subsequently used for the resulting full BLAS GEMM.

- `auto_blockings, k0_set, blockings_file`: If the `auto_blockings` parameter is set, the set of L0 blocking triples (M_0, K_0, N_0) to be benchmarked will be generated automatically from the combination of the `k0_set` parameter (see below) and the number of floating point registers available in the appropriate arithmetic precision. The values of M_0 and N_0 will be restricted to a range in such a way that $M_0 N_0$ is no more than the number of machine registers. The values of K_0 will be taken from those given in the set specified by the `k0_set` parameter.

  If the `blockings_file` parameter is specified, however, the L0 blocking space is created from the blocking sizes listed in the file whose name is given by `blockings_file`. This option exists since, for certain workloads, it might be advantageous to search only a certain set of L0 blocking sizes (see the next option).

- `auto_matrixsizes, fill_cache_percentage, matrixsizes_file`: If the `auto_matrixsizes` parameter is set, each blocking triple will be benchmarked on an automatically generated set of matrix sizes chosen to make all three matrices fit within the cache (either L1, L2, or L3 depending on the options below). The resulting MFLOPS rate is taken to be the performance. The `fill_cache_percentage` value is the relative percentage of data cache that the three matrices should occupy. For example, if this is set to 80, the matrix sizes will be chosen so that all three matrices combine to fill 80 percent of the cache.

  If the `matrixsizes_file` parameter is specified, every set of L0 blocking sizes will be benchmarked on the set of matrix sizes listed in the file name. The resulting performance is taken to be the harmonic mean of the MFLOPS rate achieved by the code on all matrix sizes. This mode of performance measure is suitable for generating high performance code for a predetermined set(s) of matrix sizes (which are typically small). The `fill_cache_percentage` parameter has no effect when this mode of performance measure is used.

- `use_fatdot, use_almstsq`: When the `auto_matrixsizes` parameter is set, one out of two performance readings is chosen to represent the performance of the current L0 blocking. This L0 blocked code is

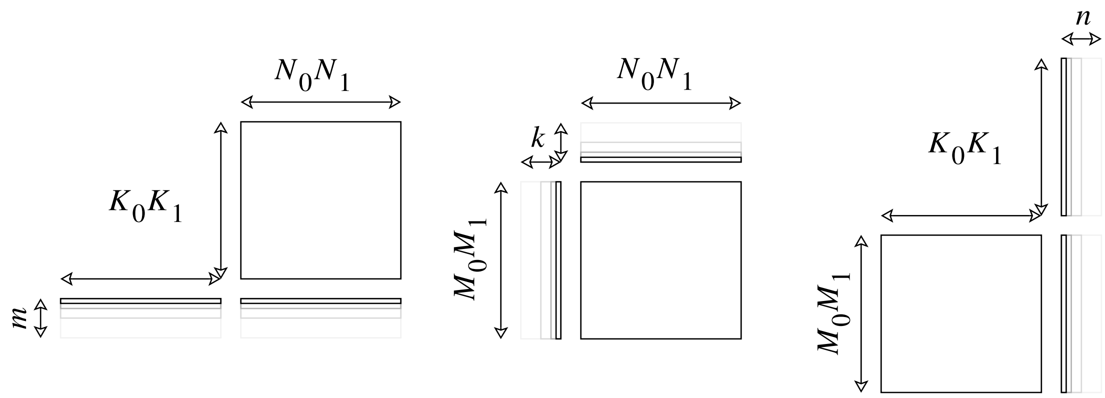


Figure 9: The L0 general search uses a workload consisting of matrix shapes as listed in this figure. The collection of matrix sizes used is as shown where m, k, or n takes on powers of two between 1 and respectively M_0, K_0, and N_0.

timed both for a fat dot-product matrix workload and for an almost-square matrix workload. A fat dot-product was described above. An almost square matrix has dimensions $M = mM_0$, $K = kK_0$, and $N = nN_0$ where m, k, and n are chosen so that the three matrices jointly occupy `fill_cache_percentage` percent of the cache and are as "square" shaped as possible. If the `use_fatdot` parameter is set, the performance for the fat dot-product workload is used as the ultimate performance for the current blocking size and if the `use_almstsq` parameter is set, the performance for the almost-square workload is used as the ultimate performance. Using different shapes to determine the ultimate performance will yield different optimal blocking sizes for differing matrix workloads.

- `do_L0_gen_frng`: When the `do_L0_gen_frng` option is set, an additional L0 blocking search is performed after the L1 search. This additional search finds the best L0 general blocking parameters for the matrix fringes (i.e., those portion of the matrices that are not a multiple of M_0, K_0, or N_0). It does this by, for each L0 blocking parameter triple, selecting and then benchmarking a set of matrix sizes that are similar in shape (see Figure 9) to the matrices that typically occur at the fringes. The ultimate result is an additional set of L0 blocking numbers M_0^g, K_0^g, or N_0^g which are used for the routine `gen_rout()` as listed in Figure 8. This option will also produce code that is faster on small matrices.
- `benchmark_l0_out_of_l2`: When the `benchmark_l0_out_of_l1` option is set, the matrix sizes that are used to benchmark the L0 parameters are set according to the L2 cache rather than the L1 cache. This is so that the core code instruction scheduling should be optimized for memory accesses that are typically out of L1 cache rather than within.
- `benchmark_l0_out_of_l3`: The `benchmark_l0_out_of_l3` is similar to the `benchmark_l0_out_of_l2` option except that the matrix size is chosen to be L3 sized rather than L2.

4.5 L1 Cache Block Search

We perform the L1 cache blocking search after the best core register blocking is known. We would like to make the L1 blocks large to increase data reuse but larger L1 blocks increase the probability of cache conflicts [LRW91]. Tradeoffs between M- and N- loop overheads, memory access patterns, and TLB structure also affect the best L1 size. We currently perform a relatively simple search of the L1 parameter space. For the D×D square case, we search the neighborhood centered at $3D^2 = L1$ where $L1$ is the L1 cache size in elements. We set M_1 to the values $\lfloor \phi D/M_0 \rfloor$ where $\phi \in \Phi$ (this is selected, as described below, by a parameter but a typical set is $\Phi = \{0.25, 0.5, 1.0, 1.5, 2.0\}$) and $D = \sqrt{L1/3}$. K_1 and N_1 are set similarly. We benchmark the resulting (for the example, 125) combinations with matrix sizes that either fit in L2 cache, or are within an upper bound (currently eight times the L1 size) if no L2 cache exists.

The following describes the search parameters relevant to L1 searching.

- `l1_use_nofringe_general, l1_use_fixed_nofringe_general`: These options select up to two different strategies that search for and time code that handles section III of the matrix in Figure 7. If only `l1_use_nofringe_general` is set, L1 blocking will be searched using only the general no fringe and general matrix subroutines (i.e., in Figure 8, the one `fixed_rout()` call will actually be a call to the `gen_nf_rout()` routine). If only `l1_use_fixed_nofringe_general` is set, L1 blocking will be searched using the fixed, general no fringe, and general matrix subroutines (i.e., in Figure 8, the one `fixed_rout()` call will actually be a call to a genuine fixed-size matrix routine). If both options are set, both cases will be timed and only the best performer will ultimately be retained. This option therefore potentially controls one aspect of the tradeoff between code-size and performance.

- `m_ratios, k_ratios, n_ratios`: These parameters specify the factors in the L1-block ratio space (i.e., the set Φ as described above). The cross-product of the three sets is used to compute the ratio triples (so if each ratio parameter contains five values, the result is 125 ratio triples). Given the optimal L0 blocking sizes (M_0,K_0,N_0) and an L1-block ratio triple (m_r,k_r,n_r), the L1 blocking sizes (M_1,K_1,N_1) are computed as follows. Using the L1 data cache size, the largest S is computed such that the three matrices A, B and C will together require `fill_cache_percentage` percent of the L1 cache assuming they are $S \times S$ square. Then, M_1, K_1, and N_1 are computed as the largest integers for which the L1 blocked matrix ($M_1 M_0$, $K_1 K_0$, $N_1 N_0$) is less than or equal to ($m_r S$, $k_r S$, $n_r S$).

- `fill_cache_percentage`: The `fill_cache_percentage` value is the percentage of the L2 data cache that is occupied by matrices used to benchmark the L1 blocking numbers.

- `benchmark_l1_out_of_l3`: When the option `benchmark_l1_out_of_l3` is set, the matrix sizes that are used to benchmark the L1 parameters are set according to the L3 cache rather than the L2 cache. This is so that the search should optimize the L1 blocking for memory accesses that are typically out of L2 cache rather than within.

4.6 L2 Cache Block Search

The L2 cache blocking search, when done, is performed similar to the L1 search. The following options are available.

- `m_ratios, k_ratios, n_ratios`: These parameters specify how to compute the set of L2 blocking sizes to be benchmarked. Like in the L1 case, they specify ratios of a square size but in this case the square size is computed using the L2 cache size.

- `fill_cache_percentage`: The `fill_cache_percentage` value is analogous to the L1 case, but here it selects a percentage the L3 data cache size.

4.7 Short/Default/Long search

As mentioned in Section 4.1, the `-short`, `-default`, and `-long` flags to the top level search script each indicate a different set of configuration files. Each of `-short`, `-default`, and `-long` specifies three files which indicate how thoroughly to perform the search. These files live in the `search-2.2/` directory and are called respectively:

```
opt_short.L0_search
opt_short.L1_search
opt_short.L2_search
opt_default.L0_search
opt_default.L1_search
opt_default.L2_search
opt_long.L0_search
opt_long.L1_search
opt_long.L2_search
```

To create a custom search, therefore, the above files should be modified. The files named `L0_search`, `L1_search`, and `L2_search` in the `runs/myhost` directory exist only for documentation and are not actually read by the search script.

	Sun Sparc-20/61	HP 712/80i	IBM RS/6000-590	SGI Indigo R4K	SGI Challenge	SGI Octane
Processor	SuperSPARC+	PA7100LC	RIOS-2	R4K	R8K	R10K
Frequency (MHz)	60	80	66.5	100	90	195
Max Instructions/cycle	3	2	6	1	4	4
Peak MFLOPS (32b/64b)	60/60	160/80	266/266	67/50	360/360	390/390
FP registers (32b/64b)	32/16	64/32	32/32	16/16	32/32	32/32
L1 Data cache (KB)	16	128	256	8	-	32
L2 Data cache (KB)	1024	-	-	1024	4096	1024
OS	SunOS 4.1.3	HP-UX 9.05	AIX 3.2	Irix 4.0.5H	IRIX6.2	IRIX6.4
C Compiler	Sun acc 2.0.1	HP c89 9.61	IBM xlc 1.3	SGI cc 3.10	SGI cc	SGI cc
Search results						
PHiPAC version	alpha	alpha	alpha	alpha	new	new
Precision	32b	64b	64b	32b	64b	64b
M_0,K_0,N_0	2,4,10	3,1,2	2,1,10	2,10,3	2,4,14	4,2,6
M_1,K_1,N_1	26,10,4	30,60,30	105,70,28	30,4,10	200,80,25	12,24,9
CFLAGS	-fast	-O	-O3 -qarch=pwr2	-O2 -mips2	-n32 -mips4 -O3	

Table 1: Workstation system details and results of matrix multiply parameter search.

5 Results

We ran the search scripts to find the best register and L1 cache blocking parameters for six commercial workstation systems. These systems have different instruction set architectures and widely varying microarchitectures and memory hierarchies. The results are summarized in Table 1.

The SGI R8K and R10K searches used version 1.0 of the code generator and search scripts, while other results were obtained with our earlier PHiPAC alpha release. Figures 10–13 plot the performance of the resulting routines for all square matrices $M = K = N = D$, where D runs over powers of 2 and 3, multiples of 10, and primes, up to a maximum of 300. We compare with the performance of a vendor-optimized BLAS GEMM where available. In each case, PHiPAC yields a substantial fraction of peak performance and is competitive with vendor BLAS. Due to limited availability, we could only perform an incomplete search on the R8K and R10K, and so these are preliminary performance numbers. There is also potential for improvement on the other machines when we rerun with the newer version. For completeness, we also show the poor performance obtained when compiling a simple three nested loop version of GEMM with FORTRAN or C optimizing compilers.

The PHiPAC methodology can also improve performance even if there is no scope for memory blocking. In Figure 16 we plot the performance of a dot product code generated using PHiPAC techniques. Although the parameters used were obtained using a short manual search, we can see that we are competitive with the assembly-coded SGI BLAS SDOT.

In some of the plots, we see that PHiPAC routines suffer from cache conflicts. Our measurements exaggerate this effect by including all power-of-2 sized matrices, and by allocating all regions contiguously in memory. For matrix multiply, we can reduce cache conflicts by copying to contiguous memory when pathological strides are encountered [LRW91]. Unfortunately, this approach does not help dot product. One drawback of the PHiPAC approach is that we can not control the order compilers schedule independent loads. We've occasionally found that exchanging two loads in the assembly output for dot product can halve the number of cache misses where conflicts occur, without otherwise impacting performance.

6 Status, Availability, and Future Work

This paper has demonstrated our ability to write portable, high performance ANSI C code for matrix multiply using parameterized code generators and a timing-driven search strategy. We have described the PHiPAC V1.0 release which contains matrix multiply generators, search scripts written in `perl`, and timing libraries.

In general, the current PHiPAC release finds extremely good L0 core matrix multiply routines. Our current strategy,

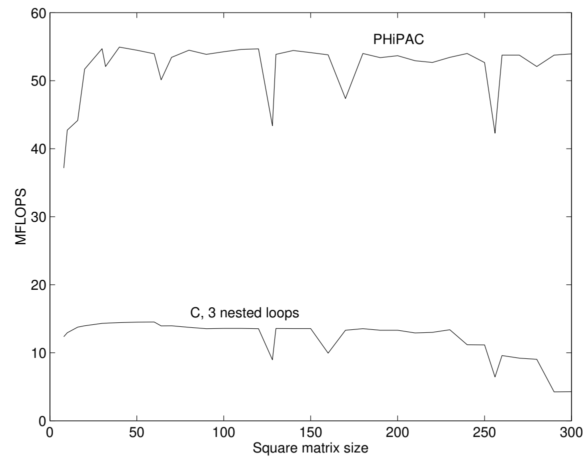


Figure 10: Performance of single precision matrix multiply on a Sparcstation-20/61.

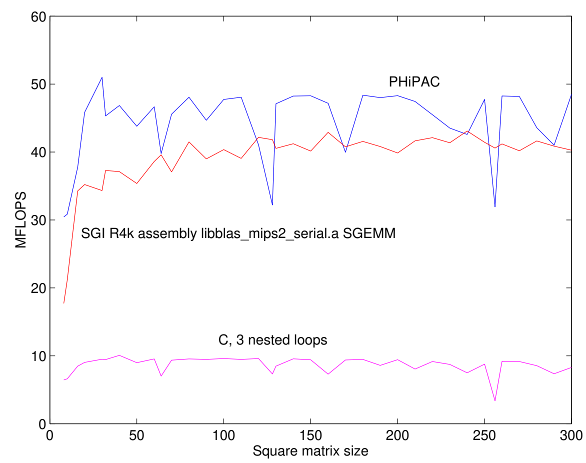


Figure 11: Performance of single precision matrix multiply on a 100 MHz SGI Indigo R4K. We show the SGEMM from SGI's `libblas_mips2_serial.a` library.

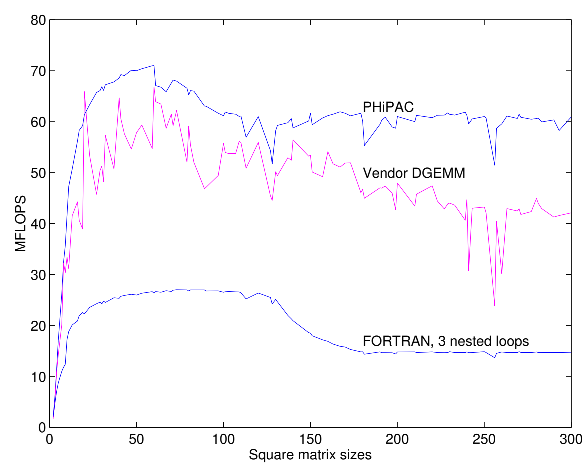


Figure 12: Performance of double precision matrix multiply on a HP 712/80i. We show DGEMM from the `pa1.1` version of `libvec.a` in HP's compiler distribution.

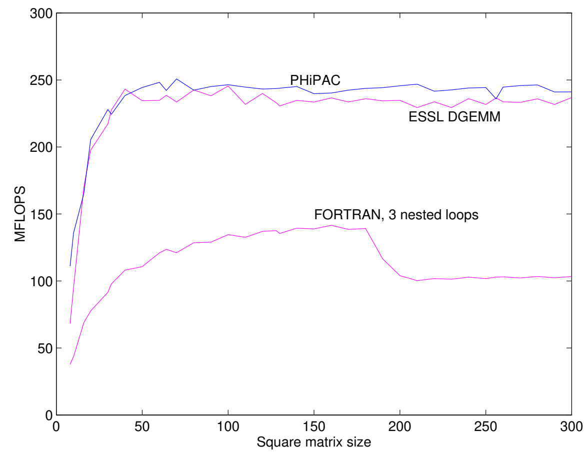


Figure 13: Performance of double precision matrix multiply on an IBM RS/6000-590. We show the DGEMM from IBM's POWER2-optimized ESSL library.

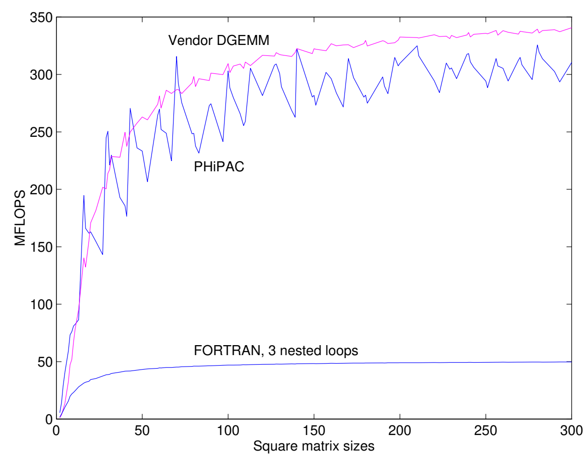


Figure 14: Preliminary performance of double precision matrix multiply on an SGI R8K Power Challenge. We show the DGEMM from SGI's R8K-optimized `libblas`.

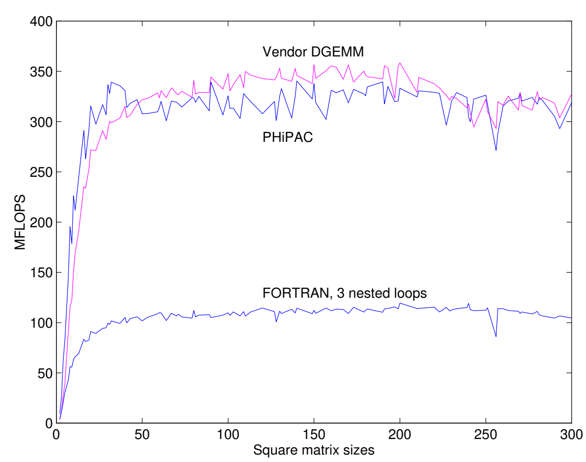


Figure 15: Preliminary performance of double precision matrix multiply on an SGI R10K Octane. We show the DGEMM from SGI's R10K-optimized `libblas`.

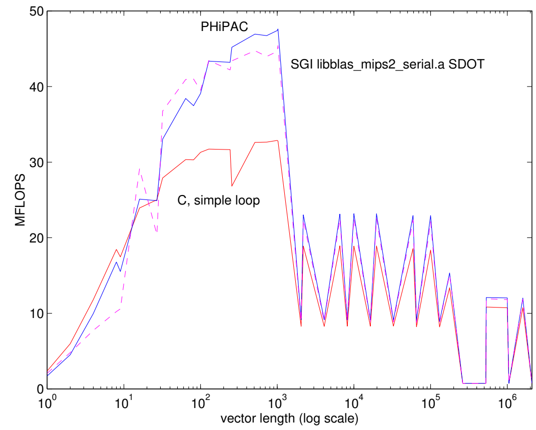


Figure 16: Performance of single precision unit-stride dot-product on a 100 MHz SGI R4K.

however, for decomposing matrices for performing L1 and L2 matrix multiply and the associated search strategy is rather naive. We are currently working, therefore, on a better L1 and L2 blocking strategy and accompanying methods for search based on more intelligent criteria [LRW91].

The PHiPAC GEMM can be used with Bo Kågström's GEMM-based BLAS3 package [BLL93] and LAPACK [ABB+92]. We have also written parameterized generators for matrix-vector and vector-matrix multiply, dot product, AXPY, convolution, and outer-product, and further generators, such as for FFT, are planned.

We have created a Web site from which the release, and all relevant documentation, is available and on which we plan at some point to list blocking parameters and GEMM libraries for many systems [BAD+].

We wish to thank Ed Rothberg of SGI for help obtaining the R8K and R10K performance plots. We also wish to thank Nelson Morgan who provided initial impetus for this project, Dominic Lam for work on the initial search scripts, and Richard Vudoc and Sriram Iyer for new work on the PHiPAC project.

References

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. LAPACK users' guide, release 1.0. In *SIAM*, Philadelphia, 1992.

[ACF95] B. Alpern, L. Carter, and J. Ferrante. Space-limited procedures: A methodology for portable high-performance. In *International Working Conference on Massively Parallel Programming Models*, 1995.

[AGZ94] R. Agarwal, F. Gustavson, and M. Zubair. *IBM Engineering and Scientific Subroutine Library, Guide and Reference*, 1994. Available through IBM branch offices.

[Asaa] K. Asanović. The IPM WWW home page. `http://www.icsi.berkeley.edu/~krste/IPM.html`.

[Asab] K. Asanović. The RPRF WWW home page. `http://www.icsi.berkeley.edu/~krste/RPRF.html`.

[BACD97] J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC.

[BAD+] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. The PHiPAC WWW home page. `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[BAD+96] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. LAPACK working note 111, University of Tennessee, 1996.

[BLL93] B.Kågström, P. Ling, and C. Van Loan. Portable high performance GEMM-based level 3 BLAS. In R.F. Sincovec et al., editor, *Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993. SIAM Publications.

[BLS91] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:97–371, 1991.

[CDD+96] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPAC: A portable linear algebra library for distributed memory computers - design issues and performance. LAPACK working note 95, University of Tennessee, 1996.

[CFH95] L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.

[DCDH90] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[DCHH88] J. Dongarra, J. Du Cros, S. Hammarling, and R.J. Hanson. An extended set of FORTRAN basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14:1–17, March 1988.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[KHM94] C. Kamath, R. Ho, and D.P. Manley. DXML: A high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, Summer 1994.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[LRW91] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of ASPLOS IV*, pages 63–74, April 1991.

[MS95] J.D. McCalpin and M. Smotherman. Automatic benchmark generation for cache optimization of matrix algorithms. In R. Geist and S. Junkins, editors, *Proceedings of the 33rd Annual Southeast Conference*, pages 195–204. ACM, March 1995.

[SMP+96] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proceedings of the 10th International Parallel Processing Symposium*, April 15–19 1996.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[Wol96] M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.