Achieving accurate & context-sensitive timing for code optimization

or

How do we measure success for tuning & performance?

R. Clint Whaley
whaley@cs.utsa.edu
www.cs.utsa.edu/∼whaley

Anthony M. Castaldo
castaldo@cs.utsa.edu

Timer Presentation Outline

I(a). Problem Motivation

Why do context-sensitive timings matter?

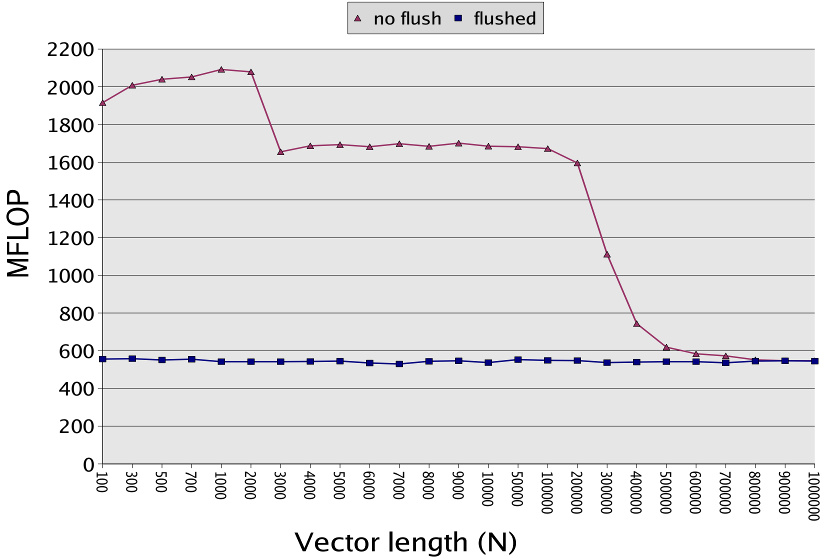
Problem Definition

- Literature contains much discussion of optimizations, little discussion on how to measure transformation results
- Performance of optimization usually measured by home-grown timer
- If timer does not replicate the calling *context* found in target application(s), timer results are often misleading
  - Most important context is probably cache state

Does Lack of Context Sensitivity Matter?

- Changes magnitude of speedup enormously (next slide)
- Changes best parameters for most optimizations
- Changes viability of many optimizations altogether

Performance of DDOT using flushed & non-flushed timers



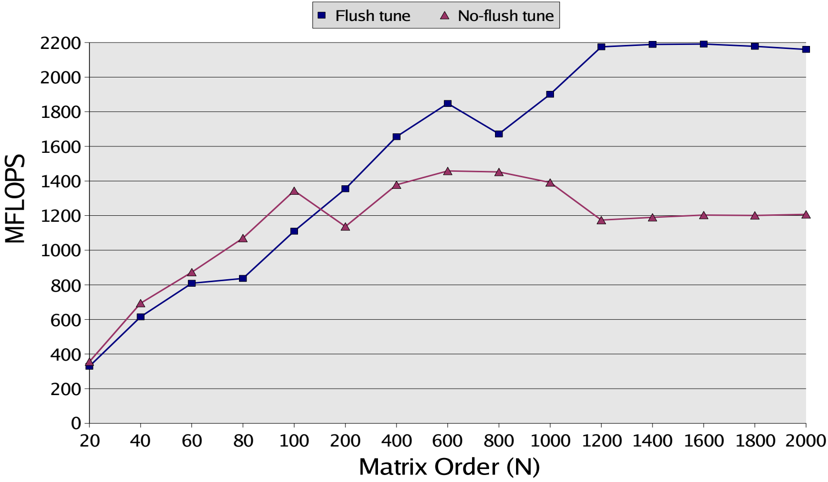
Demonstrated strong effect of in- vs. out-of-cache timing on all considered optimizations in:

- R. Clint Whaley and David B. Whalley, "Tuning High Performance Kernels through Empirical Compilation", In *The 2005 International Conference on Parallel Processing*, June 2005.

Less formally, consider:

- Optimizations like: load/use pipelining, data prefetch, tiling
  - $\Rightarrow$ All may show no benefit, slowdown, get wrong param value when timed in-cache, but used out-of-cache
  - $\longrightarrow$ All may be **critical** for out-of-cache performance
- Does this actually occur (yes, next slide)?

ATLAS DGEMM performance when installed with/without flushing



II. Naïve Kernel and Timer

DDOT kernel

```
double dotprod(
   const int N,
   const double *X,
   const double *Y)
{
   int i;
   double dot=0.0;
   for (i=0; i<N; i++)
      dot += X[i] * Y[i];
   return(dot);
}
```

Naïve timer

```
   for (i=0; i < N; i++)
   {        // Init operands
      X[i] = rand();
      Y[i] = rand();
   }
//
// Perform timing
//
   t0 = my_time();
   dot = dotprod(N, X, Y);
   t1 = my_time();
```

- Init preloads operands to any cache large enough to hold them

LRU-based cache flush

```
dsz = sizeof(double)
cs = cacheKB*1024/dsz;
flush = calloc(cs,dsz);
for (i=0; i < N; i++) {
   X[i] = rand(); // Init
   Y[i] = rand();
}
for (i=0; i < cs; i++)
   tmp += flush[i]; //flsh
assert(tmp < 10.0);
t0 = my_time();
dot = dotprod(N, X, Y);
t1 = my_time();
```

OneCallFlushLRU Notes

⇒ Access unrelated area $\geq$ cache size to force flush

- Relies on LRU for flush
  - → For non-LRU caches, increase `cacheKB`
- Vary flush level wt `cacheKB`
- Allow specific ops in-cache by initing after flush
- Paper has x86-specific method using explicit cache-flush instructions

IV(a). Cache Flushing for Multiple Calls

Timing when each measurement contains multiple kernel invokations

When kernel call below repeatable clock resolution, can time loop that invokes kernel *nrep* times to get timing interval above resolution:

- Cannot start & stop timers inside loop
  - $\longrightarrow$ each interval below resolution, so timing mostly error
    - $\longrightarrow$ adding them up gives erroneous time

$\Rightarrow$ Must start timer before *nrep* loop, stop after:

- If you call with same operands, will be in-cache
- If you use prior technique, last $nrep - 1$ calls in-cache
- If you put flush inside loop, flush time added to kernel time
  - $\longrightarrow$ Cannot time flush only loop and subtract, since flush time may vary strongly depending on external access

$\Rightarrow$ Must lay out operands in mem, and move so that each kernel invocation uses out-of-cache data (next slide)

bad idea1 - no resolution

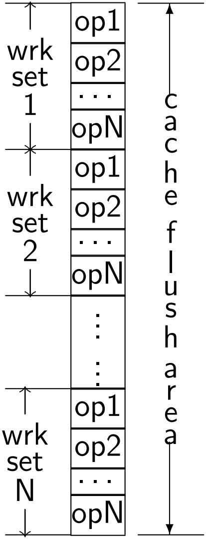
```
dsz = sizeof(double)
flush = calloc(cs,dsz);
for (j=0; j < nrep; j++)
{
   for (i=0; i < cs; i++)
      tmp += flush[i];
   assert(tmp < 10.0);
   t0 = my_time();
   dot = dotprod(N, X, Y);
   t1 += my_time() - t0;
}
```

bad idea2 - flush prob

```
t0 = my_time();
for (j=0; j < nrep; j++) {
   for (i=0; i < cs; i++)
      tmp += flush[i];
   assert(tmp < 10.0);
   dot = dotprod(N, X, Y);
}
t1 = my_time() - t0;
t0 = my_time();
for (j=0; j < nrep; j++) {
   for (i=0; i < cs; i++)
      tmp += flush[i];
}
t1 -= my_time() - t0;
```

IV(c). Cache Flushing for Multiple Calls

Cache flushing when kernel called multiple times in one sample



Multiple call dot product timer

```
cs = cacheKB*(1024/sizeof(double));
setsz = N + N; // 2 N-length ops in wrk set
nset = (cs + setsz-1)/setsz;
if (nset < 1) nset=1;
Nt = nset * setsz;
X = vp = malloc(sizeof(double)*Nt);
X += Nt - setsz; Y = X + N;
for (x=vp,i=Nt-1; i >= 0; i--)
   x[i] = my_drand();
x=X; y=Y; k=0; alpha = 1.0;
t0 = my_time();
for (i=0; i < nrep; i++) {
   dot += alpha*dotprod(N, X, Y)
   if (++k != nset) {x -= setsz; y -= setsz;}
   else {x=X;y=Y;k=0;alpha = -alpha;}
}
time = (my_time()-t0)/((double)nrep);
```

V. Timer Refinements

List of additional timing techniques/tips covered in paper

Paper provides techniques for avoiding:

- Floating point over/under-flow,
- Lazy page zeroing,
- Virtual memory instruction load,
- Incorrect timings due to CPU throttling.

Paper discusses methods for:

- Choosing best system timer
- Getting more repeatable results using both CPU and WALL timers,
- Varying type and thoroughness of flush,
- Enforcing memory (mis)alignment,
- Adapting cache flushing for parallel timings.

VI. Further Information

- **Presenter homepage**: `www.cs.utsa.edu/~whaley/`
- **Timing paper**: Clint Whaley and Anthony M. Castaldo, "Achieving accurate and context-sensitive timing for code optimization", accepted for publication in *Software: Practice & Experience*

  → `www.cs.utsa.edu/~whaley/papers/timing_SPE08.pdf`
- **ATLAS homepage**: `math-atlas.sourceforge.net`
- **iFKO paper**: R. Clint Whaley and David B. Whalley, "Tuning High Performance Kernels through Empirical Compilation", In *The 2005 International Conference on Parallel Processing*, June 2005.

  → `www.cs.utsa.edu/~whaley/papers/icpp05_8.ps`

Picking losing fights

Self-tuned libraries will always outperform compiler-generated code

$\Rightarrow$ **In theory, no, in practice, yes.**

$\longrightarrow$ Probably will compete on selected benchmarks, but be crushed for actual use.

Picking losing fights

Self-tuned libraries will always outperform compiler-generated code

⇒ **In theory, no, in practice, yes.**

→ Probably will compete on selected benchmarks, but be crushed for actual use.

Why: Three anti-HPC Compiler Traditions

1. My assumptions trump your experimental results
   - Libraries eventually have users wt. applications
     → keeps them honest to some degree
2. All problems solved 20 years ago → nothing works today
   - HPC weak, but does reward raw performance improvement
   - We haven't solved this prob in serial:
     ⇒ Let's solve it on heterogeneous massively parallel machine!
3. 10,000 front-ends, 0 HPC backends
   - CISC compaction, front-end (arch) optimization, inst alignment, inst selection & sched