Performance-oriented programming on multicore-based systems, with a focus on the Cray XE6/ XC30

Georg Hager\(^{(a)}\), Jan Treibig\(^{(a)}\), and Gerhard Wellein\(^{(a,b)}\)

\(^{(a)}\)HPC Services, Erlangen Regional Computing Center (RRZE)
\(^{(b)}\)Department for Computer Science
    Friedrich-Alexander-University Erlangen-Nuremberg

Cray XE6 optimization workshop, March 17-20, 2014, HLRS
The Rules™

There is no alternative to knowing what is going on between your code and the hardware

Without performance modeling, optimizing code is like stumbling in the dark
Agenda

- Basics of multicore processor and node architecture
- Multicore performance and tools
  - Affinity enforcement
  - Performance counter measurements
  - Basics and best practice for performance counter profiling
- Microbenchmarking for architectural exploration
- Roadblocks for scalability on multicore chips
  - Scaling properties and typical OpenMP overhead
  - Bandwidth saturation in cache and main memory
- Simple Performance Modeling: The Roofline model
- Optimal utilization of parallel resources
  - Programming for SIMD parallelism
  - Programming in ccNUMA environments
- Case study: The roofline model for a 3D Jacobi solver
  - Understanding performance characteristics
  - Model-guided optimization
- Case study: sparse matrix-vector multiplication
  - What we can do if the roofline model “does not work”
Multicore processor and system architecture – an overview

Performance composition
Memory organization: UMA vs. ccNUMA
Simultaneous Multi-Threading (SMT)
Data paths in HPC systems
Memory access
Single Instruction Multiple Data (SIMD)
Topology and programming models
There is no longer a single driving force for chip performance!

Floating Point (FP) Performance:

\[ P = n_{\text{core}} \times F \times S \times \nu \]

- **\( n_{\text{core}} \)**: number of cores: 8
- **\( F \)**: FP instructions per cycle: 2 (1 MULT and 1 ADD)
- **\( S \)**: FP ops / instruction: 4 (dp) / 8 (sp) (256 Bit SIMD registers – “AVX”)
- **\( \nu \)**: Clock speed: 2.5 GHz

\[ P = 160 \text{ GF/s (dp)} / 320 \text{ GF/s (sp)} \]

But: \( P=5 \text{ GF/s (dp)} \) for serial, non-SIMD code
Today: Dual-socket Intel (Westmere) node:

- **Cache-coherent Non-Uniform Memory Architecture (ccNUMA)**
  - HT / QPI provide scalable bandwidth at the price of ccNUMA architectures:
    - Where does my data finally end up?

On AMD it is even more complicated → ccNUMA within a socket!
Back to the 2-chip-per-case age
16 core AMD Interlagos – a 2x8-core ccNUMA socket

- **AMD: single-socket ccNUMA since Magny Cours**
  - 1 socket: 16-core Interlagos built from two 8-core chips → 2 NUMA domains
  - 2 socket server → 4 NUMA domains
  - 4 socket server: → 8 NUMA domains

- **WHY?** → Shared resources are hard to scale:
  - \(2 \times 2\) memory channels vs. \(1 \times 4\) memory channels per socket
Cray XE6 (Hermit) “Interlagos” 16-core dual socket node

- Two 8- (integer-) core chips per socket @ 2.3 GHz (3.3 @ turbo)
- Separate DDR3 memory interface per chip
  - ccNUMA on the socket!
- Shared FP unit per pair of integer cores ("module")
  - 2 128bit FMA FP units
  - SSE4.2, AVX, FMA4
- 16 kB L1 data cache per core
- 2 MB L2 cache per module
- 8 MB L3 cache per chip (6 MB usable)
SMT Makes a single physical core appear as two or more “logical” cores → multiple threads/processes run concurrently

- SMT principle (2-way example):
Another flavor of “SMT”
AMD Interlagos / Bulldozer

- Up to 16 cores (8 Bulldozer modules) in a single socket
- Max. 2.6 GHz (+ Turbo Core)
- $P_{\text{max}} = (2.6 \times 8 \times 8) \text{ GF/s} = 166.4 \text{ GF/s}$

Each Bulldozer module:
- 2 “lightweight” cores
- 1 FPU: 4 MULT & 4 ADD (double precision) / cycle
- Supports AVX
- Supports FMA4

2 DDR3 (shared) memory channel > 15 GB/s

2 NUMA domains per socket
Cray XC30 “SandyBridge-EP” 8-core dual socket node

- 8 cores per socket 2.7 GHz (3.5 @ turbo)
- DDR3 memory interface with 4 channels per chip
- Two-way SMT
- Two 256-bit SIMD FP units
  - SSE4.2, AVX
- 32 kB L1 data cache per core
- 256 kB L2 cache per core
- 20 MB L3 cache per chip

Cray Workshop

Performance for Multicore
Latency and bandwidth in modern computer environments

We care about this region today

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

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

---

**Standard stores**

LD \( C(1) \)  
MISS

ST \( A(1) \)  
MISS

LD \( C(2..N_{cl}) \)  
HIT

ST \( A(2..N_{cl}) \)  
HIT

3 CL transfers

**Nontemporal (NT) stores**

LD \( C(1) \)  
MISS

NTST \( A(1) \)

LD \( C(2..N_{cl}) \)  
HIT

NTST \( A(2..N_{cl}) \)

2 CL transfers

50% performance boost for COPY
SIMD-processing – Basics

- Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on “wide” registers.
- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
- Adding two registers holding double precision floating point operands

```
R0  R1  R2
A[0] B[0] C[0]
64 Bit
```

```
R0  R1  R2
256 Bit
```

SIMD execution:
V64ADD [R0,R1] → R2

Scalar execution:
R2 ← ADD [R0,R1]
Challenges of modern compute nodes

Heterogeneous programming
SIMD + OpenMP + MPI + CUDA, OpenCL,…

Core:
- SIMD vectorization
- SMT

Socket:
- Parallelization
- Shared Resources

Node:
- ccNUMA/data locality

Accelerators:
- Data transfer to/from host
Parallelism in modern computer systems

- Parallel and shared resources within a shared-memory node

**Parallel resources:**
- Execution/SIMD units
- Cores
- Inner cache levels
- Sockets / memory domains
- Multiple accelerators

**Shared resources:**
- Outer cache level per socket
- Memory bus per socket
- Intersocket link
- PCIe bus(es)
- Other I/O resources

How does your application react to all of those details?
Parallel programming models on multicore multisocket nodes

- **Shared-memory (intra-node)**
  - Good old MPI (current standard: 2.2)
  - OpenMP (current standard: 3.0)
  - POSIX threads
  - Intel Threading Building Blocks
  - Cilk++, OpenCL, StarSs,… you name it

- **Distributed-memory (inter-node)**
  - MPI (current standard: 2.2)
  - PVM (gone)

- **Hybrid**
  - Pure MPI
  - MPI+OpenMP
  - MPI + any shared-memory model

All models require awareness of topology and affinity issues for getting best performance out of the machine!
Parallel programming models:

Pure MPI

- Machine structure is invisible to user:
  - → Very simple programming model
  - → MPI “knows what to do”!

- Performance issues
  - Intranode vs. internode MPI
  - Node/system topology
Parallel programming models:  
_Pure threading on the node_

- **Machine structure is invisible to user**
  - \( \rightarrow \) Very simple programming model
  - Threading SW (OpenMP, pthreads, TBB,…) should know about the details

- **Performance issues**
  - Synchronization overhead
  - Memory access
  - Node topology
Multicore Performance and Tools

Probing node topology

- Standard tools
- likwid-topology
How do we figure out the node topology?

**Topology**
- Where in the machine does core \#n reside? And do I have to remember this awkward numbering anyway?
- Which cores share which cache levels?
- Which hardware threads ("logical cores") share a physical core?

**Linux**
- `cat /proc/cpuinfo` is of limited use
- Core numbers may change across kernels and BIOSes even on identical hardware
- `numactl --hardware` prints ccNUMA node information

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 8189 MB
node 0 free: 3824 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 8192 MB
node 1 free: 28 MB
node 2 cpus: 18 19 20 21 22 23
node 2 size: 8192 MB
node 2 free: 8036 MB
node 3 cpus: 12 13 14 15 16 17
node 3 size: 8192 MB
node 3 free: 7840 MB
```
How do we figure out the node topology?

- **LIKWID** tool suite:

  Like
  I
  Knew
  What
  I’m
  Doing

- Open source tool collection (developed at RRZE):

  [http://code.google.com/p/likwid](http://code.google.com/p/likwid)


Likwid Tool Suite

- **Command line tools for Linux:**
  - easy to install
  - works with standard linux 2.6 kernel
  - simple and clear to use
  - supports Intel and AMD CPUs

- **Current tools:**
  - **likwid-topology**: Print thread and cache topology
  - **likwid-pin**: Pin threaded application without touching code
  - **likwid-perfctr**: Measure performance counters
  - **likwid-mpirun**: mpirun wrapper script for easy LIKWID integration
  - **likwid-bench**: Low-level bandwidth benchmark generator tool
  - … some more
Output of `likwid-topology -g`
on one node of Cray XE6 “Hermit”

---

**CPU type:** AMD Interlagos processor

---

**Hardware Thread Topology**

---

**Sockets:** 2  
**Cores per socket:** 16  
**Threads per core:** 1  

---

<table>
<thead>
<tr>
<th>HWThread</th>
<th>Thread</th>
<th>Core</th>
<th>Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>[...]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Socket 0:** ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 )  
**Socket 1:** ( 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 )

---

**Cache Topology**

---

**Level:** 1  
**Size:** 16 kB  
Output of likwid-topology continued

Level: 2
Size: 2 MB

Level: 3
Size: 6 MB
Cache groups: ( 0 1 2 3 4 5 6 7 ) ( 8 9 10 11 12 13 14 15 ) ( 16 17 18 19 20 21 22 23 ) ( 24 25 26 27 28 29 30 31 )

NUMA Topology

NUMA domains: 4

Domain 0:
Processors: 0 1 2 3 4 5 6 7
Memory: 7837.25 MB free of total 8191.62 MB

Domain 1:
Processors: 8 9 10 11 12 13 14 15
Memory: 7860.02 MB free of total 8192 MB

Domain 2:
Processors: 16 17 18 19 20 21 22 23
Memory: 7847.39 MB free of total 8192 MB

Domain 3:
Processors: 24 25 26 27 28 29 30 31
Memory: 7785.02 MB free of total 8192 MB
Output of likwid-topology continued

Graphical:

Socket 0:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
+ + + + + + + + + + + + + + + + + + + +
16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB
+ + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + +

6MB
```

Socket 1:

```
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
+ + + + + + + + + + + + + + + + + + + +
16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB | 16kB
+ + + + + + + + + + + + + + + + + + + +
+ + + + + + + + + + + + + + + + + + + +

6MB
```
Enforcing thread/process-core affinity under the Linux OS

- Standard tools and OS affinity facilities under program control
- likwid-pin
- aprun (Cray)
Example: STREAM benchmark on 12-core Intel Westmere: Anarchy vs. thread pinning

There are several reasons for caring about affinity:

- Eliminating performance variation
- Making use of architectural features
- Avoiding resource contention
Generic thread/process-core affinity under Linux

Overview

- `taskset [OPTIONS] [MASK | -c LIST] \ [PID | command [args]...]`

- `taskset` binds processes/threads to a set of CPUs. Examples:

  ```bash
  taskset 0x0006 ./a.out
  taskset -c 4 33187
  mpirun -np 2 taskset -c 0,2 ./a.out # doesn’t always work
  ```

- Processes/threads can still move within the set!
- Alternative: let process/thread bind itself by executing syscall

  ```c
  #include <sched.h>
  int sched_setaffinity(pid_t pid, unsigned int len,
                        unsigned long *mask);
  ```

- Disadvantage: which CPUs should you bind to on a non-exclusive machine?

- Still of value on multicore/multisocket cluster nodes, UMA or ccNUMA
Generic thread/process-core affinity under Linux

- Complementary tool: `numactl`

**Example:** `numactl --physcpubind=0,1,2,3 command [args]`  
Bind process to specified physical core numbers

**Example:** `numactl --cpunodebind=1 command [args]`  
Bind process to specified ccNUMA node(s)

- Many more options (e.g., interleave memory across nodes)
  - → see section on ccNUMA optimization

- Diagnostic command (see earlier):  
  `numactl --hardware`

- Again, this is not suitable for a shared machine
More thread/Process-core affinity ("pinning") options

- **Highly OS-dependent system calls**
  - But available on all systems
  - Linux: `sched_setaffinity()`, PLPA (see below) → hwloc
  - Solaris: `processor_bind()`
  - Windows: `SetThreadAffinityMask()`
  ...

- **Support for “semi-automatic” pinning in some compilers/environments**
  - Intel compilers > V9.1 (`KMP_AFFINITY` environment variable)
  - PGI, Pathscale, GNU
  - SGI Altix `dplace` (works with logical CPU numbers!)
  - Generic Linux: `taskset`, `numactl`, `likwid-pin` (see below)

- **Affinity awareness in MPI libraries**
  - SGI MPT
  - OpenMPI
  - Intel MPI
  - ...

Example for program-controlled affinity: Using PLPA under Linux!
Likwid-pin

Overview

- Pins processes and threads to specific cores **without touching code**
- Directly supports pthreads, gcc OpenMP, Intel OpenMP
- Based on combination of wrapper tool together with overloaded pthread library → **binary must be dynamically linked!**
- Can also be used as a superior replacement for **taskset**
- Supports **logical core numbering** within a node and within an existing CPU set
  - Useful for running inside CPU sets defined by someone else, e.g., the MPI start mechanism or a batch system

Usage examples:

- `likwid-pin -c 0,2,4-6 ./myApp parameters`
- `likwid-pin -c S0:0-3 ./myApp parameters`
Running the STREAM benchmark with likwid-pin:

```
$ export OMP_NUM_THREADS=4
$ likwid-pin -s 0x1 -c 0,1,4,5 ./stream
[likwid-pin] Main PID -> core 0 - OK

Double precision appears to have 16 digits of accuracy
Assuming 8 bytes per DOUBLE PRECISION word

[... some STREAM output omitted ...]
The *best* time for each test is used
*EXCLUDING* the first and last iterations

[pthread wrapper] PIN_MASK: 0->1 1->4 2->5
[pthread wrapper] SKIP MASK: 0x1
[pthread wrapper 0] Notice: Using libpthread.so.0
  threadid 1073809728 -> SKIP
[pthread wrapper 1] Notice: Using libpthread.so.0
  threadid 1078008128 -> core 1 - OK
[pthread wrapper 2] Notice: Using libpthread.so.0
  threadid 1082206528 -> core 4 - OK
[pthread wrapper 3] Notice: Using libpthread.so.0
  threadid 1086404928 -> core 5 - OK
[... rest of STREAM output omitted ...]
```
Core numbering may vary from system to system even with identical hardware

- Likwid-topology delivers this information, which can then be fed into likwid-pin
- Alternatively, likwid-pin can abstract this variation and provide a purely logical numbering (physical cores first)

Across all cores in the node:

```
OMP_NUM_THREADS=8 likwid-pin -c N:0-7 ./a.out
```

Across the cores in each socket and across sockets in each node:

```
OMP_NUM_THREADS=8 likwid-pin -c S0:0-3@S1:0-3 ./a.out
```
### Possible unit prefixes

- **N** node
- **S** socket
- **M** NUMA domain
- **C** outer level cache group

Default if `-c` is not specified!
aprun on Cray

- See Cray workshop slides

- **aprun supports only physical core numbering**
  - This is OK since the cores are always numbered consecutively on Crays
  - Use `-ss` switch to restrict allocation to local NUMA domain (see later for more on ccNUMA)
  - Use `-d $OMP_NUM_THREADS` or similar for MPI+OMP hybrid code

- See later on how using multiple cores per module/chip/socket affects performance
Multicore performance tools: Probing performance behavior

likwid-perfctr
Basic approach to performance analysis

1. **Runtime profile / Call graph (gprof)**
2. **Instrument those parts which consume a significant part of runtime**
3. **Find performance signatures**

Possible signatures:
- **Bandwidth saturation**
- **Instruction throughput limitation** (real or language-induced)
- **Latency impact** (irregular data access, high branch ratio)
- **Load imbalance**
- **ccNUMA issues** (data access across ccNUMA domains)
- **Pathologic cases** (false cacheline sharing, expensive operations)
Probing performance behavior

- **How do we find out about the performance properties and requirements of a parallel code?**
  - Profiling via advanced tools is often overkill
- **A coarse overview is often sufficient**
  - likwid-perfctr (similar to “perfex” on IRIX, “hpmcount” on AIX, “lipfpm” on Linux/Altix)
  - Simple end-to-end measurement of hardware performance metrics
  - “Marker” API for starting/stopping counters
  - Multiple measurement region support
  - Preconfigured and extensible metric groups, list with likwid-perfctr -a

- BRANCH: Branch prediction miss rate/ratio
- CACHE: Data cache miss rate/ratio
- CLOCK: Clock of cores
- DATA: Load to store ratio
- FLOPS_DP: Double Precision MFlops/s
- FLOPS_SP: Single Precision MFlops/s
- FLOPS_X87: X87 MFlops/s
- L2: L2 cache bandwidth in MBytes/s
- L2CACHE: L2 cache miss rate/ratio
- L3: L3 cache bandwidth in MBytes/s
- L3CACHE: L3 cache miss rate/ratio
- MEM: Main memory bandwidth in MBytes/s
- TLB: TLB miss rate/ratio
$ env OMP_NUM_THREADS=4 likwid-perfctr -C N:0-3 -g FLOPS_DP ./stream.exe

CPU type: Intel Core Lynnfield processor  
CPU clock: 2.93 GHz

Measuring group FLOPS_DP

YOUR PROGRAM OUTPUT

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>1.97463e+08</td>
<td>2.31001e+08</td>
<td>2.30963e+08</td>
<td>2.31885e+08</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>9.56999e+08</td>
<td>9.58401e+08</td>
<td>9.58637e+08</td>
<td>9.57338e+08</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED</td>
<td>4.00294e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR</td>
<td>882</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION</td>
<td>4.00303e+07</td>
<td>3.08927e+07</td>
<td>3.08866e+07</td>
<td>3.08904e+07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.326242</td>
<td>0.32672</td>
<td>0.326801</td>
<td>0.326358</td>
</tr>
<tr>
<td>CPI</td>
<td>4.84647</td>
<td>4.14891</td>
<td>4.15061</td>
<td>4.12849</td>
</tr>
<tr>
<td>DP MFlops/s (DP assumed)</td>
<td>245.399</td>
<td>189.108</td>
<td>189.024</td>
<td>189.304</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>122.698</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>0.00270351</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SP MUOPS/s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DP MUOPS/s</td>
<td>122.701</td>
<td>94.554</td>
<td>94.5121</td>
<td>94.6519</td>
</tr>
</tbody>
</table>

Always measured

Configured metrics (this group)

Derived metrics
Best practices for runtime counter analysis

Things to look at (in roughly this order)

- Load balance (flops, instructions, BW)
- In-socket memory BW saturation
- Shared cache BW saturation
- Flop/s, loads and stores per flop metrics
- SIMD vectorization
- CPI metric
- # of instructions, branches, mispredicted branches

Caveats

- Load imbalance may not show in CPI or # of instructions
  - Spin loops in OpenMP barriers/MPI blocking calls
  - Looking at “top” or the Windows Task Manager does not tell you anything useful
- In-socket performance saturation may have various reasons
- Cache miss metrics are overrated
  - If I really know my code, I can often calculate the misses
  - Runtime and resource utilization is much more important
- Instructions retired / CPI may not be a good indication of useful workload – at least for numerical / FP intensive codes.

- Floating Point Operations Executed is often a better indicator.

- Waiting / “Spinning” in barrier generates a high instruction count.

```
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, I
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```
likwid-perfctr
... and load-balanced codes

```
env OMP_NUM_THREADS=6 likwid-perfctr -C S0:0-5 -g FLOPS_DP ./a.out
```

### Higher CPI but better performance

```
!$OMP PARALLEL DO
DO I = 1, N
  DO J = 1, N
    x(I) = x(I) + A(J,I) * y(J)
  ENDDO
ENDDO
!$OMP END PARALLEL DO
```
Detecting latency-bound codes

Example: graph and tree data structures

<table>
<thead>
<tr>
<th>Metric</th>
<th>Red-Black tree</th>
<th>Optimized data structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions retired</td>
<td>1.34268e+11</td>
<td>1.28581e+11</td>
</tr>
<tr>
<td>CPI</td>
<td>9.0176</td>
<td>0.71887</td>
</tr>
<tr>
<td>L3-MEM data volume [GB]</td>
<td>301</td>
<td>3.22</td>
</tr>
<tr>
<td>TLB misses</td>
<td>3.71447e+09</td>
<td>4077</td>
</tr>
<tr>
<td>Branch rate</td>
<td>36%</td>
<td>8.5%</td>
</tr>
<tr>
<td>Branch mispredicted ratio</td>
<td>7.8%</td>
<td>0.0000013%</td>
</tr>
<tr>
<td>Memory bandwidth [GB/s]</td>
<td>10.5</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Useful likwid-perfctr groups: L3, L3CACHE, MEM, TLB, BRANCH

High CPI, near perfect scaling if using SMT threads (Intel).
Note: Latency bound code can still produce significant aggregated bandwidth.
Language-induced problems

- The object-oriented programming paradigm implements functionality resulting in many calls to small functions
- The ability of the compiler to inline functions (and still generate the best possible machine code) is limited
- Frequent pattern with complex C++ codes

Symptoms:
- Low (“good”) CPI
- Low resource utilization (Flops/s, bandwidth)
- Orders of magnitude more general purpose than arithmetic floating point instructions
- High branch rate

Solution:
- Use basic data types and plain arrays in compute intensive loops
- Use plain C-like code
- Keep things simple – do not obstruct the compiler’s view on the code
Microbenchmarking for architectural exploration

The vector triad
Serial, throughput, and parallel benchmarks
The parallel vector triad benchmark
A “swiss army knife” for microbenchmarking

Simple streaming benchmark:

```fortran
double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

double precision, dimension(N) :: A,B,C,D
A=1.d0; B=A; C=A; D=A

! Simple streaming benchmark:
! 1. Report performance for different N
! 2. Choose NITER so that accurate time measurement is possible
! 3. 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)

Cray Workshop

Performance for Multicore
\[ A(\cdot) = B(\cdot) + C(\cdot) \times D(\cdot) \text{ on one Sandy Bridge core (3 GHz)} \]

Theoretical limit: 4 W / iteration → 128 GB/s

Data far away → smaller SIMD impact

2.66x SIMD impact

4 W / iteration → 48 GB/s
The throughput-parallel vector triad benchmark

- Every core runs its own, independent triad benchmark

```fortran
!$OMP PARALLEL private(i,j,A,B,C,D)
allocate(A(1:N),B(1:N),C(1:N),D(1:N))
A=1.d0; B=A; C=A; D=A
do j=1,NITER
  do i=1,N
    A(i) = B(i) + C(i) * D(i)
  enddo
  if(.something.that.is.never.true.) then
    call dummy(A,B,C,D)
  endif
enddo
!$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
The OpenMP-parallel vector triad benchmark

- OpenMP work sharing in the benchmark loop

```fortran
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
```

Implicit barrier
OpenMP vector triad on Sandy Bridge socket (3 GHz)

- sync overhead grows with # of threads
- bandwidth scalability across memory interfaces
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 ENDDO
!$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 Interlagos

**Barrier overhead in CPU cycles**

<table>
<thead>
<tr>
<th>2 Threads</th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L2</td>
<td>258</td>
<td>3995</td>
<td>1503</td>
<td>128623</td>
</tr>
<tr>
<td>Shared L3</td>
<td>698</td>
<td>2853</td>
<td>1076</td>
<td>128611</td>
</tr>
<tr>
<td>Same socket</td>
<td>879</td>
<td>2785</td>
<td>1297</td>
<td>128695</td>
</tr>
<tr>
<td>Other socket</td>
<td>940</td>
<td>2740 / 4222</td>
<td>1284 / 1325</td>
<td>128718</td>
</tr>
</tbody>
</table>

Intel compiler barrier very expensive on Interlagos

OpenMP & Cray compiler

<table>
<thead>
<tr>
<th>Full domain</th>
<th>Cray 8.03</th>
<th>GCC 4.6.2</th>
<th>PGI 11.8</th>
<th>Intel 12.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared L3</td>
<td>2272</td>
<td>27916</td>
<td>5981</td>
<td>151939</td>
</tr>
<tr>
<td>Socket</td>
<td>3783</td>
<td>49947</td>
<td>7479</td>
<td>163561</td>
</tr>
<tr>
<td>Node</td>
<td>7663</td>
<td>167646</td>
<td>9526</td>
<td>178892</td>
</tr>
</tbody>
</table>
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

Intel compiler 😊

<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>
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 ☹️.
Simple performance modeling: The Roofline Model
The Roofline Model\textsuperscript{1,2}

1. \( P_{\text{max}} = \text{Applicable peak performance of a loop, assuming that data comes from L1 cache} \)

2. \( I = \text{Computational intensity ("work" per byte transferred) over the slowest data path utilized ("the bottleneck")} \)
   - Code balance \( B_C = I^{-1} \)

3. \( b_S = \text{Applicable peak bandwidth of the slowest data path utilized} \)

Expected performance:

\[
P = \min(P_{\text{max}}, I \cdot b_S)
\]
A simple Roofline example

Example:  
\[
\text{do } i=1,N; \ s=s+a(i); \ \text{enddo}
\]
in double precision on hypothetical 3 GHz CPU, 4-way SIMD, N large

\[
P = \min(P_{\text{max}}, I \cdot b_S)
\]

- ADD peak (half of full peak)
- no SIMD
- 4-cycle latency per ADD if not unrolled

Computational intensity
Applicable peak for the summation loop

Plain scalar code, no SIMD

LOAD r1.0 ← 0
i ← 1
loop:
  LOAD r2.0 ← a(i)
  ADD r1.0 ← r1.0 + r2.0
  ++i →? loop
result ← r1.0

ADD pipes utilization:

⇒ 1/16 of ADD peak
Applicable peak for the summation loop

Scalar code, 4-way unrolling

```
LOAD r1.0 ← 0
LOAD r2.0 ← 0
LOAD r3.0 ← 0
LOAD r4.0 ← 0
i ← 1
loop:
    LOAD r5.0 ← a(i)
    LOAD r6.0 ← a(i+1)
    LOAD r7.0 ← a(i+2)
    LOAD r8.0 ← a(i+3)
    ADD r1.0 ← r1.0+r5.0
    ADD r2.0 ← r2.0+r6.0
    ADD r3.0 ← r3.0+r7.0
    ADD r4.0 ← r4.0+r8.0
i+4 →? loop
result ← r1.0+r2.0+r3.0+r4.0
```

ADD pipes utilization:

```
  t
```

→ 1/4 of ADD peak
Applicable peak for the summation loop

SIMD-vectorized, 4-way unrolled

```
LOAD [r1.0,...,r1.3] ← [0,0]
LOAD [r2.0,...,r2.3] ← [0,0]
LOAD [r3.0,...,r3.3] ← [0,0]
LOAD [r4.0,...,r4.3] ← [0,0]
i ← 1
loop:
  LOAD [r5.0,...,r5.3] ← [a(i),...,a(i+3)]
  LOAD [r6.0,...,r6.3] ← [a(i+4),...,a(i+7)]
  LOAD [r7.0,...,r7.3] ← [a(i+8),...,a(i+11)]
  LOAD [r8.0,...,r8.3] ← [a(i+12),...,a(i+15)]
  ADD r1 ← r1+r5
  ADD r2 ← r2+r6
  ADD r3 ← r3+r7
  ADD r4 ← r4+r8
  i+=16 →? loop
result ← r1.0+r1.1+...+r4.2+r4.3
```
Input to the roofline model

... on the example of: do i=1,N; s=s+a(i); enddo

**Throughput:** 1 ADD + 1 LD/cy  
**Pipeline depth:** 4 cy (ADD)  
**4-way SIMD**

**Memory-bound @ large N!**  
\[ P_{\text{max}} = 1.25 \text{ GF/s} \]

**Maximum memory bandwidth:** 10 GB/s

**Cray Workshop**  
**Performance for Multicore**
A very bandwidth-bound kernel

Example: Vector triad $A(:)=B(:)+C(:)*D(:)$ on 2.3 GHz Interlagos

- $b_S = 34$ GB/s
- $B_c = (4+1)$ Words / 2 Flops = 2.5 W/F (including write allocate)
  $\Rightarrow I = 0.4$ F/W = 0.05 F/B

Lightspeed:

$I \cdot b_S = 1.7$ GF/s (1.2 % of peak performance)
Assumptions for the Roofline Model

- The balance metric formalism is based on some (crucial) assumptions:
  - There is a clear concept of “work” vs. “traffic”
    - “work” = flops, updates, iterations…
    - “traffic” = required data to do “work”
  - Attainable bandwidth of code = input parameter! Determine effective bandwidth via simple streaming benchmarks to model more complex kernels and applications
  - Data transfer and core execution overlap perfectly!
  - Slowest data path is modeled only; all others are assumed to be infinitely fast
  - If data transfer is the limiting factor, the bandwidth of the slowest data path can be utilized to 100% (“saturation”)
  - Latency effects are ignored, i.e. perfect streaming mode
Factors to consider in the roofline model

Bandwidth-bound (simple case)
- Accurate traffic calculation (write-allocate, strided access, …)
- Practical ≠ theoretical BW limits
- Erratic access patterns

Core-bound (may be complex)
- Multiple bottlenecks: LD/ST, arithmetic, pipelines, SIMD, execution ports
- See next slide…
Complexities of in-core execution

Multiple bottlenecks:

- L1 Icache bandwidth
- Decode/retirement throughput
- Port contention (direct or indirect)
- Arithmetic pipeline stalls (dependencies)
- Overall pipeline stalls (branching)
- L1 Dcache bandwidth (LD/ST throughput)
- Scalar vs. SIMD execution
- ...

- Register pressure
- Alignment issues
Shortcomings of the roofline model

- **Saturation effects in multicore chips are not explained**
  - Reason: “saturation assumption”
  - Cache line transfers and core execution do sometimes not overlap perfectly
  - Only increased “pressure” on the memory interface can saturate the bus
    → need more cores!

- **ECM model gives more insight**

Optimal utilization of parallel resources

Hardware-software interaction
SIMD parallelism
Computer Architecture

The evil of hardware optimizations

Architect’s view:
Make the common case fast!

Flexible, but optimization is hard!

Provide improvements for relevant software
What are the technical opportunities?
Economical concerns
Multi-way special purpose

What is your relevant aspect of the architecture?

EDSAC 1949
ENIAC 1948
Hardware-Software Co-Design?
From algorithm to execution

The machine view:
ISA (Machine code)

Reality:
Algorithm
Programming language
Compiler tries

Hardware = Black Box
1. **Instruction execution**
   This is the primary resource of the processor. All efforts in hardware design are targeted towards increasing the instruction throughput.

2. **Data transfer bandwidth**
   Data transfers are a consequence of instruction execution and therefore a secondary resource. Maximum bandwidth is determined by the request rate of executed instructions and technical limitations (bus width, speed).

**Real machine:** Processors are imperfect and have technical limitations. This results in hazards preventing to fully exploit the elementary resources.
Things to remember

Goals for optimization:

1. Map your work to an instruction mix with highest throughput using the most effective instructions.

2. Reduce data volume over slow data paths fully utilizing available bandwidth.

3. Avoid possible hazards/overhead which prevent reaching goals one and two.
Coding for Single Instruction Multiple Data-processing
SIMD processing – Basics

- **Single Instruction Multiple Data (SIMD) operations** allow the concurrent execution of the same operation on “wide” registers.

- **x86 SIMD instruction sets:**
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands

- **Adding two registers holding double precision floating point operands**

![Diagram showing scalar and SIMD execution](image)
SIMD processing – Basics

Steps (done by the compiler) for “SIMD processing”

```
for(int i=0; i<n;i++)
    C[i]=A[i]+B[i];
```

“Loop unrolling”

```
for(int i=0; i<n;i+=4){
    C[i] =A[i] +B[i];
    C[i+1]=A[i+1]+B[i+1];
}
```

//remainder loop omitted

Load 256 Bits starting from address of A[i] to register R0

Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2

Store R2 (256 Bit) to address starting at C[i]

LABEL1:

```
VLOAD R0 ← A[i]
VLOAD R1 ← B[i]
V64ADD[R0,R1] → R2
VSTORE R2 → C[i]
i←i+4
i<(n-4)? JMP LABEL1
```

//remainder loop omitted
SIMD processing – Basics

- No SIMD-processing for loops with data dependencies

```c
for(int i=0; i<n; i++)
    A[i] = A[i-1] * s;
```

- “Pointer aliasing” may prevent compiler from SIMD-processing

```c
void scale_shift(double *A, double *B, double *C, int n) {
    for(int i=0; i<n; ++i)
        C[i] = A[i] + B[i];
}
```

- C/C++ allows that A \rightarrow &C[-1] and B \rightarrow &C[-2]
  \rightarrow C[i] = C[i-1] + C[i-2]: dependency \rightarrow No SIMD-processing

- If no “Pointer aliasing” is used, tell the compiler, e.g. use -fno-alias switch for Intel compiler \rightarrow SIMD-processing
SIMD processing – Basics

- SIMD processing of a vector norm

```
float s = 0.0;
for (int i = 0; i < n; i++)
    s = s + A[i] * A[i];
```

Data dependency on `s` must be resolved for SIMD-processing

```
float s0 = 0.0;
s1 = 0.0;
s2 = 0.0;
s3 = 0.0;
for (int i = 0; i < n; i += 4)
{
    s0 = s0 + A[i] * A[i];
    s1 = s1 + A[i+1] * A[i+1];
    s2 = s2 + A[i+2] * A[i+2];
    s3 = s3 + A[i+3] * A[i+3];
}

s = s0 + s1 + s2 + s3
```

Compiler does transformation – if programmer allows it to do so! (~`O3` instead of ~`O1`)

```
V64MULT(R1, R2) \rightarrow R1
V64ADD(R0, R1) \rightarrow R0
```

...
Reading x86 assembly code
Basic approach to check the instruction code

- Get the assembler code (Intel compiler):
  ```
  icc -S -O3 -xHost triad.c -o triad.s
  ```

- Disassemble Executable:
  ```
  objdump -d ./cacheBench | less
  ```

- Things to check for:
  - Is the code vectorized? Search for pd/ps suffix.
    ```
    mulpd, addpd, vaddpd, vmulpd
    ```
  - Is the data loaded with 16 byte moves?
    ```
    movapd, movaps, vmovupd
    ```
  - For memory-bound code: Search for nontemporal stores:
    ```
    movntpd, movntps
    ```

The x86 ISA is documented in:

Intel Software Development Manual (SDM) 2A and 2B
Basics of the x86-64 ISA

- Instructions have 0 to 2 operands
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: A[i] equivalent to *(A+i) (a pointer has a type: A+i*8)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```
```
movaps %xmm4, 48(%rdi,%rax,8)
addq $8, %rax
js .B1.4
```
```
401b9f: 0f 29 5c c7 30
401ba4: 48 83 c0 08
401ba8: 78 a6
```
```
movaps %xmm3,0x30(%rdi,%rax,8)
add $0x8,%rax
js 401b50 <triad_asm+0x4b>
```
16 general Purpose Registers (64bit):
rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15
alias with eight 32 bit register set:
eax, ebx, ecx, edx, esi, edi, esp, ebp

Floating Point SIMD Registers:
xmm0–xmm15   SSE (128bit) alias with 256bit registers
ymm0–ymm15   AVX (256bit)

SIMD instructions are distinguished by:
AVX (VEX) prefix: v
Operation: mul, add, mov
Modifier: non temporal (nt), unaligned (u), aligned (a), high (h)
Data type: single (s), double (d)
Basics of x86-64 ABI

- Regulations how functions are called on binary level
- Differs between 32 bit / 64 bit and Operating Systems

x86-64 on Linux:

- **Integer or address** parameters are passed in the order: 
  
  - rdi, rsi, rdx, rcx, r8, r9

- **Floating Point** parameters are passed in the order xmm0–xmm7

- **Registers which must be preserved across function calls:**
  
  - rbx, rbp, r12–r15

- **Return values** are passed in rax/rdx and xmm0/xmm1
Case Study: summation

```c
float sum = 0.0;

for (int j=0; j<size; j++){
    sum += data[j];
}
```

Instruction code:

```
Instruction address
401d08: f3 0f 58 04 82
401d0d: 48 83 c0 01
401d11: 39 c7
401d13: 77 f3
```

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>addss</td>
<td>(%rdx,%rax,4),%xmm0</td>
</tr>
<tr>
<td>add</td>
<td>$0x1,%rax</td>
</tr>
<tr>
<td>cmp</td>
<td>%eax,%edi</td>
</tr>
<tr>
<td>ja</td>
<td>401d08</td>
</tr>
</tbody>
</table>

To get object code use:

```
objdump -d on object file or executable or compile with -S
```
Summation code variants

Unrolling with sub sums to break up register dependency

1:
addss xmm0, [rsi + rax * 4]
add rax, 1
cmp eax,edi
js 1b

3 cycles add pipeline latency

SSE SIMD vectorization

1:
addps xmm0, [rsi + rax * 4]
addps xmm1, [rsi + rax * 4 + 16]
addps xmm2, [rsi + rax * 4 + 32]
addps xmm3, [rsi + rax * 4 + 48]
add rax, 16
cmp eax,edi
js 1b
**SIMD** influences instruction execution in the core – other bottlenecks stay the same!

**Full benefit in L1 cache**

**Data transfers are overlapped with execution**

**Peak**

- **Scalar**
- **Plain**
- **SIMD**

**Per-cache line cycle counts**

- **Execution**: 48, 16, 4, 4
- **Cache**: 32 byte LD & 16 byte ST
- **Memory**: 15.6 byte/cycle

**Per-cycle transfer widths**

- **Registers**: 32 cycles, 16 cycles, 4 cycles
- **L1D**: 32 byte/cycle, 2 cycles
- **L2**: 32 byte/cycle, 2 cycles
- **L3**: 15.6 byte/cycle, 4 cycles

**Data transfers are overlapped with execution**

**Some penalty for SIMD (12 cy predicted)**

**Execution cycle counts**

- **L1**: 8 cycles
- **L3**: 16 cycles
- **MEM**: 24 cycles

**Full benefit in L1 cache**
**SIMD-processing – Full chip (all cores)**

**Influence of SMT**

Bandwidth saturation is the primary performance limitation on the chip level!

**Conclusion:** If the code saturates the bottleneck, all variants are acceptable!

Full scaling using SMT due to bubbles in pipeline

All variants saturate the memory bandwidth

8 threads on physical cores

16 threads using SMT
How to leverage SIMD

- The compiler does it for you (aliasing, alignment, language)
- Compiler directives (pragmas)
- Alternative **programming models** for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use **intrinsics** the following headers are available. To enable instruction sets often additional flags are necessary:

- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX)
- `x86intrin.h` (all instruction set extensions)
- See next slide for an example
Example: array summation using C intrinsics

```c
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 = _mm_setzero_ps();
sum1 = _mm_setzero_ps();
sum2 = _mm_setzero_ps();
sum3 = _mm_setzero_ps();
for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
sum0 = _mm_add_ps(sum0, sum1);
sum0 = _mm_add_ps(sum0, sum2);
sum0 = _mm_add_ps(sum0, sum3);
sum0 = _mm_hadd_ps(sum0, sum0);
_mm_store_ss(&scalar_sum, sum0);
```
### Example: array summation from intrinsics, instruction code

<table>
<thead>
<tr>
<th>Line</th>
<th>Machine Code</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>0f 57 c9</td>
<td>xorps %xmm1,%xmm1</td>
<td>xorps %xmm1,%xmm1</td>
</tr>
<tr>
<td>17</td>
<td>31 c0</td>
<td>xor %eax,%eax</td>
<td>xor %eax,%eax</td>
</tr>
<tr>
<td>19</td>
<td>0f 28 d1</td>
<td>movaps %xmm1,%xmm2</td>
<td>movaps %xmm1,%xmm2</td>
</tr>
<tr>
<td>1c</td>
<td>0f 28 c1</td>
<td>movaps %xmm1,%xmm0</td>
<td>movaps %xmm1,%xmm0</td>
</tr>
<tr>
<td>1f</td>
<td>0f 28 d9</td>
<td>movaps %xmm1,%xmm3</td>
<td>movaps %xmm1,%xmm3</td>
</tr>
<tr>
<td>22</td>
<td>66 0f 1f 44 00 00</td>
<td>nopw 0x0(%rax,%rax,1)</td>
<td>nopw 0x0(%rax,%rax,1)</td>
</tr>
<tr>
<td>28</td>
<td>0f 10 3e</td>
<td>movups (%rsi),%xmm7</td>
<td>movups (%rsi),%xmm7</td>
</tr>
<tr>
<td>2b</td>
<td>0f 10 76 10</td>
<td>movups 0x10(%rsi),%xmm6</td>
<td>movups 0x10(%rsi),%xmm6</td>
</tr>
<tr>
<td>2f</td>
<td>0f 10 6e 20</td>
<td>movups 0x20(%rsi),%xmm5</td>
<td>movups 0x20(%rsi),%xmm5</td>
</tr>
<tr>
<td>33</td>
<td>0f 10 66 30</td>
<td>movups 0x30(%rsi),%xmm4</td>
<td>movups 0x30(%rsi),%xmm4</td>
</tr>
<tr>
<td>37</td>
<td>83 c0 10</td>
<td>add $0x10,%eax</td>
<td>add $0x10,%eax</td>
</tr>
<tr>
<td>3a</td>
<td>48 83 c6 40</td>
<td>add $0x40,%rsi</td>
<td>add $0x40,%rsi</td>
</tr>
<tr>
<td>3e</td>
<td>0f 58 df</td>
<td>addps %xmm7,%xmm3</td>
<td>addps %xmm7,%xmm3</td>
</tr>
<tr>
<td>41</td>
<td>0f 58 c6</td>
<td>addps %xmm6,%xmm0</td>
<td>addps %xmm6,%xmm0</td>
</tr>
<tr>
<td>44</td>
<td>0f 58 d5</td>
<td>addps %xmm5,%xmm2</td>
<td>addps %xmm5,%xmm2</td>
</tr>
<tr>
<td>47</td>
<td>0f 58 cc</td>
<td>addps %xmm4,%xmm1</td>
<td>addps %xmm4,%xmm1</td>
</tr>
<tr>
<td>4a</td>
<td>39 c7</td>
<td>cmp %eax,%edi</td>
<td>cmp %eax,%edi</td>
</tr>
<tr>
<td>4c</td>
<td>77 da</td>
<td>ja 28 &lt;compute_sum_SSE+0x18&gt;</td>
<td>ja 28 &lt;compute_sum_SSE+0x18&gt;</td>
</tr>
<tr>
<td>4e</td>
<td>0f 58 c3</td>
<td>addps %xmm3,%xmm0</td>
<td>addps %xmm3,%xmm0</td>
</tr>
<tr>
<td>51</td>
<td>0f 58 c2</td>
<td>addps %xmm2,%xmm0</td>
<td>addps %xmm2,%xmm0</td>
</tr>
<tr>
<td>54</td>
<td>0f 58 c1</td>
<td>addps %xmm1,%xmm0</td>
<td>addps %xmm1,%xmm0</td>
</tr>
<tr>
<td>57</td>
<td>f2 0f 7c c0</td>
<td>haddps %xmm0,%xmm0</td>
<td>haddps %xmm0,%xmm0</td>
</tr>
<tr>
<td>5b</td>
<td>f2 0f 7c c0</td>
<td>haddps %xmm0,%xmm0</td>
<td>haddps %xmm0,%xmm0</td>
</tr>
<tr>
<td>5f</td>
<td>c3</td>
<td>retq</td>
<td>retq</td>
</tr>
</tbody>
</table>

**Loop body**
Vectorization and the Intel compiler

- Intel compiler will try to use SIMD instructions when enabled to do so
  - “Poor man’s vector computing”
  - Compiler will emit messages about vectorized loops:

    plain.c(11): (col. 9) remark: LOOP WAS VECTORIZED.

- Use option `-vec_report3` to get full compiler output about which loops were vectorized and which were not and why (data dependencies!)
- Some obstructions will prevent the compiler from applying vectorization even if it is possible

- You can use source code directives to provide more information to the compiler
Vectorization compiler options

- The compiler will vectorize starting with `-O2`.
- To enable specific SIMD extensions use the `-x` option:
  - `-xSSE2` vectorize for SSE2 capable machines
  - Available SIMD extensions: `SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX`
  - `-xAVX` on Sandy Bridge processors

Recommend option:
- `-xHost` will optimize for the architecture you compile on

On AMD Opteron: use plain `-O3` as the `-x` options may involve CPU type checks.
Vectorization compiler options

- **Controlling non-temporal stores**

  - `--opt-streaming-stores always|auto|never`

    - **always**: use NT stores, assume application is memory bound (use with caution!)
    - **auto**: compiler decides when to use NT stores
    - **never**: do not use NT stores unless activated by source code directive
Rules for vectorizable loops

1. Countable
2. Single entry and single exit
3. Straight line code
4. No function calls (exception intrinsic math functions)

Better performance with:
1. Simple inner loops with unit stride
2. Minimize indirect addressing
3. Align data structures (SSE 16 bytes, AVX 32 bytes)
4. In C use the restrict keyword for pointers to rule out aliasing

Obstacles for vectorization:
- Non-contiguous memory access
- Data dependencies
Vectorization source code directives

- **Fine-grained control of loop vectorization**
- **Use** `!DEC$` (Fortran) or `#pragma` (C/C++) sentinel to start a compiler directive

- `#pragma vector always`
  vectorize even if it seems inefficient (hint!)

- `#pragma novector`
  do not vectorize even if possible

- `#pragma vector nontemporal`
  use NT stores when allowed (i.e. alignment conditions are met)

- `#pragma vector aligned`
  specifies that all array accesses are aligned to 16-byte boundaries
  **(DANGEROUS! You must not lie about this!)**
User mandated vectorization

- Starting with Intel Compiler 12.0 the \texttt{simd} pragma is available
- \texttt{#pragma simd} enforces vectorization where the other pragmas fail

Prerequisites:
- Countable loop
- Innermost loop
- Must conform to for-loop style of OpenMP worksharing constructs

There are additional clauses: reduction, vectorlength, private

Refer to the compiler manual for further details

**NOTE:** Using the \texttt{#pragma simd} the compiler may generate incorrect code if the loop violates the vectorization rules!

\begin{verbatim}
#pragma simd reduction(+:x)
for (int i=0; i<n; i++) {
    x = x + A[i];
}
\end{verbatim}
Alignment issues

- Alignment of arrays in SSE calculations should be on 16-byte boundaries to allow packed loads and NT stores (for Intel processors)
  - AMD has a scalar nontemporal store instruction
- Otherwise the compiler will revert to unaligned loads and not use NT stores – even if you say vector nontemporal
- How is manual alignment accomplished?

Dynamic allocation of aligned memory (align = alignment boundary):

```c
#define _XOPEN_SOURCE 600
#include <stdlib.h>

int posix_memalign(void **ptr,
                    size_t align,
                    size_t size);
```
Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes
First touch placement policy
C++ issues
ccNUMA locality and dynamic scheduling
ccNUMA locality beyond first touch
ccNUMA performance problems
“The other affinity” to care about

- ccNUMA:
  - Whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)

- How do we make sure that memory access is always as "local" and "distributed" as possible?

- Page placement is implemented in units of OS pages (often 4kB, possibly more)
Cray XE6 Interlagos node
4 chips, two sockets, 8 threads per ccNUMA domain

- **ccNUMA map**: Bandwidth penalties for remote access
  - Run 8 threads per ccNUMA domain (1 chip)
  - Place memory in different domain → 4x4 combinations
  - STREAM triad benchmark using nontemporal stores

---

![STREAM triad performance graph](chart.png)

**Cray Workshop**

**Performance for Multicore**
ccNUMA locality tool numactl:
How do we enforce some locality of access?

- **numactl** can influence the way a binary maps its memory pages:

  ```
  numactl --membind=<nodes> a.out  # map pages only on <nodes>
  --preferred=<node> a.out  # map pages on <node>
  # and others if <node> is full
  --interleave=<nodes> a.out  # map pages round robin across
  # all <nodes>
  ```

- **Examples:**

  ```
  env OMP_NUM_THREADS=2 numactl --membind=0 --cpunodebind=1 ./stream
  env OMP_NUM_THREADS=4 numactl --interleave=0-3 "
  likwid-pin -c N:0,4,8,12 ./stream
  ```

- **But what is the default without numactl?**
ccNUMA default memory locality

- "Golden Rule" of ccNUMA:
  
  A memory page gets mapped into the local memory of the processor that first touches it!
  
  - Except if there is not enough local memory available
  - This might be a problem, see later

- Caveat: "touch" means "write", not "allocate"

- Example:

```
double *huge = (double*)malloc(N*sizeof(double));
```

```
for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0;
```

- It is sufficient to touch a single item to map the entire page
Most simple case: explicit initialization

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

A = 0.d0

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i) = 0.d0
end do
!$OMP end do
!$OMP end parallel

B(i) = function ( A(i) )
end do
!$OMP end parallel do
```

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
  A(i) = 0.d0
end do
!$OMP end do
!$OMP end parallel

B(i) = function ( A(i) )
end do
!$OMP end parallel do
```
Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```fortran
integer,parameter :: N=10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
   B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

---

- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O.
Coding for Data Locality

- **Required condition:** OpenMP loop schedule of initialization must be the same as in all computational loops
  - Only choice: *static*! Specify explicitly on all NUMA-sensitive loops, just to be sure…
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - Presupposes that all worksharing loops with the same loop length have the same thread-chunk mapping
    - Guaranteed by OpenMP 3.0 only for loops in the same enclosing parallel region and static schedule
    - In practice, it works with any compiler even across regions
  - If dynamic scheduling/tasking is unavoidable, more advanced methods may be in order

- **How about global objects?**
  - Better not use them
  - If communication vs. computation is favorable, might consider *properly placed copies* of global data
  - In C++, STL allocators provide an elegant solution (see hidden slides)
**Diagnosing Bad Locality**

- If your code is cache-bound, you might not notice any locality problems

- Otherwise, bad locality **limits scalability at very low CPU numbers** (whenever a node boundary is crossed)
  - If the code makes good use of the memory interface
  - But there may also be a general problem in your code…

- **Consider using performance counters**
  - LIKWID-perfctr can be used to measure nonlocal memory accesses
  - Example for Intel Nehalem (Core i7):

```
env OMP_NUM_THREADS=8 likwid-perfctr -g MEM -C N:0-7 ./a.out
```
Using performance counters for diagnosing bad ccNUMA access locality

- Intel Nehalem EP node:

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
<th>core 4</th>
<th>core 5</th>
<th>core 6</th>
<th>core 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>5.20725e+08</td>
<td>5.24793e+08</td>
<td>5.21547e+08</td>
<td>5.23717e+08</td>
<td>5.28269e+08</td>
<td>5.29083e+08</td>
<td>5.30103e+08</td>
<td>5.29479e+08</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>1.90447e+09</td>
<td>1.90599e+09</td>
<td>1.90619e+09</td>
<td>1.90673e+09</td>
<td>1.90583e+09</td>
<td>1.90746e+09</td>
<td>1.90632e+09</td>
<td>1.9071e+09</td>
</tr>
<tr>
<td>UNC_QMC_NORMAL_READS_ANY</td>
<td>8.17606e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8.07797e+07</td>
<td>0</td>
<td>0</td>
<td>8.07797e+07</td>
</tr>
<tr>
<td>UNC_QMC_WRITES_FULL_ANY</td>
<td>5.53837e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5.51052e+07</td>
<td>0</td>
<td>0</td>
<td>5.51052e+07</td>
</tr>
<tr>
<td>UNC_QHL_REQUESTS_REMOTE_READS</td>
<td>6.84504e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6.8107e+07</td>
<td>0</td>
<td>0</td>
<td>6.8107e+07</td>
</tr>
<tr>
<td>UNC_QHL_REQUESTS_LOCAL_READS</td>
<td>6.82751e+07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6.76274e+07</td>
<td>0</td>
<td>0</td>
<td>6.76274e+07</td>
</tr>
</tbody>
</table>

RDTSC timing: 0.827196 s

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
<th>core 4</th>
<th>core 5</th>
<th>core 6</th>
<th>core 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.714167</td>
<td>0.714733</td>
<td>0.71481</td>
<td>0.715013</td>
<td>0.714673</td>
<td>0.715286</td>
<td>0.71486</td>
<td>0.71515</td>
</tr>
<tr>
<td>Memory bandwidth [MBytes/s]</td>
<td>10610.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10513.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Remote Read BW [MBytes/s]</td>
<td>5296</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5269.43</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Uncore events only counted once per socket

Half of read BW comes from other socket!
If all fails…

- Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?
  
  - Program has erratic access patterns → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

```bash
# numactl --hardware    # idle node!
available: 2 nodes (0-1)
node 0 size: 2047 MB
node 0 free: 906 MB
node 1 size: 1935 MB
node 1 free: 1798 MB
```

```
Mem: 4065564k total, 1149400k used, 2716164k free, 43388k buffers
Swap: 2104504k total, 2656k used, 2101848k free, 1038412k cached
```
ccNUMA problems beyond first touch: Buffer cache

- **OS uses part of main memory for disk buffer (FS) cache**
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - → non-local access!
  - “sync” is not sufficient to drop buffer cache blocks

- **Remedies**
  - Drop FS cache pages after user job has run (admin’s job)
    - seems to be automatic after aprun has finished on Crays
  - User can run “sweeper” code that allocates and touches all physical memory before starting the real application
  - numactl tool or aprun can force local allocation (where applicable)
  - Linux: There is no way to limit the buffer cache size in standard kernels
ccNUMA problems beyond first touch: Buffer cache

Real-world example: ccNUMA and the Linux buffer cache

Benchmark:

1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory available in LD0

Result: By default, Buffer cache is given priority over local page placement → restrict to local domain if possible!

![Graph showing bandwidth vs file size]
ccNUMA placement and erratic access patterns

- Sometimes access patterns are just not nicely grouped into contiguous chunks:

  ```fortran
  double precision :: r, a(M)
  !$OMP parallel do private(r)
  do i=1,N
    call RANDOM_NUMBER(r)
    ind = int(r * M) + 1
    res(i) = res(i) + a(ind)
  enddo
  !$OMP end parallel do
  ```

- Or you have to use tasking/dynamic scheduling:

  ```fortran
  !$OMP parallel
  !$OMP single
  do i=1,N
    call RANDOM_NUMBER(r)
    if(r.le.0.5d0) then
      !$OMP task
      call do_work_with(p(i))
    endif
  enddo
  !$OMP end single
  !$OMP end parallel
  ```

- In both cases page placement cannot easily be fixed for perfect parallel access
Worth a try: Interleave memory across ccNUMA domains to get at least some parallel access

1. Explicit placement:

```c
!$OMP parallel do schedule(static,512)
do i=1,M
  a(i) = ...
enddo
!$OMP end parallel do
```

2. Using global control via `numactl`:

```
numactl --interleave=0-3 ./a.out
```

- Fine-grained program-controlled placement via `libnuma` (Linux) using, e.g., `numa_alloc_interleaved_subset()`, `numa_alloc_interleaved()` and others

Observe page alignment of array to get proper placement!

This is for all memory, not just the problematic arrays!
The curse and blessing of interleaved placement: 
*OpenMP STREAM on a Cray XE6 Interlagos node*

- **Parallel init:** Correct parallel initialization
- **LD0:** Force data into LD0 via `numactl -m 0`
- **Interleaved:** `numactl --interleave <LD range>`
Case study:
A 3D Jacobi smoother

The basics in two dimensions
Roofline performance analysis and modeling
A Jacobi smoother

- **Laplace equation in 2D:** \( \Delta \Phi = 0 \)

- **Solve** with Dirichlet boundary conditions using Jacobi iteration scheme:

  ```
  double precision, dimension(0:imax+1,0:kmax+1,0:1) :: phi
  integer :: t0,t1
  t0 = 0 ; t1 = 1
  do it = 1,itmax       ! choose suitable number of sweeps
    do k = 1,kmax
      do i = 1,imax
        ! four flops, one store, four loads
        phi(i,k,t1) = ( phi(i+1,k,t0) + phi(i-1,k,t0)
                        + phi(i,k+1,t0) + phi(i,k-1,t0) ) * 0.25
      enddo
    enddo
    ! swap arrays
    i = t0 ; t0=t1 ; t1=i
  enddo
  ```

  - **Naive balance (incl. write allocate):**
    - \( \Phi(i,:,:) : 3 \text{ LD} + 3 \text{ ST} \)
    - \( \Phi(:,,:) : 1 \text{ ST} + 1 \text{ LD} \)

  \( \Rightarrow B_C = 5 \text{ W} / 4 \text{ FLOPs} = 1.25 \text{ W} / \text{ F} \)
Modern cache subsystems may further reduce memory traffic

→ “layer conditions”

If cache is large enough to hold at least 2 rows (shaded region): Each $\phi(:,:,t0)$ is loaded once from main memory and re-used 3 times from cache:

$$\phi(:,:,t0): 1 \text{ LD} + \phi(:,:,t1): 1 \text{ ST} + 1 \text{ LD}$$

$$B_C = \frac{3 \text{ W}}{4 \text{ F}} = 0.75 \frac{\text{ W}}{\text{ F}}$$

If cache is too small to hold one row:

$$\phi(:,:,t0): 2 \text{ LD} + \phi(:,:,t1): 1 \text{ ST} + 1 \text{ LD}$$

$$B_C = \frac{5 \text{ W}}{4 \text{ F}} = 1.25 \frac{\text{ W}}{\text{ F}}$$
Performance metrics: 2D Jacobi

- **Alternative implementation** ("Macho FLOP version")

```
  do k = 1, kmax
    do i = 1, imax
      phi(i,k,t1) = 0.25 * phi(i+1,k,t0) + 0.25 * phi(i-1,k,t0) 
                   + 0.25 * phi(i,k+1,t0) + 0.25 * phi(i,k-1,t0)
    enddo
  enddo
```

- **MFlops/sec increases by 7/4 but time to solution remains the same**

- **Better metric** (for many iterative stencil schemes): **Lattice Site Updates per Second (LUPs/sec)**

2D Jacobi example: Compute LUPs/sec metric via

\[
P[LUPs/s] = \frac{it_{\text{max}} \cdot imax \cdot k_{\text{max}}}{T_{\text{wall}}}
\]
2D → 3D

- **3D sweep:**
  
  ```
  do k=1,kmax
    do j=1,jmax
      do i=1,imax
        phi(i,j,k,t1) = 1/6. * (phi(i-1,j,k,t0) + phi(i+1,j,k,t0) &
                               + phi(i,j-1,k,t0) + phi(i,j+1,k,t0) &
                               + phi(i,j,k-1,t0) + phi(i,j,k+1,t0))
      enddo
    enddo
  enddo
  ```

- **Best case balance:** 1 LD  
  1 ST + 1 write allocate  
  6 flops

  \[ B_C = 0.5 \text{ W/F (24 bytes/LUP)} \]

- **No 2-layer condition but 2 rows fit:** \( B_C = 5/6 \text{ W/F (40 bytes/LUP)} \)

- **Worst case (2 rows do not fit):** \( B_C = 7/6 \text{ W/F (56 bytes/LUP)} \)
3D Jacobi solver

Performance of vanilla code on one Interlagos chip (8 cores)

Problem size: $N^3$

- Cache
- Memory

- Roofline inappropriate for unsaturated case

- 2 layers of source array drop out of L2 cache

- Performance model (mem.)

- $T=1$
- $T=2$
- $T=8$
Conclusions from the Jacobi example

- We have made sense of the memory-bound performance vs. problem size
  - “Layer conditions” lead to predictions of code balance
  - Achievable memory bandwidth is input parameter

- The model works only if the bandwidth is “saturated”
  - In-cache modeling is more involved

- Optimization == reducing the code balance by code transformations
  - See below
Data access optimizations

Case study: Optimizing the 3D Jacobi solver
Remember the 3D Jacobi solver on Interlagos?

![Graph showing performance versus linear problem size N.](image)
Jacobi iteration (2D): No spatial Blocking

- **Assumptions:**
  - cache can hold 32 elements (16 for each array)
  - Cache line size is 4 elements
  - Perfect eviction strategy for source array

This element is needed for three more updates; but 29 updates happen before this element is used for the last time
Jacobi iteration (2D): No spatial blocking

**Assumptions:**
- cache can hold 32 elements (16 for each array)
- Cache line size is 4 elements
- Perfect eviction strategy for source array

This element is needed for three more updates but has been evicted
Jacobi iteration (2D): Spatial Blocking

- divide system into blocks
- update block after block
- same performance as if three complete rows of the systems fit into cache
Spatial blocking reorders traversal of data to account for the data update rule of the code

Elements stay sufficiently long in cache to be fully reused

Spatial blocking improves temporal locality!
(Continuous access in inner loop ensures spatial locality)

This element remains in cache until it is fully used (only 6 updates happen before last use of this element)
Jacobi iteration (3D): Spatial blocking

- **Implementation:**

  ```fortran
  do ioffset=1,imax,iblock
    do joffset=1,jmax,jblock
      do k=1,kmax
        do j=joffset, min(jmax,joffset+jblock-1)
          do i=ioffset, min(imax,ioffset+iblock-1)
            phi(i,j,k,t1) = ( phi(i-1,j,k,t0)+phi(i+1,j,k,t0)
                            + ... + phi(i,j,k-1,t0)+phi(i,j,k+1,t0) )/6.d0
          enddo
        enddo
      enddo
    enddo
  enddo
  enddo
  enddo
  enddo
  ```

- **Guidelines:**
  - Blocking of inner loop levels (traversing continuously through main memory)
  - Blocking sizes large enough to fulfill “layer condition”
  - Cache size is a hard limit!
  - Blocking loops may have some impact on ccNUMA page placement
3D Jacobi solver (problem size 400\(^3\))

Blocking different loop levels (8 cores Interlagos)

Cray Workshop
Performance for Multicore
3D Jacobi solver (problem size 400³)

Calculating the optimal block size (8 cores Interlagos)

- **Interlagos chip**: aggregate L2 size of 8 MB (say 4 MB to be safe)

- **Static OpenMP scheduling**: 0.5 MB cache per core

- **Layer condition with j-loop blocking**:
  
  2 layers of size \( N \times b_j \) must fit into the cache

  \[ 2 \times N \times b_j \times 8 \text{ byte} < 0.5 \text{ MB} \]

  \( \Rightarrow b_j < 78 \)
Jacobi iteration (3D): Nontemporal stores

- **Intel x86:** NT stores are packed SIMD stores with 16-byte aligned address
  - Sometimes hard to apply

- **AMD x86:** Scalar NT stores without alignment restrictions available

**Options for using NT stores**

- Let the compiler decide → unreliable
- Use compiler options
  - Intel: `-opt-streaming-stores never|always|auto`
- Use compiler directives
  - Intel: `!DIR$ vector [non]temporal`
  - Cray: `!DIR$ LOOP_INFO cache[_nt](...)`

- Compiler must be able to “prove” that the use of SIMD and NT stores is “safe”!
  - “line update kernel” concept: Make critical loop its own subroutine
Jacobi iteration (3D): Nontemporal stores for Cray

- **Line update kernel (separate compilation unit or -fno-inline):**

```fortran
subroutine jacobi_line(d,s,top,bottom,front,back,n)
    integer :: n,i,start
    double precision, dimension(*) :: d,s,top,bottom,front,back
    double precision, parameter :: oos=1.d0/6.d0
!DIR$ LOOP_INFO cache_nt(d)
    do i=2,n-1
        d(i) = oos*(s(i-1)+s(i+1)+top(i)+bottom(i)+front(i)+back(i))
    enddo
end subroutine
```

- **Main loop:**

```fortran
do joffset=1,jmax,jblock
    do k=1,kmax
        do j=joffset, min(jmax,joffset+jblock-1)
            call jacobi_line(phi(1,j,k,t1),phi(1,j,k,t0),phi(1,j,k-1,t0), &
                             phi(1,j,k+1,t0),phi(1,j-1,k,t0),phi(1,j+1,k,t0),
                             size)
        enddo
    enddo
enddo
```

Cray Workshop

Performance for Multicore
3D Jacobi solver

Spatial blocking + nontemporal stores

Cray Workshop

Performance for Multicore
Conclusions from the Jacobi optimization example

- “What part of the data comes from where” is a crucial question
- Avoiding slow data paths == re-establishing the layer condition
- Improved code showed the speedup predicted by the model
- Optimal blocking factor can be predicted
  - Be guided by the cache size the layer condition
  - No need for exhaustive scan of “optimization space”
Case study: OpenMP-parallel sparse matrix-vector multiplication

A simple (but sometimes not-so-simple) example for bandwidth-bound code and saturation effects in memory
Sparse matrix-vector multiply (spMVM)

- Key ingredient in some matrix diagonalization algorithms
  - Lanczos, Davidson, Jacobi-Davidson
- Important for sparse solvers (CG,...)

- Store only $N_{nz}$ nonzero elements of matrix and RHS, LHS vectors with $N_r$ (number of matrix rows) entries
- “Sparse”: $N_{nz} \sim N_r$

General case: some indirect addressing required!
CRS matrix storage scheme

- val[] stores all the nonzeros (length $N_{nz}$)
- col_idx[] stores the column index of each nonzero (length $N_{nz}$)
- row_ptr[] stores the starting index of each new row in val[] (length: $N_r$)
Case study: Sparse matrix-vector multiply

- Important kernel in many applications (matrix diagonalization, solving linear systems)
- Strongly memory-bound for large data sets
  - Streaming + partially indirect access:

```c
!$OMP parallel do
do i = 1,Nr
   do j = row_ptr(i), row_ptr(i+1) - 1
      c(i) = c(i) + val(j) * b(col_idx(j))
   enddo
endo
do i = 1,Nr
```

- Usually many spMVMs required to solve a problem
- Code balance / computational intensity? (erratic RHS access!)
- Saturation / scaling behavior?
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

Case 1: Large matrix

Intrasocket bandwidth bottleneck

Good scaling across sockets

Cant, 62451x62451, non-zero: 4007383

MFLOPS/s

0 1000 2000 3000 4000 5000 6000 7000 8000

0 5 10 15 20 25

threads
Application: Sparse matrix-vector multiply

Strong scaling on one XE6 Magny-Cours node

Case 1: Large matrix

Performance model???
Application: Sparse matrix-vector multiply
Strong scaling on one XE6 Magny-Cours node

Case 2: Medium size

- CRS-magnycours
  - mc2depi, 525825x525825, non-zero: 2100225

- Working set fits in aggregate cache

- Intrasocket bandwidth bottleneck
Application: Sparse matrix-vector multiply

Strong scaling on one Magny-Cours node

Case 3: Small size

No bandwidth bottleneck

Parallelization overhead dominates

Cray Workshop

Performance for Multicore
Roofline performance model for CRS spMVM

**Code balance for double precision FP and 4-byte index:**

\[
\begin{align*}
B(\alpha, N_{nzr}) &= \left( \frac{8 + 4}{2} \right) + \frac{8\alpha + 16/N_{nzr}}{2} \quad \text{bytes/flop} \\
&= \left( 6 + 4\alpha + \frac{8}{N_{nzr}} \right) \quad \text{bytes/flop}
\end{align*}
\]


Cray Workshop  Performance for Multicore  149
The $\alpha$ parameter

Corner case scenarios:

1. $\alpha = 0 \quad \rightarrow \text{RHS in cache}$
2. $\alpha = \frac{1}{N_{nzc}} \quad \rightarrow \text{Load RHS vector exactly once}$

If $N_{nzc} \gg 1$, RHS traffic is insignificant: 

$$P = \frac{b\beta}{6 \text{ bytes/flop}}$$

3. $\alpha \approx 1 \quad \rightarrow \text{Each RHS load goes to memory}$
4. $\alpha > 1 \quad \rightarrow \text{Houston, we’ve got a problem 😊}$

Determine $\alpha$ by measuring actual spMVM memory traffic (HPM)
Determine RHS traffic

$V_{meas}$ is the measured overall memory data traffic (using, e.g., likwid-perfctr)

Determine $\alpha$:

$$\alpha = \frac{1}{4} \left( \frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{8}{N_{nzc}} \right)$$

Example: kkt_power matrix on one Intel SNB socket

- $N_{nz} = 14.6 \cdot 10^6$, $N_{nzc} = 7.1$
- $V_{meas} \approx 258$ MB
- $\Rightarrow \alpha = 0.43$, $\alpha N_{nzc} = 3.1$
- $\Rightarrow$ RHS is loaded 3.1 times from memory
- and:

$$\frac{B_{CRS}^{DP}(\alpha)}{B_{CRS}^{DP}(1/N_{nzc})} = 1.15$$

15% extra traffic $\Rightarrow$ optimization potential!
Conclusions from the spMVM example

- spMVM shows “typical” bandwidth-bound scaling behavior
- Roofline is good for a first shot at modeling
- Deviations are to be expected
  - Erratic RHS access
  - Saturation bandwidth is lower than the maximum
- Deviations can be used to learn more about the code execution
  - How much excess memory traffic is generated from the indirect access?
Conclusions

There is no alternative to knowing what is going on between your code and the hardware

Without performance modeling, optimizing code is like stumbling in the dark