Automated Empirical Optimization of High Performance Floating Point Kernels

R. Clint Whaley
University of Texas, San Antonio

and

David B. Whalley
Florida State University
I. Introduction:
   a. Problem definition & traditional (partial) solutions
   b. Problems with Traditional Solutions
   c. Addressing these issues through empirical techniques
   d. Original empirical work – ATLAS

II. Our iterative and empirical compilation framework (iFKO)
   a. What is an empirical compiler?
   b. Repeatable transformations
   c. Fundamental transforms

III. Results
   a. Studied kernels
   b. Comparison of automated tuning strategies

IV. Future work

V. Summary and conclusions

VI. Related work
I(a). Problem Definition

Ultimate goal of research is to make extensive hand-tuning unnecessary for HPC kernel production:

- For many operations, no such thing as “enough” compute power
- Therefore, need to extract near peak performance even as hardware advances according to Moore’s Law
- Achieving near-optimal performance is tedious, time consuming, and requires expertise in many fields
- Such optimization is neither portable or persistent

**Traditional (partial) Solutions:**

- Kernel library API definition + hand tuning
- Compilation research + hand tuning
I(b). Problems with Traditional solutions

Hand-tuning libraries
- Demand for hand tuners outstrips supply
  → if kernel not widely used, will not be tuned
- Hand-tuning tedious, time consuming, and error prone
  → By time lib fully optimized, hardware on way towards obsolescence

Traditional Compilation
- This level of opt counterproductive
- Compilation models are too simplified
  → Must account for all lvels of cache, all PE interact, be spec to the kernel
  – Model goes out of date with hardware
- Resource allocation intractable a priori
- Many modern ISAs do not allow compiler to control machine in detail
I(c). Empirical Techniques Can Address These Problems

- **AEOS**: Automated Empirical Optimization of Software
- **Key idea**: make optimization decisions using automated timings:
  - Can adapt to both kernel and architecture
  - Can solve resource allocation prob backwards
- **Goal**: Optimized, portable library available for new arch in minutes or hours rather than months or years

**AEOS Requires:**
- Define simplest and most reusable kernels
- Sophisticated timers
- Robust search
- Method of code transformation:
  1. Parameterization
  2. Multiple implementation
  3. Source generation
  4. **Iterative empirical compiler**
I(d). Original AEOS Effort: Automatically Tuned Linear Algebra Software (ATLAS)

- Level 3 BLAS very well optimized
  - Pthreads for SMP support
  - Performance from gemm kernel:
    * Source gen + param
    * Mul. implem. + param
- Level 1 and 2 BLAS optimized
  - Mul. implem. + param
- ATLAS has unambiguously demonstrated that AEOS techniques represent a successful new paradigm for high performance optimization

ATLAS is widely used and cited:
- Problem Solving Environments
  - Maple, Matlab, Octave
- Operating systems
  - Apple’s OS 10.2, FreeBSD, and various Linux distributions
- Used by large range of individual projects
  - Usages ranging from scientific applications to home digital photography
- Highly cited in literature
  - 310 citeseer citations
II(a). Overview of iFKO Framework

iFKO composed of:
1. A collection of search drivers,
2. the compiler specialized for empirical floating point kernel optimization (FKO)
   - Specialized in analysis, HIL, type and flexibility of supported transforms

Drawbacks:
- External timers add significant overhead.
- Compile time expanded enormously.
  ⇒ Only use for extreme performance
II(b). Repeatable Optimizations

- Applied in any order,
- to a relatively arbitrary scope,
- in optimization blocks,
- while successfully transforming the code.
- Presently, not empirically tuned.

Supported repeatable transformations are:

1. **ra**: Register allocation (Xblock, wt. hoisting/pushing)
2. **cp**: Forward copy propagation (Xblock)
3. **rc**: Reverse copy propagation
4. **u1**: Remove one-use loads
5. **lu**: Last use load removal
6. **uj**: Useless jump elimination (Xblock)
7. **ul**: Useless label elimination (Xblock)
8. **bc**: Branch chaining (Xblock)
II(c). Fundamental Optimizations

- Applied only to optloop,
- Applied in known order (to ease analysis),
- Applied before repeatable opt (mostly high-level)
- Empirically tuned by search.
- Presently supported fundamental optimization (in application order, with search default shown in parentheses):
  1. **SV**: SIMD vectorization (if legal)
  2. **UR**: Loop unrolling (line size)
  3. **LC**: Optimize loop control (always)
  4. **AE**: Accumulator Expansion (None)
  5. **PF**: Prefetch (inst=’nta’, dist=2*LS)
  6. **WNT**: Non-temporal writes (No)
III(a). Studied kernels

- Start with Level 1 BLAS to concentrate on inner loop
  - ATLAS work shows main compilation problems in inner loop
- Speedups possible even on such simple (and bus-bound) operations
- Can already beat icc/gcc for Level 3, but not yet competitive with hand-tuned
- Results for two archs (p4e/opt) and two contexts (in/out cache)

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation Summary</th>
<th>flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap</td>
<td>for (i=0; i &lt; N; i++) {tmp=y[i]; y[i] = x[i]; x[i] = tmp}</td>
<td>N</td>
</tr>
<tr>
<td>scal</td>
<td>for (i=0; i &lt; N; i++) y[i] *= alpha;</td>
<td>N</td>
</tr>
<tr>
<td>copy</td>
<td>for (i=0; i &lt; N; i++) y[i] = x[i];</td>
<td>N</td>
</tr>
<tr>
<td>axpy</td>
<td>for (i=0; i &lt; N; i++) y[i] += alpha * x[i];</td>
<td>2N</td>
</tr>
<tr>
<td>dot</td>
<td>for (dot=0.0,i=0; i &lt; N; i++) dot += y[i] * x[i];</td>
<td>2N</td>
</tr>
<tr>
<td>asum</td>
<td>for (sum=0.0,i=0; i &lt; N; i++) sum += fabs(x[i])</td>
<td>2N</td>
</tr>
<tr>
<td>iamax</td>
<td>for (imax=0, maxval=fabs(x[0]), i=1; i &lt; N; i++) {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (fabs(x[i]) &gt; maxval)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{ imax = i; maxval = fabs(x[i]); }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>2N</td>
</tr>
</tbody>
</table>
III(b) 1. Relative speedups of various tuning methods

2.8Ghz Pentium4E, N=80000, out-of-cache
III(b)2. Relative speedups of various tuning methods

1.6Ghz Opteron, N=80000, out-of-cache
III(b)3. Relative speedups of various tuning methods

2.8Ghz Pentium4E, N=1024, in-L2-cache
III(b)4. Relative speedups of various tuning methods

1.6GHz Opteron, N=1024, in-L2-cache
III(c)5. Key points on results

- iFKO best tuning mechanism on avg ∀ architectures/contexts
  - IAMAX and COPY present only real losses
    → Lack of vectorization and block fetch
  - icc+prof slower than icc+ref for swap/axpy OC Opt
- All tuned paras provide speedup
  - Vary strongly by kernel, arch, & context
  - Vary weakly by precision
  - PF helps IC overcome conflicts
    - for OC, PF dist critical
    - for IC, AE and PF inst critical
- More bus-bound a kernel is, less PF helps
  - OC, iFKO gives more benefit for less bus-bound ops
V. Future work

Near-term:
1. Improve PF search
2. Software pipelining
3. Specialized array indexing
4. Outer loop UR (unroll & jam)
5. Scalar replacement for register blocking
6. PF of non-loop data
7. Multiple accumulator reduction optimization
8. Loop peeling for SV alignment

Long-term:
1. Block fetch
2. Loop invariant code motion
3. PPC/Altivec support
4. General SV alignment
5. Complex data type
6. Tiling/blocking
7. Search refinements
8. Timer resolution improvement
9. Timer generation
VI. Summary and conclusions

1. Have shown empirical optimization can auto-adapt to varying arch, operation, and context
2. Addressed kernel-specific adaptation in ATLAS work
3. Presented kernel-independent iFKO
4. Demonstrated iFKO can auto-tune simple kernels:
   - As kernel complexity and optimization set grows, empirical advantage should increase
     → Need increasingly sophisticated search
5. As we expand opt. support, need for hand-tuning in HPC should go down drastically
6. Will open up new areas of research (as ATLAS did):
   - iFKO can help build better models of archs
   - FKO provides realistic testbed for search optimization
VII. Related Work

1. ATLAS, FFTW, PHiPAC
   - Kernel-specific
   - High-level code generation

2. OCEANS
   - Handles very few transforms/kernels
   - Code generation at high-level
   - Degree of automation and generality unclear
   - Papers on search very different from our approach

   - Not empirical, uses iteration to optimize resource competition by examining generated code (static analysis rather than heuristic)

4. SPIRAL project (autotuning DSP libraries)
   - Code generation at high level (F77)
   - Search in library tuner, not compiler
VIII. Further Information

- **ATLAS**: math-atlas.sourceforge.net
- **BLAS**: www.netlib.org/blas
- **LAPACK**: www.netlib.org/lapack
- **BLACS**: www.netlib.org/blacs
- **ScaLAPACK**: www.netlib.org/scalapack/scalapack_home.html
- **Publications**: www.cs.utsa.edu/~whaley/papers.html
IX. Common ATLAS Misconception

- If you turn off part of ATLAS’s search to compare with your own code, you must say so, so that readers do not believe ATLAS is slower than it is. You should also report the version of ATLAS (and other math libs) used.

- If you are doing search work, you should be aware that ATLAS’s search does not fail to find large (usually L2) blockings for GEMM performance, rather it avoids them on purpose because large blocking factors can cause dramatic performance losses:
  - http://math-atlas.sourceforge.net/faq.html#NB80

- ATLAS is neither strictly a 1-D line search, nor global:
  - http://math-atlas.sourceforge.net/faq.html#srch_type

- ATLAS’s search is not designed to be fast, but rather to be robust in the face of hardware change (i.e. to require the minimum amount of “model refinement”):
  - http://math-atlas.sourceforge.net/faq.html#srch_speed
III(b). Percent Improvement due to iterative search

Compared against default values:
- SV = 'Yes', WNT = 'No', PF\((\text{inst,dist}) = (nta',2*LS), UR = L_e, AE = 'None'

Percent speedup by transform due to empirical search
III(b)5. iFKO speeds in MFLOPS by platform
II(b) Key Design Decisions

1. *iFKO both iterative and empirical*, as motivated in intro.
2. *Transforms done at low level in backend*, allowing for exploitation of low-level arch features such as SIMD vect & CISC inst formats.
3. *Search is built into the compilation framework*, to ensure the generalization of the search.
4. *We provide for extensive user markup*, to enable key optimizations, and maintain backend focus.
5. *We first concentrate on inner loop*, which is the key weakness in present compilers, and needed for all studied kernels.

⇒ To focus work, start with basic inner loop operations, and add support as required by expanding kernel pool.
II(c). ATLAS Shortcomings

1. Compiler caused problems:
   a. Would often transform perfectly optimized code
   b. Changing compilers changed arch defaults
   c. Could not take advantage of key architectural features such as SIMD vectorization and prefetch
      ● Mult. impl. provides kludgy workaround

2. All empirical optimization kernel specific
   ● ATLAS is helpful for BLAS, but not for even similar ops
   ⇒ Next step was to perform AEOS-style optimization in a compiler (iFKO)
III(f)1. Accumulator Expansion (AE)

Specialized version of scalar expansion employed to avoid unnecessary pipeline stalls due to true dependency.

Unrolled DDOT example before and after AE:

Without AE

```plaintext
dot = start;
for (i=0; i < N; i += 2) {
    dot += X[0] * Y[0];
    dot += X[1] * Y[1];
    X += 2;  Y += 2;
}
```

With AE=2

```plaintext
dot = start; dot1 = 0.0;
for (i=0; i < N; i += 2) {
    dot  += X[0] * Y[0];
    dot1 += X[1] * Y[1];
    X  += 2;  Y  += 2;
}
dot += dot1;
```
III(f)2. Prefetch (PF)

For each array that is legal prefetch target, chooses:

- Prefetch instruction type to employ:
  - prefetchnta
  - prefetcht0
  - prefetcht1
  - prefetcht2
  - prefetchw

- Prefetch distance: how many bytes ahead from present array access to prefetch

- Whether or not it helps to prefetch array

Since prefetches are discarded if bus is busy, all PF inst are crudely scheduled:

- If only one PF inst needed, put at top of loop
- Otherwise, distribute evenly throughout loop
- Other crude schedulings supported, but not presently used
III(d). Supported Architectures

Focus on x86, but design includes many targets so backend is not overly specialized:

1. **IA-32** – AKA: x86, x86-32. Ex.: P4, P4E, Athlon, etc. Initial focus of research (along with x86-64):
   - Most widely used ISA in general purpose computing.
   - ISA has almost no relation to underlying hardware.
   ⇒ Particularly useful target for empirical compilation.

2. **x86-64** – AKA: IA-32e, x86, x86-64. Ex.: Opteron, Athlon-64, P4E (new). Fully supported (not just as using IA-32 compatibility).


4. **UltraSPARC** – Ex.: UltraSPARC II, UltraSPARC III, etc.
ANSI C and HIL Implement of Simple Dot Product

ANSI C:

```c
double ATL_UDOT
    (const int N,
     const double *X, const int incX,
     const double *Y, const int incY)
{
    register double dot=ATL_rzero;
    int i;
    for (i=0; i < N; i++)
        dot += X[i] * Y[i];
    return(dot);
}
```

HIL:

```c
ROUTINE ATL_UDOT;
PARAMS:: N, X, incX, Y, incY;
INT :: N, incX, incY;
DOUBLE_PTR :: X, Y;
ROUT_LOCALS
INT :: i;
DOUBLE :: x, y, dot;
CONST_INIT :: dot=0.0;
ROUT_BEGIN
LOOP i = 0, N
    LOOP_BODY
        x = X[0];
        y = Y[0];
        dot += x * y;
        X += 1;
        Y += 1;
    LOOP_END
RETURN dot;
ROUT_END
```
Basic Linear Algebra Subprograms (BLAS)

- Level 3 – matrix-matrix operations
  - gemm, symm, hemm, syrk, herk, syr2k, her2k, trmm, trsm

- Level 2 – matrix-vector operations
  - gemv, hemv, symv, trmv, trsv
    - ger, geru, gerc, her, her2, syr2

- Level 1 – vector-vector operations
  - swap, scal, copy, axpy, dot, nrm2, asum, iamax

- Packed & banded routines
Level 3 Kernel: On-chip Matmul

- On-chip multiply (fixed dimension, 1 trans case) optimizes entire Level 3 BLAS
- Source generator optimizations include:
  - Loop unrollings (all loops)
  - Register blocking
  - MAC or sep mul/add
  - Software pipelining

ATLAS uses best of:
- General source generator cases
  - Strict ANSI C, general techniques, no system-specific kludges
- Multiple implementation
  - Can be ANSI C or assembler, general or very system-specific

\[ \begin{array}{c}
\begin{array}{c}
M \quad N \\
C_{3,2}
\end{array} \\
\begin{array}{c}
A_{3,1} \quad A_{3,2}
\end{array}
\end{array} \quad \leftarrow M \quad \begin{array}{c}
A \quad K
\end{array} \times \begin{array}{c}
B_{1,2} \quad B_{2,2} \quad B_{3,2}
\end{array} \quad \begin{array}{c}
N \quad K
\end{array} \]

One step of matrix-matrix multiply
### III(c)1. Parameters found for out-of-cache tuning (N=80000)

<table>
<thead>
<tr>
<th>BLAS</th>
<th>SV: WNT</th>
<th>PF X INS:DST</th>
<th>PF Y INS:DST</th>
<th>UR: AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sswap</td>
<td>Y:Y</td>
<td>t0:56</td>
<td>t0:40</td>
<td>4:0</td>
</tr>
<tr>
<td>dswap</td>
<td>Y:Y</td>
<td>t0:128</td>
<td>t0:64</td>
<td>2:0</td>
</tr>
<tr>
<td>scopy</td>
<td>Y:Y</td>
<td>none:0</td>
<td>none:0</td>
<td>2:0</td>
</tr>
<tr>
<td>dcopy</td>
<td>Y:Y</td>
<td>none:0</td>
<td>none:0</td>
<td>1:0</td>
</tr>
<tr>
<td>sasum</td>
<td>Y:N</td>
<td>nta:1024</td>
<td>n/a:0</td>
<td>5:5</td>
</tr>
<tr>
<td>dasum</td>
<td>Y:N</td>
<td>t0:1024</td>
<td>n/a:0</td>
<td>5:5</td>
</tr>
<tr>
<td>saxpy</td>
<td>Y:Y</td>
<td>nta:1408</td>
<td>nta:32</td>
<td>2:0</td>
</tr>
<tr>
<td>daxpy</td>
<td>Y:Y</td>
<td>t0:768</td>
<td>t0:40</td>
<td>2:0</td>
</tr>
<tr>
<td>sdot</td>
<td>Y:N</td>
<td>nta:1024</td>
<td>nta:384</td>
<td>3:3</td>
</tr>
<tr>
<td>ddot</td>
<td>Y:N</td>
<td>nta:768</td>
<td>nta:384</td>
<td>5:5</td>
</tr>
<tr>
<td>sscal</td>
<td>Y:Y</td>
<td>nta:1792</td>
<td>n/a:0</td>
<td>1:0</td>
</tr>
<tr>
<td>dscal</td>
<td>Y:Y</td>
<td>none:0</td>
<td>none:0</td>
<td>2:0</td>
</tr>
<tr>
<td>isamax</td>
<td>N:N</td>
<td>nta:640</td>
<td>n/a:0</td>
<td>8:0</td>
</tr>
<tr>
<td>idamax</td>
<td>N:N</td>
<td>t0:1664</td>
<td>n/a:0</td>
<td>8:0</td>
</tr>
</tbody>
</table>

#### 2.8Ghz Pentium 4E
- Vary strongly by kernel, architecture & context

<table>
<thead>
<tr>
<th>BLAS</th>
<th>SV: WNT</th>
<th>PF X INS:DST</th>
<th>PF Y INS:DST</th>
<th>UR: AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sswap</td>
<td>Y:N</td>
<td>w:1792</td>
<td>w:448</td>
<td>2:0</td>
</tr>
<tr>
<td>dswap</td>
<td>Y:N</td>
<td>nta:960</td>
<td>nta:704</td>
<td>1:0</td>
</tr>
<tr>
<td>scopy</td>
<td>Y:Y</td>
<td>none:0</td>
<td>none:0</td>
<td>1:0</td>
</tr>
<tr>
<td>dcopy</td>
<td>Y:Y</td>
<td>none:0</td>
<td>none:0</td>
<td>1:0</td>
</tr>
<tr>
<td>sasum</td>
<td>Y:N</td>
<td>t0:1664</td>
<td>n/a:0</td>
<td>4:4</td>
</tr>
<tr>
<td>dasum</td>
<td>Y:N</td>
<td>nta:1920</td>
<td>n/a:0</td>
<td>4:4</td>
</tr>
<tr>
<td>saxpy</td>
<td>Y:N</td>
<td>t0:1536</td>
<td>t0:448</td>
<td>4:0</td>
</tr>
<tr>
<td>daxpy</td>
<td>Y:N</td>
<td>nta:1472</td>
<td>t0:832</td>
<td>4:0</td>
</tr>
<tr>
<td>sdot</td>
<td>Y:N</td>
<td>nta:1600</td>
<td>nta:1664</td>
<td>3:3</td>
</tr>
<tr>
<td>ddot</td>
<td>Y:N</td>
<td>t0:1728</td>
<td>t0:704</td>
<td>4:4</td>
</tr>
<tr>
<td>sscal</td>
<td>Y:N</td>
<td>nta:640</td>
<td>n/a:0</td>
<td>1:0</td>
</tr>
<tr>
<td>dscal</td>
<td>Y:N</td>
<td>nta:1344</td>
<td>n/a:0</td>
<td>1:0</td>
</tr>
<tr>
<td>isamax</td>
<td>N:N</td>
<td>nta:768</td>
<td>n/a:0</td>
<td>16:0</td>
</tr>
<tr>
<td>idamax</td>
<td>N:N</td>
<td>nta:1920</td>
<td>n/a:0</td>
<td>32:0</td>
</tr>
</tbody>
</table>

#### 1.6Ghz Opteron
- Vary only weakly by precision
### III(c)2. Parameters found for in-L2-cache tuning (N=1024)

<table>
<thead>
<tr>
<th>BLAS</th>
<th>SV: WNT</th>
<th>PF X</th>
<th>PF Y</th>
<th>UR: AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sswap</td>
<td>Y:N</td>
<td>nta:512</td>
<td>nta:32</td>
<td>16:0</td>
</tr>
<tr>
<td>dswap</td>
<td>Y:N</td>
<td>t0:384</td>
<td>t0:40</td>
<td>32:0</td>
</tr>
<tr>
<td>scopy</td>
<td>Y:N</td>
<td>nta:512</td>
<td>nta:1408</td>
<td>2:0</td>
</tr>
<tr>
<td>dcopy</td>
<td>Y:N</td>
<td>nta:1152</td>
<td>t0:1152</td>
<td>2:0</td>
</tr>
<tr>
<td>sasum</td>
<td>Y:N</td>
<td>t0:1408</td>
<td>n/a:0</td>
<td>16:2</td>
</tr>
<tr>
<td>dasum</td>
<td>Y:N</td>
<td>nta:1792</td>
<td>n/a:0</td>
<td>16:2</td>
</tr>
<tr>
<td>saxpy</td>
<td>Y:N</td>
<td>t0:768</td>
<td>t0:1152</td>
<td>8:0</td>
</tr>
<tr>
<td>daxpy</td>
<td>Y:N</td>
<td>t0:768</td>
<td>t0:384</td>
<td>8:0</td>
</tr>
<tr>
<td>sdot</td>
<td>Y:N</td>
<td>nta:896</td>
<td>nta:1664</td>
<td>64:4</td>
</tr>
<tr>
<td>ddot</td>
<td>Y:N</td>
<td>nta:1280</td>
<td>nta:1792</td>
<td>32:4</td>
</tr>
<tr>
<td>sscal</td>
<td>Y:N</td>
<td>nta:256</td>
<td>n/a:0</td>
<td>2:0</td>
</tr>
<tr>
<td>dscal</td>
<td>Y:N</td>
<td>nta:1536</td>
<td>n/a:0</td>
<td>2:0</td>
</tr>
<tr>
<td>isamax</td>
<td>N:N</td>
<td>t0:1152</td>
<td>n/a:0</td>
<td>32:0</td>
</tr>
<tr>
<td>idamax</td>
<td>N:N</td>
<td>nta:256</td>
<td>n/a:0</td>
<td>32:0</td>
</tr>
</tbody>
</table>

**2.8Ghz Pentium 4E**

<table>
<thead>
<tr>
<th>BLAS</th>
<th>SV: WNT</th>
<th>PF X</th>
<th>PF Y</th>
<th>UR: AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>sswap</td>
<td>Y:N</td>
<td>w:256</td>
<td>w:128</td>
<td>32:0</td>
</tr>
<tr>
<td>dswap</td>
<td>Y:N</td>
<td>w:128</td>
<td>w:128</td>
<td>32:0</td>
</tr>
<tr>
<td>scopy</td>
<td>Y:N</td>
<td>t0:64</td>
<td>none:0</td>
<td>4:0</td>
</tr>
<tr>
<td>dcopy</td>
<td>Y:N</td>
<td>nta:192</td>
<td>none:0</td>
<td>64:0</td>
</tr>
<tr>
<td>sasum</td>
<td>Y:N</td>
<td>nta:64</td>
<td>n/a:0</td>
<td>64:3</td>
</tr>
<tr>
<td>dasum</td>
<td>Y:N</td>
<td>t0:256</td>
<td>n/a:0</td>
<td>4:4</td>
</tr>
<tr>
<td>saxpy</td>
<td>Y:N</td>
<td>nta:128</td>
<td>w:128</td>
<td>4:0</td>
</tr>
<tr>
<td>daxpy</td>
<td>Y:N</td>
<td>nta:32</td>
<td>w:128</td>
<td>4:0</td>
</tr>
<tr>
<td>sdot</td>
<td>Y:N</td>
<td>nta:192</td>
<td>nta:320</td>
<td>16:4</td>
</tr>
<tr>
<td>ddot</td>
<td>Y:N</td>
<td>nta:256</td>
<td>nta:448</td>
<td>6:3</td>
</tr>
<tr>
<td>sscal</td>
<td>Y:N</td>
<td>w:256</td>
<td>n/a:0</td>
<td>32:0</td>
</tr>
<tr>
<td>dscal</td>
<td>Y:N</td>
<td>w:128</td>
<td>n/a:0</td>
<td>4:0</td>
</tr>
<tr>
<td>isamax</td>
<td>N:N</td>
<td>t0:32</td>
<td>n/a:0</td>
<td>16:0</td>
</tr>
<tr>
<td>idamax</td>
<td>N:N</td>
<td>t0:768</td>
<td>n/a:0</td>
<td>32:0</td>
</tr>
</tbody>
</table>

**1.6Ghz Opteron**