Microbenchmarking for architectural exploration

Probing of the memory hierarchy
Saturation effects in cache and memory
Typical OpenMP overheads
Motivation for Microbenchmarking as a tool

- Isolate small kernels to:
  - Separate influences
  - Determine specific machine capabilities (light speed)
  - Gain experience about software/hardware interaction
  - Determine programming model overhead
  - …

- Possibilities:
  - Readymade benchmark collections (epcc OpenMP, IMB)
  - STREAM benchmark for memory bandwidth
  - Implement own benchmarks (difficult and error prone)
  - **likwid-bench** tool: Offers collection of benchmarks and framework for rapid development of assembly code kernels
Latency and bandwidth in modern computer environments

Avoiding slow data paths is the key to most performance optimizations!
Recap: Data transfers in a memory hierarchy

- How does data travel from memory to the CPU and back?
- Example: Array copy \( A(:) = C(:) \)

![Diagram showing data transfers](image)

**Standard stores**

- 3 CL transfers

- **Nontemporal (NT) stores**
- 2 CL transfers
- 50% performance boost for COPY
The parallel vector triad benchmark
A “swiss army knife” for microbenchmarking

Simple streaming benchmark:

\[
\text{double precision, dimension}(N) :: A,B,C,D
\]
\[
A=1.d0; \quad B=A; \quad C=A; \quad D=A
\]

\[
do \; j=1,NITER
\quad do \; i=1,N
\quad \quad A(i) = B(i) + C(i) \times D(i)
\quad enddo
\quad if(.something.that.is.never.true.) \; \text{then}
\quad \quad call \; \text{dummy}(A,B,C,D)
\quad endif
\quad enddo
\]

- Report performance for different N
- Choose NITER so that accurate time measurement is possible
- This kernel is limited by data transfer performance for all memory levels on all current architectures!

Prevents smarty-pants compilers from doing “clever” stuff
A(:) = B(:) + C(:) * D(:) on one Sandy Bridge core (3 GHz)

Are the performance levels plausible?

What about multiple cores?

Do the bandwidths scale?

4 W / iteration $\rightarrow$ 128 GB/s

5 W / it. $\rightarrow$ 18 GB/s (incl. write allocate)
Throughput capabilities of the Intel Sandy Bridge

- **Per cycle with AVX**
  - 1 load instruction (256 bits) **AND** ½ store instruction (128 bits)
  - 1 AVX MULT and 1 AVX ADD instruction (4 DP / 8 SP flops each)
  - Overall maximum of 4 micro-ops

- **Per cycle with SSE or scalar**
  - 2 load instruction **OR** 1 load and 1 store instruction
  - 1 MULT and 1 ADD instruction
  - Overall maximum of 4 micro-ops

- **Data transfer between cache levels (L3 ↔ L2, L2 ↔ L1)**
  - 256 bits per cycle, half-duplex (i.e., full CL transfer == 2 cy)
A(:, :) = B(:, :) + C(:, :) * D(:, :) on one Sandy Bridge core (3 GHz)

Max. LD/ST throughput:
1 AVX Load & ½ AVX Store per cycle
→ 3 cy / 8 Flops ↔ 8 Flops/3 cy

(2 LD or 1 LD & 1 ST) / cy
→ 2 Flops/2 cy

See later for more on SIMD benefits

Theoretical limit

4 W / iteration → 128 GB/s

4 W / iteration → 48 GB/s

2.66x SIMD impact

Data far away → smaller SIMD impact

AVX

scalar

Microbenchmarking
A riddle with huge consequences

Save for later!

L2/L3 theoretical limit

MEM limit

(c) RRZE 2013
The throughput-parallel vector triad benchmark

Every core runs its own, independent triad benchmark

double precision, dimension(:), allocatable :: A,B,C,D

!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
do j=1,NITER
   do i=1,N
      A(i) = B(i) + C(i) * D(i)
   enddo
   if(.something.that.is.never.true.) then
      call dummy(A,B,C,D)
   endif
enddo
!$OMP END PARALLEL

→ pure hardware probing, no impact from OpenMP overhead
Throughput vector triad on Sandy Bridge socket (3 GHz)

Saturation effect in memory

Scalable BW in L1, L2, L3 cache
Bandwidth limitations: **Main Memory**

Scalability of shared data paths *inside a NUMA domain* (V-Triad)

- **Saturation with 3 threads**
- **Saturation with 2 threads**
- **1 thread cannot saturate bandwidth**
- **Saturation with 4 threads**

Graph showing bandwidth in GBytes/s against the number of cores.

- **Westmere**
- **Sandy Bridge**
- **Interlagos**

1 NUMA domain vs 2 NUMA domains.
Attainable memory bandwidth: Comparing architectures

Intel Sandy Bridge

AMD Interlagos

ECC=on

Intel Xeon Phi 5110P

2-socket CPU node

ECC=on

NVIDIA K20

(c) RRZE 2013
Bandwidth limitations: **Outer-level cache**

**Scalability of shared data paths in L3 cache**

![Graph showing bandwidth limitations and scalability of shared data paths in L3 cache.](image)

- **Intel SB:** New scalable L3 design
- **AMD:** Optimize for L2 cache!
Consequences from the saturation pattern

- Clearly distinguish between “saturating” and “scalable” performance on the chip level

![Graphs showing performance vs. cores for saturating and scalable types](image-url)
Consequences from the saturation pattern

- There is no clear bottleneck for single-core execution
- Code profile for single thread ≠ code profile for multiple threads
  - → Single-threaded profiling may be misleading
The OpenMP-parallel vector triad benchmark

OpenMP work sharing in the benchmark loop

double precision, dimension(:,), allocatable :: A,B,C,D

allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
 !$OMP PARALLEL private(i,j)
do j=1,NITER
 !$OMP DO
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
 !$OMP END DO
if(.something.that.is.never.true.) then
  call dummy(A,B,C,D)
endif
enddo
 !$OMP END PARALLEL
OpenMP vector triad on Sandy Bridge socket (3 GHz)

- **Performance [GFlop/s]**
- **Loop length**
- **Sync overhead grows with # of threads**
- **L1 core limit**
- **Bandwidth scalability across memory interfaces**

- T=1
- T=8 (1 socket)
- T=16 (2 sockets)
OpenMP performance issues on multicore

Synchronization (barrier) overhead
Welcome to the multi-/many-core era

*Synchronization of threads may be expensive!*

```plaintext
$OMP PARALLEL ...
...
$OMP BARRIER
$OMP DO
...
$OMP END DO
$OMP END PARALLEL
```

Threads are synchronized at **explicit** AND **implicit** barriers. These are a main source of overhead in OpenMP programs.

Determine costs via modified OpenMP Microbenchmarks testcase (epcc)

---

On x86 systems there is no hardware support for synchronization!

- Next slide: Test **OpenMP** Barrier performance...
- for different compilers
- and different topologies:
  - shared cache
  - shared socket
  - between sockets
- and different thread counts
  - 2 threads
  - full domain (chip, socket, node)
### Thread synchronization overhead on SandyBridge-EP

**Barrier overhead in CPU cycles**

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>384</td>
<td>5242</td>
<td>4616</td>
</tr>
<tr>
<td>SMT threads</td>
<td>2509</td>
<td>3726</td>
<td>3399</td>
</tr>
<tr>
<td>Other socket</td>
<td>1375</td>
<td>5959</td>
<td>4909</td>
</tr>
</tbody>
</table>

GCC still not very competitive

---

<table>
<thead>
<tr>
<th>Full domain</th>
<th>Intel 13.1.0</th>
<th>GCC 4.7.0</th>
<th>GCC 4.6.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>1497</td>
<td>14546</td>
<td>14418</td>
</tr>
<tr>
<td>Node</td>
<td>3401</td>
<td>34667</td>
<td>29788</td>
</tr>
<tr>
<td>Node +SMT</td>
<td>6881</td>
<td>59038</td>
<td>58898</td>
</tr>
</tbody>
</table>

Intel compiler 🎉
# Thread Synchronization Overhead on Intel Xeon Phi

## Barrier Overhead in CPU Cycles

<table>
<thead>
<tr>
<th></th>
<th>SMT1</th>
<th>SMT2</th>
<th>SMT3</th>
<th>SMT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>One core</td>
<td>n/a</td>
<td>1597</td>
<td>2825</td>
<td>3557</td>
</tr>
<tr>
<td>Full chip</td>
<td>10604</td>
<td>12800</td>
<td>15573</td>
<td>18490</td>
</tr>
</tbody>
</table>

That does not look bad for 240 threads!

Still the pain may be much larger, as more work can be done in one cycle on Phi compared to a full Sandy Bridge node:

- **3.75 x cores (16 vs 60) on Phi**
- **2 x more operations per cycle on Phi**
- **2.7 x more barrier penalty (cycles) on Phi**

7.5 x more work done on Xeon Phi per cycle

One barrier causes 2.7 x 7.5 = 20x more pain 😞.
likwid-bench ...

1. is an extensible, flexible benchmarking framework
2. allows rapid development of low level kernels
3. already includes many ready to use threaded benchmark kernels

- **Benchmarking runtime cares for:**
  - Thread management and placement
  - Data allocation and NUMA aware initialization
  - Timing and result presentation

- **likwid-bench focuses on assembly code interface and therefore keeps out programming model or compiler issues**
likwid-bench Example

- Implement micro benchmark in abstract assembly
- The benchmark file is automatically converted, compiled and added to the benchmark application
- Benchmark files are located in the ./bench directory

```
$ likwid-bench -t clcopy -g 1 -i 1000 -w S0:1MB:2

$ likwid-bench -t load -g 2 -i 100 -w S1:1GB -w S0:1GB-0:S1,1:S0
```

<table>
<thead>
<tr>
<th>STREAMS 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE DOUBLE</td>
</tr>
<tr>
<td>FLOPS 0</td>
</tr>
<tr>
<td>BYTES 16</td>
</tr>
<tr>
<td>LOOP 32</td>
</tr>
<tr>
<td>movaps FPR1, [STR0 + GPR1 * 8 ]</td>
</tr>
<tr>
<td>movaps FPR2, [STR0 + GPR1 * 8 + 64 ]</td>
</tr>
<tr>
<td>movaps FPR3, [STR0 + GPR1 * 8 + 128 ]</td>
</tr>
<tr>
<td>movaps FPR4, [STR0 + GPR1 * 8 + 192 ]</td>
</tr>
<tr>
<td>movaps [STR1 + GPR1 * 8 ], FPR1</td>
</tr>
<tr>
<td>movaps [STR1 + GPR1 * 8 + 64 ], FPR2</td>
</tr>
<tr>
<td>movaps [STR1 + GPR1 * 8 + 128 ], FPR3</td>
</tr>
<tr>
<td>movaps [STR1 + GPR1 * 8 + 192 ], FPR4</td>
</tr>
</tbody>
</table>

Data streams used in benchmark

Flops performed and bytes transferred in one operation

Iterations performed in one loop iteration

(c) RRZE 2013

Microbenchmarking
likwid-bench command line syntax

likwid-bench -h
likwid-bench -a  list available benchmarks

Required options:

likwid-bench -t copy -g 1 -w S1:1GB
-<benchmark case>
-g <# thread groups>  need equivalent # working groups
-w <thread domain>:<working set size (kB, MB or GB)>
(-i <# iterations> adjust to get reasonable runtime)

Specify number of threads (Default: all processors in thread domain):

likwid-bench -t copy -g 1 -w S1:1GB:2

Specify data placement (Default: in same NUMA domain as threads):

likwid-bench -t copy -g 1 -w S1:1GB:2-0:S0,1:S1

Syntax similar to likwid-pin expression based syntax

(c) RRZE 2013  Microbenchmarking
Conclusions from the microbenchmarks

- **Affinity matters!**
  - Almost all performance properties depend on the position of
    - Data
    - Threads/processes
  - Consequences
    - Know where your threads are running
    - Know where your data is

- **Bandwidth bottlenecks are ubiquitous**

- **Synchronization overhead may be an issue**
  - … and also depends on affinity!
  - Many-core poses new challenges in terms of synchronization