Kernel (De)Composition

A Compositional Approach applied to Linear Algebra Libraries

Denis Barthou

[1] University of Versailles St Quentin, France

Dagstuhl 07

Outline

Automatic Generation of High Performance Libraries

- Context: high performance dense linear algebra libraries
- One Core Case: issues, performance prediction, results
- Multicore Case
- Concluding remarks

One Core Case: Problem Statement

High performance dense linear algebra library

- Automatic generation: ATLAS, PhiPAC.
  - Uses algorithmic knowledge,
  - Optimizes first for cache usage,
  - Explores optimization space by empirical search or model.
- Hand-tuned assembly: constructor library (MKL, ESSL), K.Goto's BLAS.

One Core Case: Problem Statement

High performance dense linear algebra library

- Automatic generation: ATLAS, PhiPAC.
  - Uses algorithmic knowledge,
  - Optimizes first for cache usage,
  - Explores optimization space by empirical search or model.
- Hand-tuned assembly: constructor library (MKL, ESSL), K.Goto's BLAS.

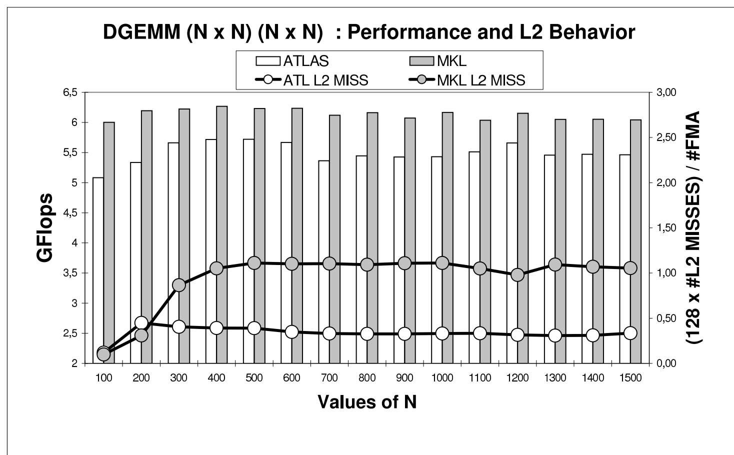
Hand-tuned code outperforms ATLAS (Itanium/Pentium).

Is there something missing in compilers and/or ATLAS ?

Performance Analysis MKL/ATLAS: L2 misses

ATLAS version 5.6, MKL version 8.02 on Itanium2

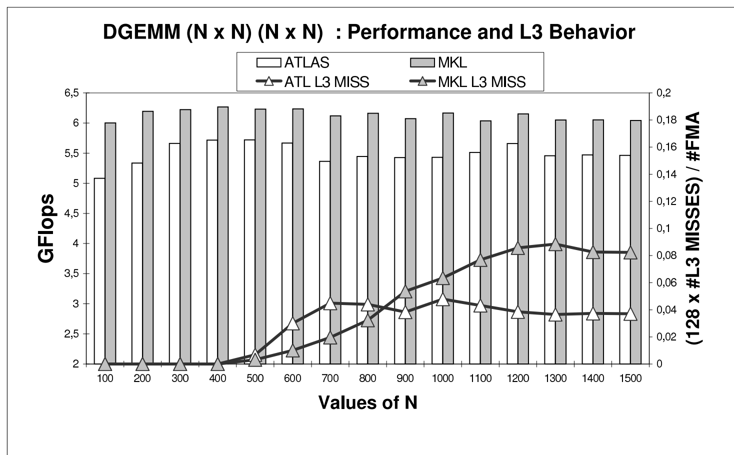
ICC compiler v9.0



Performance Analysis MKL/ATLAS: L3 misses

ATLAS version 5.6, MKL version 8.02 on Itanium2

ICC compiler v9.0



Issues for an Accurate Performance Model

Many different architecture mechanisms, some dynamic and particular behaviors make things hairy

- Cache behavior:
  - Automatic prefetch, cache replacement policy (the real one),
  - Impact of page allocation (TLB misses, cache conflicts due to non consecutive pages)
  - Limited bandwith to the cache, to individual memory banks
- Latency of a sequence of instructions:
  - Real impact of some instruction variants unclear
  - Out of order mechanism
  - Microcode and dispersal rules
  - Underspecified latencies for instruction sequences

Issues for an Accurate Performance Model

Many different architecture mechanisms, some dynamic and particular behaviors make things hairy

- Cache behavior:
  - Automatic prefetch, cache replacement policy (the real one),
  - Impact of page allocation (TLB misses, cache conflicts due to non consecutive pages)
  - Limited bandwith to the cache, to individual memory banks
- Latency of a sequence of instructions:
  - Real impact of some instruction variants unclear
  - Out of order mechanism
  - Microcode and dispersal rules
  - Underspecified latencies for instruction sequences

Even if the code is regular,
Some dynamic and specific case behaviors make things hairy

Issues for an Accurate Performance Model

Many different architecture mechanisms, some dynamic and particular behaviors make things hairy

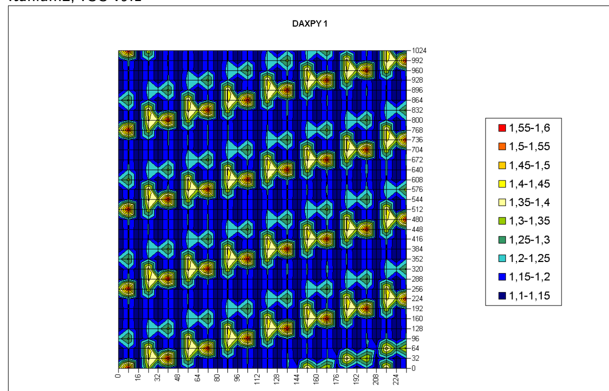
- Cache behavior:
  - Automatic prefetch, cache replacement policy (the real one),
  - Impact of page allocation (TLB misses, cache conflicts due to non consecutive pages)
  - Limited bandwith to the cache, to individual memory banks
- Latency of a sequence of instructions:
  - Real impact of some instruction variants unclear
  - Out of order mechanism
  - Microcode and dispersal rules
  - Underspecified latencies for instruction sequences

Even if the code is regular,
Some dynamic and specific case behaviors make things hairy
Producing reliable performance measures is difficult

Effects of cache memory banks

For a DAXPY code: $Y[i] = a * X[i] + Y[i]$, 200 iterations
Performance per iteration according to offset of X and Y:

Itanium2, ICC v9.1



ATLAS: Iterative search or performance model ?

Performance tuning by iterative search

- Optimizations set according to a underlying performance model: minimizes cache misses
  $\rightarrow$ Decompose matrices in square blocks for matrix multiplication
- Performance measured for different optimization parameters (tile sizes, unrolling factor)
- Best optimization parameters selected for library generation

Performance tuning with a performance model

- Optimizations and optimization parameters set according to performance model
- Performance prediction guides library generation

Mixed Approach Performance Measure/Prediction

Using only compiler and source to source transformations

1. Decompose function code into high performance kernels/codelets
2. Measure performance on these codelets
3. Predict library function code performance and optimize further with performance model
4. Build library function based on input data size and performance model

Mixed Approach Performance Measure/Prediction

Using only compiler and source to source transformations

1. Decompose function code into high performance kernels/codelets
   Kernel decomposition, CGO07
   - Explore different optimization sequences (tiling, unroll and jam, loop interchange, blocking)
     Xlanguage, LCPC07
   - Extract inner loops, simplify data layout
2. Measure performance on these codelets
3. Predict library function code performance and optimize further with performance model
4. Build library function based on input data size and performance model

Mixed Approach Performance Measure/Prediction

Using only compiler and source to source transformations

1. Decompose function code into high performance kernels/codelets
   Kernel decomposition, CGO07
   - Explore different optimization sequences (tiling, unroll and jam, loop interchange, blocking)
     Xlanguage, LCPC07
   - Extract inner loops, simplify data layout
2. Measure performance on these codelets
   - All data in cache (perform copies to be sure it is in cache)
   - Array alignment constant
3. Predict library function code performance and optimize further with performance model
4. Build library function based on input data size and performance model

Example of Decompositions for DGEMM

- Original tile

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

Example of Decompositions for DGEMM

- Original tile

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

- Unroll i and j loops

```
for (i = 0; i < NI; i+=2)
 for (j = 0; j < NJ; j+=2 )
  for (k = 0; k < NK; k++)
   c[i][j] += a[i][k] * b[k][j];
   c[i+1][j] += a[i+1][k] * b[k][j];
   c[i][j+1] += a[i][k] * b[k][j+1];
   c[i+1][j+1] += a[i+1][k] * b[k][j+1];
```

Example of Decompositions for DGEMM

- Original tile

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

- Unroll i and j loops

```
for (i = 0; i < NI; i+=2)
 for (j = 0; j < NJ; j+=2 )
  for (k = 0; k < NK; k++)
   c[i][j] += a[i][k] * b[k][j];
   c[i+1][j] += a[i+1][k] * b[k][j];
   c[i][j+1] += a[i][k] * b[k][j+1];
   c[i+1][j+1] += a[i+1][k] * b[k][j+1];
```

- Extracted kernel: dotproduct

```
for (k = 0; k < NK; k++)
 c00 += a0[k] * b0[k];
 c10 += a1[k] * b0[k];
 c01 += a0[k] * b1[k];
 c11 += a1[k] * b1[k];
```

dotproduct nm

```
for(i = 0 ; i < ni ; i++)
   c11 += a1[i] * b1[i];
   ...
   c1n += a1[i] * bn[i];
   c21 += a2[i] * b1[i];
   ...
   cmn += am[i] * bn[i];
```

Example of Decompositions for DGEMM

- Original tile

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

- Interchange j,k, Unroll i and k loops

```
for (i = 0; i < NI; i+=2)
 for (k = 0; k < NK; k+=2 )
  for (j = 0; j < NJ; j++)
   c[i][j] += a[i][k] * b[k][j];
   c[i+1][j] += a[i+1][k] * b[k][j];
   c[i][j] += a[i][k+1] * b[k+1][j];
   c[i+1][j] += a[i+1][k+1] * b[k+1][j];
```

dotproduct nm

```
for(i = 0 ; i < ni ; i++)
   c11 += a1[i] * b1[i];
   ...
   c1n += a1[i] * bn[i];
   c21 += a2[i] * b1[i];
   ...
   cmn += am[i] * bn[i];
```

Example of Decompositions for DGEMM

- Original tile

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

- Interchange j,k, Unroll i and k loops

```
for (i = 0; i < NI; i+=2)
 for (k = 0; k < NK; k+=2 )
  for (j = 0; j < NJ; j++)
   c[i][j] += a[i][k] * b[k][j];
   c[i+1][j] += a[i+1][k] * b[k][j];
   c[i][j] += a[i][k+1] * b[k+1][j];
   c[i+1][j] += a[i+1][k+1] * b[k+1][j];
```

- Extracted kernel: daxpy

```
for (j = 0; j < NJ; j++)
 c0 += a00 * b0[j];
 c1 += a10 * b0[j];
 c0 += a01 * b1[j];
 c1 += a11 * b1[j];
```

dotproduct nm

```
for(i = 0 ; i < ni ; i++)
   c11 += a1[i] * b1[i];
   ...
   c1n += a1[i] * bn[i];
   c21 += a2[i] * b1[i];
   ...
   cmn += am[i] * bn[i];
```

daxpy nm

```
for(i = 0 ; i < ni ; i++)
   c1[i] += a11 * b1[i];
   ...
   c1[i] += a1n * bn[i];
   c2[i] += a2n * b1[i];
   ...
   cm[i] += amn * bn[i];
```

Example of Decompositions for DGEMM

- Original tile

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

- Permute i and k

```
for (k = 0; k < NK; k++)
 for (i = 0; i < NI; i++ )
  for (j = 0; j < NJ; j++)
   c[i][j] += a[i][k] * b[k][j];
```

dotproduct nm

```
for(i = 0 ; i < ni ; i++)
   c11 += a1[i] * b1[i];
   ...
   c1n += a1[i] * bn[i];
   c21 += a2[i] * b1[i];
   ...
   cmn += am[i] * bn[i];
```

daxpy nm

```
for(i = 0 ; i < ni ; i++)
   c1[i] += a11 * b1[i];
   ...
   c1[i] += a1n * bn[i];
   c2[i] += a2n * b1[i];
   ...
   cm[i] += amn * bn[i];
```

Example of Decompositions for DGEMM

- Original tile

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

- Permute i and k

```
for (k = 0; k < NK; k++)
 for (i = 0; i < NI; i++ )
  for (j = 0; j < NJ; j++)
   c[i][j] += a[i][k] * b[k][j];
```

- Extracted kernel: outerproduct

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  c[i][j] += a[i] * b[j];
```

dotproduct nm

```
for(i = 0 ; i < ni ; i++)
   c11 += a1[i] * b1[i];
   ...
   c1n += a1[i] * bn[i];
   c21 += a2[i] * b1[i];
   ...
   cmn += am[i] * bn[i];
```

daxpy nm

```
for(i = 0 ; i < ni ; i++)
   c1[i] += a11 * b1[i];
   ...
   c1[i] += a1n * bn[i];
   c2[i] += a2n * b1[i];
   ...
   cm[i] += amn * bn[i];
```

outerproduct n

```
for (i = 0; i < ni ; i++)
   for (j = 0; j < nj ; j++)
     c[i][j] += a1[i] * b1[j];
     ...
     c[i][j] += an[i] * bn[j];
```

Kernel Optimization

Kernels tuned with two parameters:

- Loop bound values
  - Unrolling factor, SWP parameters, ...
- Array alignments
  - Vectorization
  - Memory bank conflicts

Rely on compiler for:

- Dependence analysis
- Vectorization (SIMD)
- Register allocation
- Instruction scheduling

Kernel Properties

Kernel Performance

- Independent of the application context,
- Only depends on cache level of data.

Additional benefits of kernels

- Execution time much lower than for whole application,
- Possible reuse among different applications.

Composition of Kernels

Original version

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

Composition of Kernels

Original version

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

Version1

```
for (i = 0; i < NI; i+=2)
 for (j = 0; j < NJ; j+=2 )
  for (k = 0; k< NK; k+=200)
   COPY(a[i][k:k+199]⟶ A[0:199])
   COPY(b[k:k+199][j]⟶ B[0:199])
   DOT-200(A[0:199],B[0:199],..)
```

if NK>200

Version2

```
for (i = 0; i < NI; i+=2)
 for (k = 0; k < NK; k+=2 )
  for (j = 0; j< NJ; j+=400)
   COPY(b[k][j:j+399]⟶ B[0:399])
   COPY(c[i][j:j+399]⟶ C[0:399])
   DAXPY-400(B[0:399],C[0:399],..)
```

if NJ > 400

Kernels are new vector instructions

Simplified performance model on these versions

Further optimizations possible (hoisting, coalescing copies...)

Composition of Kernels

Original version

```
for (i = 0; i < NI; i++)
 for (j = 0; j < NJ; j++)
  for (k = 0; k < NK; k++ )
   c[i][j] += a[i][k] * b[k][j];
```

Version1

```
for (i = 0; i < NI; i+=2)
 for (j = 0; j < NJ; j+=2 )
  for (k = 0; k< NK; k+=200)
   COPY(a[i][k:k+199]→ A[0:199])
   COPY(b[k:k+199][j]→ B[0:199])
   DOT-200(A[0:199],B[0:199],..)
```

if NK>200

Version2

```
for (i = 0; i < NI; i+=2)
 for (k = 0; k < NK; k+=2 )
  for (j = 0; j< NJ; j+=400)
   COPY(b[k][j:j+399]→ B[0:399])
   COPY(c[i][j:j+399]→ C[0:399])
   DAXPY-400(B[0:399],C[0:399],..)
```

if NJ > 400

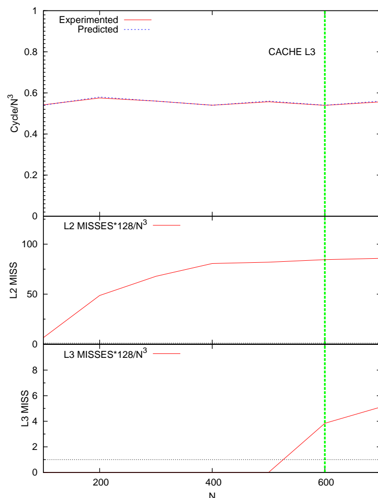
Kernels are new vector instructions

Simplified performance model on these versions

Further optimizations possible (hoisting, coalescing copies...)

Construction of library function by selection of the best versions

Quality of Performance Model



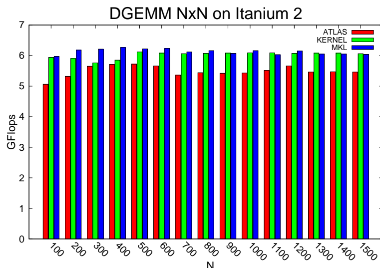
Itanium2, ICC v9.1

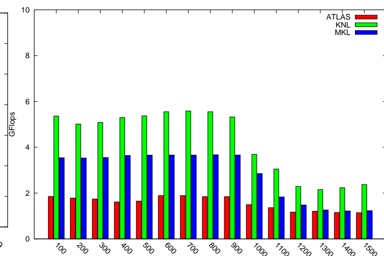
Performance predicted corresponds to measure

Results for matrix multiplication

Performance comparison for

- square matrices (left),
- rectangular matrices (Nx6)x(6xN)(right)



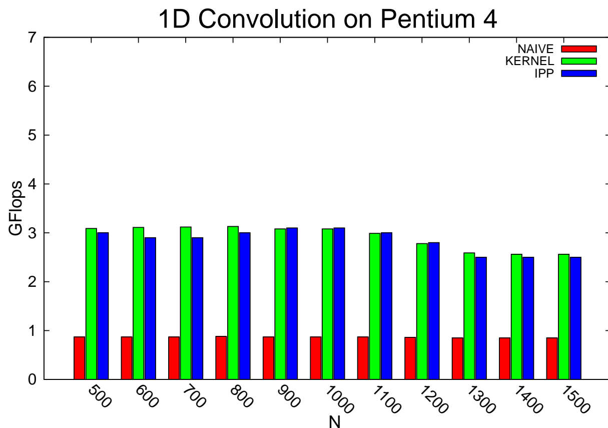


Itanium2, ICC v9.1

Results for 1D convolution

1D convolution

```
for(i=0;i<N-n;i++)
 for(j=0;j<2*n;j++)
  a[i] += b[j] * c[i-j+n]
```

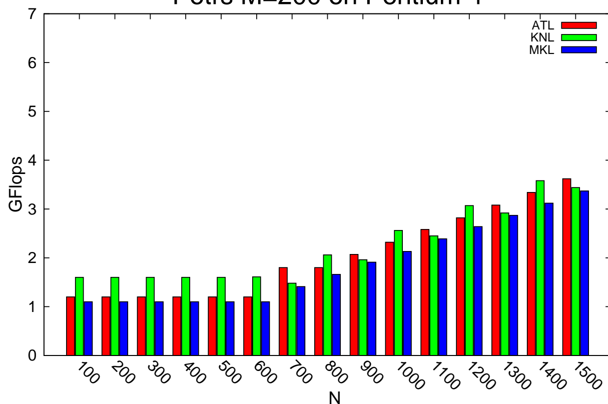


Pentium4 Prescott, ICC v9.1

Results for Cholesky resolution

POTRS

```
for (j=N; j>=1; --j)
 for (k=M; k>=1; --k)
  B[k][j] = alpha*B[k][j]
  B[k][j] /= A[k][k]
  for (i=0; i<k-1; i++)
   B[i][j] -= A[i][k]*B[k][j]
```



Pentium4 Prescott, ICC v9.1

Multicore and parallel library

Parallelism expression:

- Parallelization with pragmas (exploring different schedule) or user
- OpenMP

Criteria in kernel selection

- Single core performance according to input size
- Total bandwith required

Synchronization selection: if performance model is accurate, busy waiting is an option.

Preliminary results: 2% less performance than MKL for square matrices.

Multicore and parallel library

Parallelism expression:

- Parallelization with pragmas (exploring different schedule) or user
- OpenMP

Criteria in kernel selection

- Single core performance according to input size
- Total bandwith required

Synchronization selection: if performance model is accurate, busy waiting is an option.
Preliminary results: 2% less performance than MKL for square matrices.
Adaptation to topology for multinodes: if bandwith precisely described (or benchmarked), take into account topology of architecture with threads grouped by affinity (Marcel thread library).

Conclusion and remarks

Performance Prediction vs Performance Measure

- Performance predictability of small kernels can drive coarser grain optimizations
- Performance measure of these kernels: cheap way to avoid much complex performance model
- Small number of different kernels for many functions
- Counterpart:
  - suboptimal code due to introduction of copies
  - creation of kernels by large exploration of optimization space

Parallel library: Tradeoff between thread performance and

- Degree of parallelism
- Memory bandwith
- Synchronization costs