Optimizing for Speed

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se

What is the potential gain?
- Latency difference L1$ and mem: ~50x
- Bandwidth difference L1$ and mem: ~20x
- Repeated TLB misses adds a factor ~2-3x
- Execute from L1$ instead from mem => 50-150x improvement
- At least a factor 2-4x is within reach

Optimizing for cache performance
- Keep the active footprint small
- Use the entire cache line once it has been brought into the cache
- Fetch a cache line prior to its usage
- Let the CPU that already has the data in its cache do the job
- ...

What can go Wrong?
A Simple Example...
Perform a diagonal copy 10 times
**Example: Loop order**

//Optimized Example A
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A[i][j] = A[i-1][j-1];
    }
}

//Unoptimized Example A
for (j=1; j<N; j++) {
    for (i=1; i<N; i++) {
        A[i][j] = A[i-1][j-1];
    }
}

**Performance Difference: Loop order**

Performance Difference:

Array side: 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

Speedup vs UnOpt:

- Athlon64 x2
- Pentium D
- Core 2 Duo

**Example: Sparse data**

//Optimized Example A
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A_data[i][j] = A_data[i-1][j-1];
    }
}

//Unoptimized Example A
for (i=1; i<N; i++) {
    for (j=1; j<N; j++) {
        A[i][j].data = A[i-1][j-1].data;
    }
}

**Performance Difference: Sparse Data**

Performance Difference:

Array side: 16, 32, 64, 128, 256, 512, 1024, 2048, 4096

Speedup vs UnOpt:

- Athlon64 x2
- Pentium D
- Core 2 Duo
Loop Merging

/* Unoptimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        c[i][j] = K * b[i][j] + d[i][j]/2;

/* Optimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        a[i][j] = 2 * b[i][j];
    c[i][j] = K * b[i][j] + d[i][j]/2;

Padding of data structures

Padding of data structures

allocate more memory than needed

Blocking

/* Unoptimized ARRAY: x = y * z */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        r = 0;
        for (k = 0; k < N; k = k + 1)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;

/* Optimized */
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        r = 0;
        for (k = 0; k < N; k = k + 1)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;

X: j
Y: k
Z: j
**Blocking**

/* Optimized ARRAY: X = Y * Z */
for (jj = 0; jj < N; jj = jj + B)
for (i = 0; i < N; i = i + 1)
for (j = jj; j < min(jj+B,N); j = j + 1)
{r = 0;
for (k = kk; k < min(kk+B,N); k = k + 1)
r = r + y[i][k] * z[k][j];
x[i][j] += r;}

**Blocking: the Movie!**

/* Optimized ARRAY: X = Y * Z */
for (jj = 0; jj < N; jj = jj + B)
for (i = 0; i < N; i = i + 1)
for (j = jj; j < min(jj+B,N); j = j + 1)
{r = 0;
for (k = kk; k < min(kk+B,N); k = k + 1)
r = r + y[i][k] * z[k][j];
x[i][j] += r;}

**Prefetching**

/* Unoptimized */
for (j = 0; j < N; j++)
for (i = 0; i < N; i++)
x[j][i] = 2 * x[j][i];

/* Optimized */
for (j = 0; j < N; j++)
for (i = 0; i < N; i++)
PREFETCH x[j+1][i]
x[j][i] = 2 * x[j][i];

(Typically, the HW prefetcher will successfully prefetch sequential streams)

**Cache Affinity**

- Schedule the process on the processor it last ran
- Allocate and free data buffers in a LIFO order
Optimize for "other caches"

- TLB
  - Avoid random accesses to huge data structs (Ex. Huge hashing table)
  - Avoid few access per page (very sparse data)

Commercial Break:
Acumem’s Multicore Tools

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se

Acumem SlowSpotter™

Source: C, C++, Fortran, OpenMP...

Mission:
Find the SlowSpots™
Asses their importance
Enable for non-experts to fix them
Improve the productivity of performance experts

Help!

What? Where? How?

Any Compiler
Sampler
Binary

How?

Advice
Analysis
Target System Parameters

Any Compiler
Sampler
Binary
Finger Print (~4MB)

Mission:
Find the SlowSpots™
Asses their importance
Enable for non-experts to fix them
Improve the productivity of performance experts

Any Compiler
Sampler
Binary
Finger Print (~4MB)
A One-Click Report Generation

Fill in the following fields:
- Application to run
- Input arguments
- Working dir (where to run the app)
  (Limit, if you like, data gathered here, e.g., start gathering after after 10 sec. and stop after 10 sec.)
- Cache size of the target system for optimization (e.g., L1 or L2 size)

Click this button to create a report

Predicted fetch rate
(if utilization \( \rightarrow 100\% \))

Cache size

Miss rate

Cache utilization = Fraction of cache data utilized

Optimized fetch rate

Loop Focus Tab

Spotting the crime

List of bad loops

Explaining what to do
**Resource Sharing Example**

**Libquantum**
A quantum computer simulation
widely used in research (download from: [http://www.libquantum.de/](http://www.libquantum.de/))
4000+ lines of C, fairly complex code.
Runs an experiment in ~30 min

**Utilization Analysis**

**Libquantum**

**Original Code**

```c
for (i=0; i++; i<MAX) {
    ... = huge_data[i].status + ...;
}
```

**Utilization Optimization**

```c
for (i=0; i++; i<MAX) {
    ... = huge_data_status[i] + ...;
}
```

**SlowSpotter’s First Advice: Improve Utilization**
- Change one data structure
- Involves ~20 lines of code
- Takes a non-expert 30 min

**Throughput improvement:**

<table>
<thead>
<tr>
<th>Number of Cores Used</th>
<th>Relative Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

**Predicted fetch rate if utilization = 100%**
- Cache utilization = Fraction of cache data utilized
- Cache size
- Need 32 MB per thread!

**Utilization Optimization**

```c
for (i=0; i++; i<MAX) {
    ... = huge_data_status[i] + ...;
}
```
After Utilization Optimization

Libquantum

Old fetch rate

Original Code

Utilization Optimization

Cache Utilization = 95%

Cache size

Predicted fetch rate = New fetch rate

OPT 29

Utilization Optimization

Original Code

Utilization Optimization

Cache Utilization = 95%

Cache size

Predicted fetch rate = New fetch rate

Two positive effects from better utilization
1. Each fetch brings in more useful data → lower fetch rate
2. The same amount of useful data can fit in a smaller cache → shift left

OPT 30

Reuse Analysis

Libquantum

Utilization Optimization

Utilization + Fusion Optimization

Second-Fifth SlowSpotter Advice: Improve reuse of data
Fuse functions traversing the same data
- Here: four fused functions created
- Takes a non-expert < 2h

SPEC CPU2006-462.libquantum

Old fetch rate

New fetch rate

Utilization Optimization

Utilization + Fusion Optimization

Effect: Reuse Optimization

The miss in the second loop goes away
- Still need the same amount of cache to fit “all data”
Utilization + Reuse Optimization

- Fetch rate down to 1.3% for 2MB
- Same as a 32 MB cache originally

Summary

Libquantum

- Original
- Utilization Optimization
- Utilization + Fusion

Utilization Optimization

New fetch rate

Utilization + Fusion Optimization

Old fetch rate

Fetch rate down to 1.3% for 2MB

Same as a 32 MB cache originally

# Cores Used

Throughput

0 1 2 3 4

Original

Utilization Optimization

Utilization + Fusion

2.7x

Optimization

Optimization

Edit-compile-analysis cycle ≈ 1min

Original Cigar Throughput

Performance

0 1 2 3 4

# Cores

Throughput scalability is a different way to look at the performance of an application. Here, several single-threaded instances of the application is run at the same time. Even though the different instances do not explicitly depend on each other, they will nevertheless fight over the shared resources, e.g., running four threads on four cores implies that each thread will get one quarter of the shared cache. A system using four cores to run four instances of Cigar will actually result in a lower throughput than if only three cores were used.
The optimization puts a much lower pressure on the shared cache resulting in a 33x better throughput for four cores.

AMD's new six-core Istanbul processor can enjoy a 17x better throughput due to the optimization on six cores.

Intel's new four-core i7 (Nehalem) processor enjoy a 13x better throughput due to the Optimization on four cores. Note that each core can run up to two threads.

Cache sharing issues

Erik Hagersten
Uppsala University, Sweden
eh@it.uu.se
Fighting for shared resources

1st Order MC Performance Problems

- Additional multicore issues:
  - Even less cache resources per application
  - Sharing of cache resources
  - Wasted cache usage

Example: Hints to avoid cache pollution (non-temporal prefetches)

- The larger cache, the better
- Hint: Don’t allocate!

Example: Hints for mixed workloads (non-temporal prefetches)

- "bigger is better"
- "streaming"

Some performance tools

Free licenses
- Oprofile
- GNU: gprof
- AMD: code analyst
- Google performance tools
- Virtual Inst: High Productivity Supercomputing (http://www.vi-hps.org/tools/)
- Sun Studio ...

Not free
- Intel: Vtune and many more
- Alinea, TotalView,... (for MPI...)
- Acumem (of course)
- HP: Multicore toolkit (some free, some not)