OpenMP* vs. OpenCL™: Performance Comparison

Sergei Vinogradov, Julia Fedorova, James Cownie - Intel

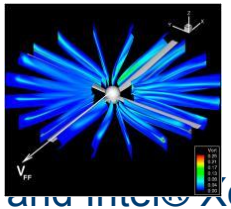
Dan Curran, Simon McIntosh-Smith - University of Bristol

OpenMPCon 28-30 SEPT, 2015 | RWTH AACHEN UNIVERSITY, GERMANY

Why OpenMP vs. OpenCL

- For HPC applications where performance is critical which programming model enables achieving optimal performance easier?
- Can’t prove an answer as a theorem but can experiment and learn contributing factors

- Guinea pigs
  - Benchmarks: Rodinia
  - Application: RotorSim



  - Hardware: Intel® Xeon® processor and Intel® Xeon Phi™ coprocessor

Agenda

- Environment Setup
- Rodinia Experiment
- RotorSim Experiment
- Conclusion

Environment Setup

- **Rodinia** benchmark suite developed by University of Virginia
  - Designed to benchmark heterogeneous systems
  - Equivalent implementation in OpenMP, OpenCL, CUDA* software technology

Device	# cores	# HW threads
Intel® Xeon® E5-2687W processor @ 3.10GHz	2 sockets 8 cores	32
Intel® Xeon Phi™ 7120P coprocessor @ 1.1 GHz (code-named Knights Corner)	61 cores	244

- **RotorSim** app developed by University of Bristol
  - Multi-grid CFD code

Device	# cores	# HW threads
Intel® Xeon® E5-2660 processor @ 2.2 GHz	2 sockets 10 cores	40
Intel® Xeon Phi™ 3120P coprocessor @ 1.1 GHz (code-named Knights Corner)	57 cores	228

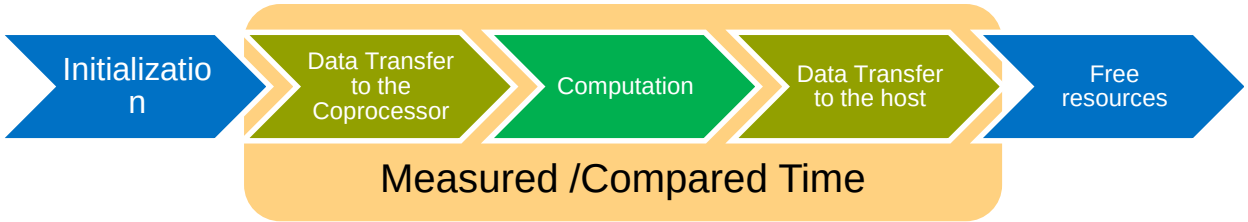
- Intel® C/C++ Compiler 15.0 with OpenMP 4.0 support
- GNU* C/C++ Compiler 4.9 with OpenMP 4.0 support
- Intel OpenCL SDK 1.2

Rodinia. Experiment Setup

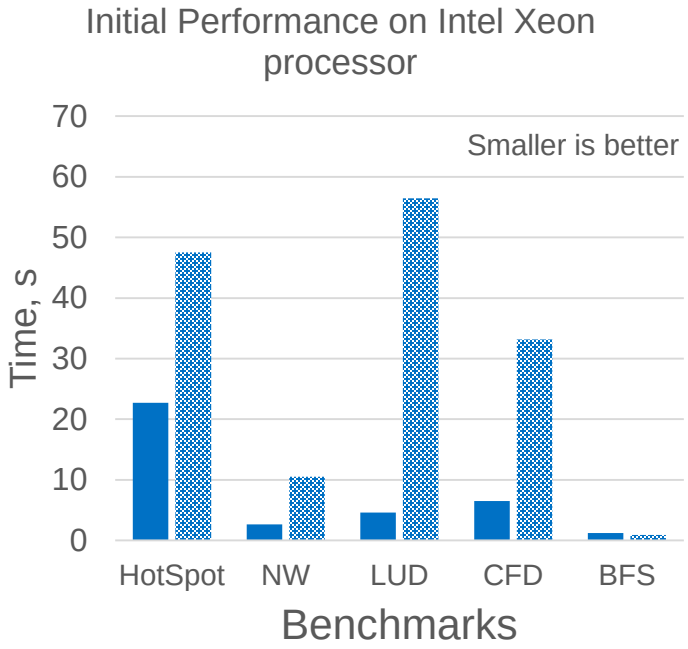
http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators

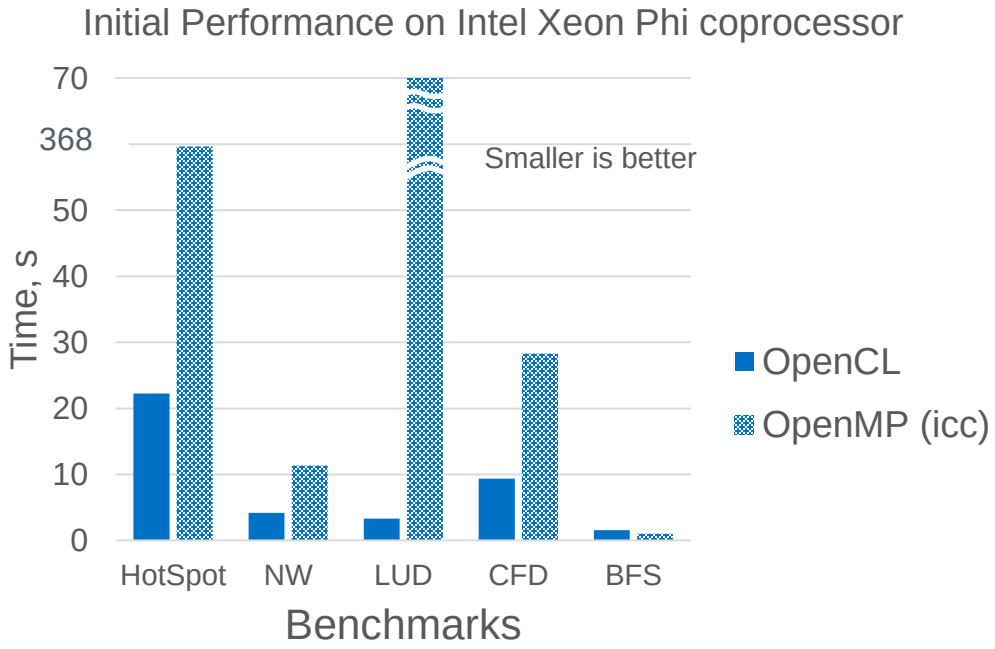
Analyzed 5 benchmarks: Hotspot, LUD, CFD, NW, BFS

- All use `gettimeofday()` function to capture timestamps
- Two sub-experiments for each benchmark
  - Computation done on the Intel Xeon processor
  - Computation offloaded to the Intel Xeon Phi coprocessor
- Time = $T_{\text{Data Transfer}} + T_{\text{Compute}}$ (i.e. time to solution of the problem)



Rodinia. Initial Results

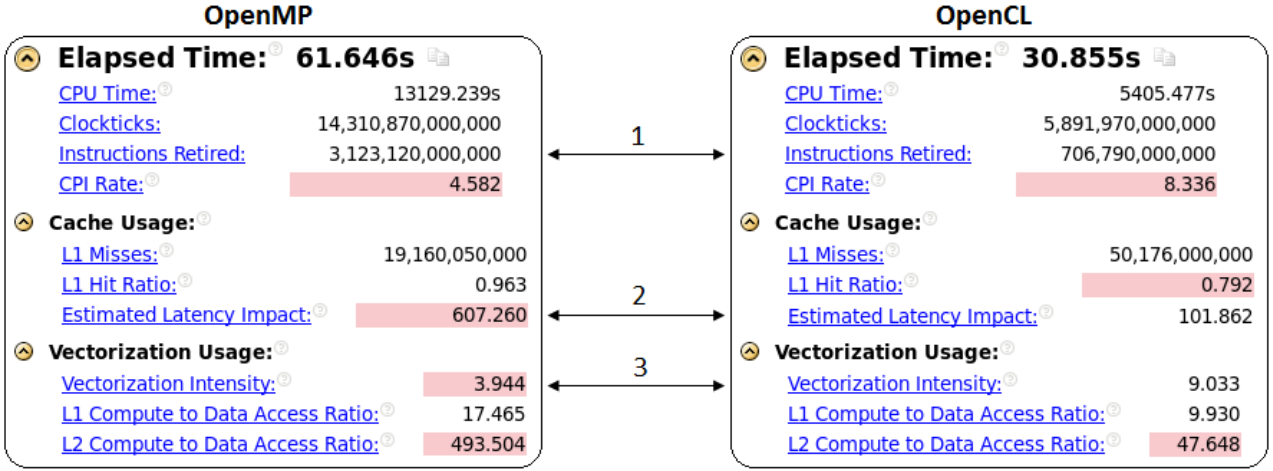




Initial OpenMP implementation is much slower. So we dive there…

HotSpot. Initial Performance Difference

Performance profile captured on Intel Xeon Phi Coprocessor by using Intel® VTune™ Amplifier XE

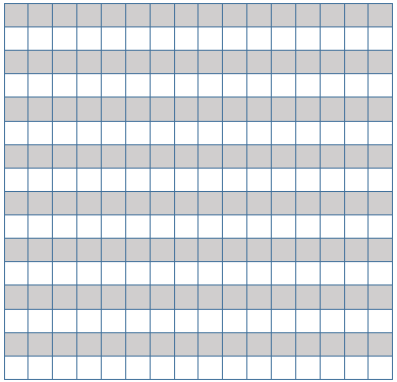


OpenMP version has

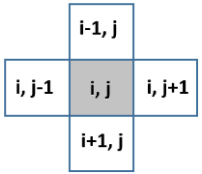
1. Much higher number of retired instructions
2. Higher Estimated Latency Impact (approximating the number of cycles taken to service L1 misses)
3. Poor vectorization (value of 16 is ideal for single precision on Intel Xeon Phi coprocessor)

HotSpot. Improving Cache Locality via Blocking

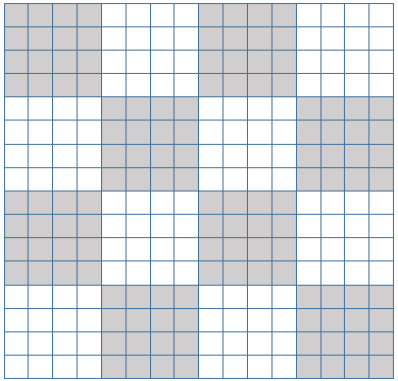
row-major work distribution in initial OpenMP version



Data access stencil



Optimized blocking work distribution in OpenCL version



We implemented blocking work distribution manually.

The new Intel C++ Compiler 16.0 offers `#pragma block_loop` that can do blocking automatically.

The optimal block size is 16 x 16 elements (the same as in OpenCL version):

1. A thread can process 16 elements in one vector instruction
2. Data required by four threads per core fits into L1 cache - 32Kb
   data size ~ 12KB = (16 x 16 x 4 (size of float) x 3 (number of arrays) x 4 (threads))

HotSpot. Enabling Vectorization

- OpenCL kernel codes are well vectorized by the OpenCL compiler using work-item & ND-range paradigms

- In the OpenMP version, some changes are required to make the compiler generate vectorized code :
  - Remove branches in an inner loop - process inner blocks separately from the border blocks
  - Use **`#pragma omp simd`** in an inner loop to make the compiler ignore assumed dependencies

HotSpot. Other Changes

- Ensure OpenCL and OpenMP versions are algorithmically equivalent
  e.g. use reciprocal and multiplication rather than division

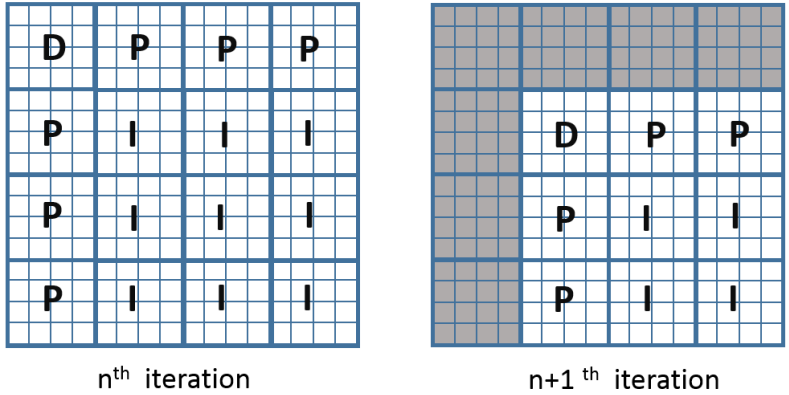
One difference:
- The OpenCL code enforces uniform processing of elements by introducing halo elements on the grid boundaries.
- The OpenMP implementation process boundaries blocks separately.

- Apply Compact Affinity to sequentially distribute the threads among the cores that share the same cache –
  leverage data reuse and locality from neighbor blocks on the same core
  `MIC_KMP_AFFINITY=compact`
- *Intel Xeon Phi coprocessor specific (default for OpenCL (Intel SDK)):*
  Instruct the offload runtime to allocate buffers greater than 64 kB into big (2MB) pages `MIC_USE_2MB_BUFFERS=64K`

LUD. Summary

- Big difference in performance between OpenCL and OpenMP versions: 3.3 s vs. 368 s (>110x!)
  - OpenMP implementation uses a naïve algorithm
  - OpenCL version implements a blocked LU factorization
- Implemented the same algorithm in OpenMP
  - Hotspot is the update of inner matrix marked as “I”
  - `#pragma omp simd` applied to the internal loops
  - Copied block data to a local array to enable a non-stride memory access

Sub-matrices in the blocked LU factorization algorithm and two iterations of the algorithm

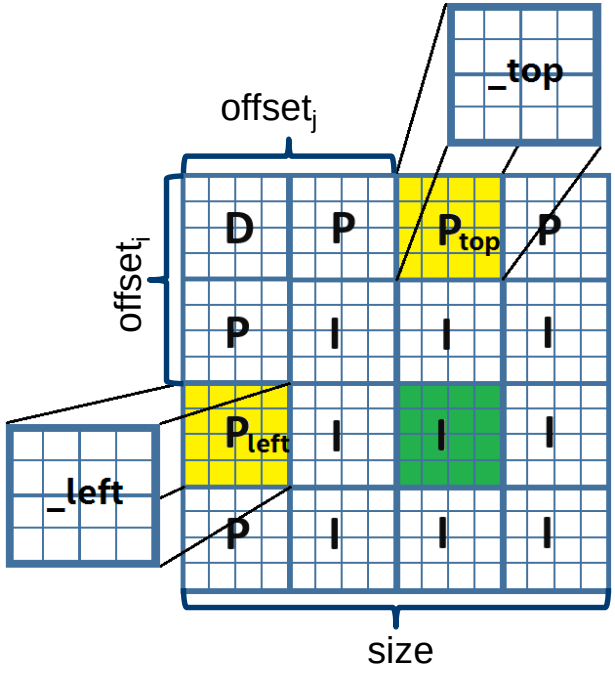


- *Intel Xeon Phi coprocessor specific:* using big pages: `MIC_USE_2MB_BUFFERS=64K`

LUD. Avoiding gather instructions

- **“I” block processing depends on Pleft and Ptop perimeter matrices**
  If access Pleft and Ptop elements from global matrix –
  ***gather* instructions generated – due to *size* unknown at compile time**
- **Copying Pleft and Ptop into local arrays allows to exchange *gathers* with *unit stride accesses***

```
#define BS 16 // block size

for(iter = 0; iter < (size-BS); ++iter) {
  #pragma parallel for
  for(…) //loop through "I" blocks
  {
    …
    float _left[BS*BS];
    float _top[BS*BS];

    // Copy perimeter elements to local arrays
    for (i = 0; i < BS; ++i) {
      #pragma omp simd
      for (j =0; j < BS; ++j) {
        _top[i*BS + j]  = a[size*(i + iter*BS) + j + offset_j ];
        _left[i*BS + j] = a[size*(i + offset_i) + iter*BS + j];
      }
    }
    …
  }
}
```



LUD. Avoiding Costly Reduction

```
#define BS 16 // block size
float _left[BS*BS];
float _top[BS*BS];

… // Copying data into local arrays _left & _top
  // to avoid costly gathers instructions

float sum;
for (i = 0; i < BS; ++i)
{
  for (j = 0; j < BS; ++j)
  {
    sum = 0.f;
#pragma omp simd reduction(+:sum)
    for (k = 0; k < BS; ++k)
    {
      sum += _left[BS*i + k] * _top[BS*j + k];
    }
    BB((i+i_offset),(j+j_offset)) -= sum;
  }
}
```

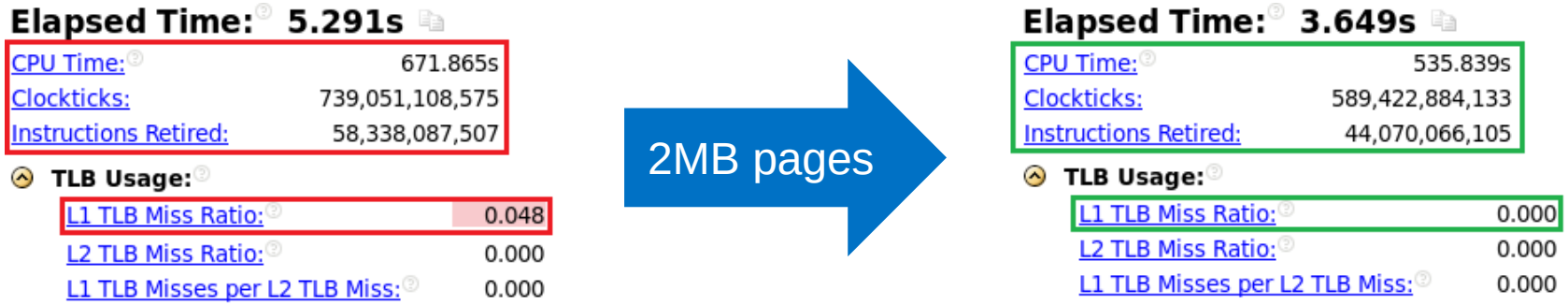
k-loop is completely unrolled and vectorized, but this causes a costly reduction from a vector result of multiply

```
#define BS 16 // block size
float _left[BS*BS];
float _top[BS*BS];

… // Copying data into local arrays _left & _top
  // to avoid costly gathers instructions

float sum[BS] ={0.f};
for (i = 0; i < BS; ++i)
{
  for (k=0; k < BS; ++k)
  {
#pragma omp simd
    for (j = 0; j < BS; ++j)
    {
      sum[j] += _left[BS*i + k] * _top[BS*k + j];
    }
  }
#pragma omp simd
  for (j = 0; j < BS; ++j) {
    BB((i+i_offset),(j+j_offset)) -= sum[j];
  }
}
```

Both j-loops are completely unrolled and vectorized

LUD. Use Big Pages for Intel Xeon Phi Coprocessor

- Intel Xeon Phi coprocessor specific:
  - Set MIC_USE_2MB_BUFFERS=64K environment to
  - The OpenCL runtime used huge pages by default



CFD. Summary

Compiler does not vectorize the loop annotated with `OMP parallel for`

```
#pragma omp parallel for
for (int i = 0; i < nelr; ++i)
{
      …
}
```

enable vectorization

```
#pragma omp parallel for
for (blk = 0; blk < nelr/block_size; ++blk)
{
      int b_start = blk*block_length;
      int b_end = (blk+1)*block_length;

#pragma omp simd
    for(int i = b_start; i < b_end; ++i)
    {
        ...
    }

}
```

Another big difference between OpenCL and OpenMP versions:

OpenMP: Array of Structures

- Natural to program
- But, leads to non-unit stride memory access when vectorized

OpenCL: Structure of Arrays

- Unit-stride accesses and vectorization

NW. Summary

OpenCL version implements a blocking algorithm to traverse the data

Implemented the same blocking algorithm in the OpenMP version

- Copy each block to a local array before doing calculations
- Apply `#pragma omp simd` to the inner loops of the parallel region
- `MIC_KMP_AFFINITY=compact` used for better data locality

Intel Xeon Phi coprocessor specific: using big pages to decrease TLB misses

BFS. Summary

Breadth-First Search algorithm contains two kernels:

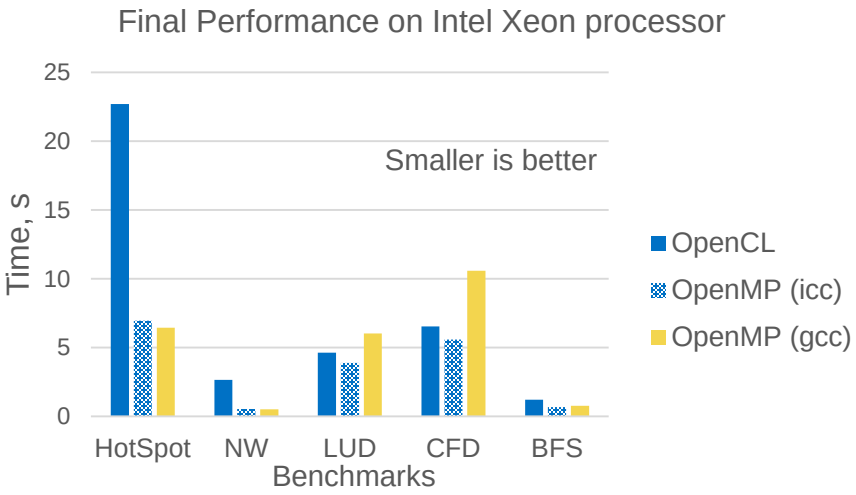
- Visits all nodes on the current layer to find non-visited children and update the corresponding costs.
- Marks visited nodes and sets a mask for the nodes that should be visited on the next layer.

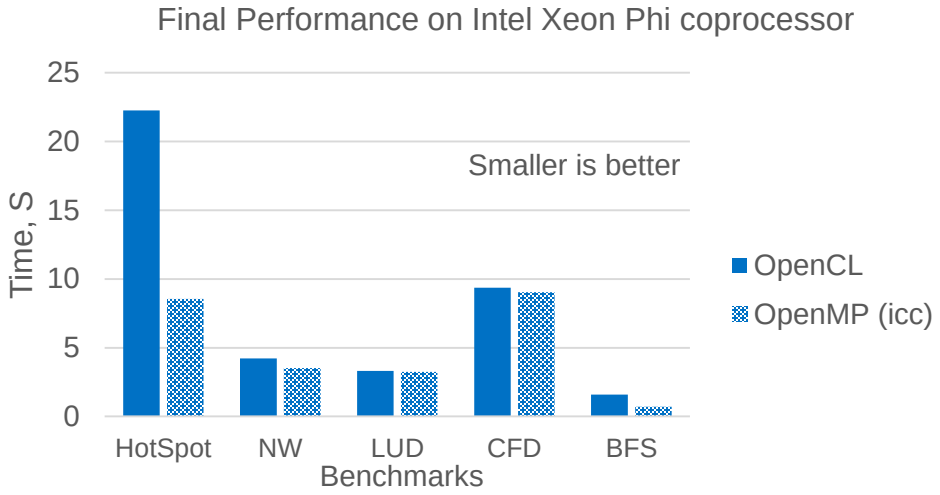
The second kernel was not run(?!) in parallel in the original OpenMP version

- simply added `#pragma omp parallel for` declaration

Intel Xeon Phi coprocessor specific: using big pages to decrease TLB misses

Rodinia. Final Results





- The OpenMP version performance is on par or better than the OpenCL version performance
  - *We did our best to optimize OpenMP versions, but keep as are the OpenCL ones. We believe the OpenCL versions could be optimized more.*
- The difference for HotSpot performance is due to the boost from the affinity enabled in the OpenMP version. And OpenCL implementation based on halo elements is less efficient than the optimized OpenMP code.
  - *The OpenCL implementation we used does not provide an affinity control out-of-the-box like OpenMP (*OMP_PROC_BIND=true). *So we tried creating sub-devices on different cores (through extension), assigning a queue to each of them and then synchronizing queues on the host at each iteration - but the synchronization kills the performance*

Performance-Sensitive Factors

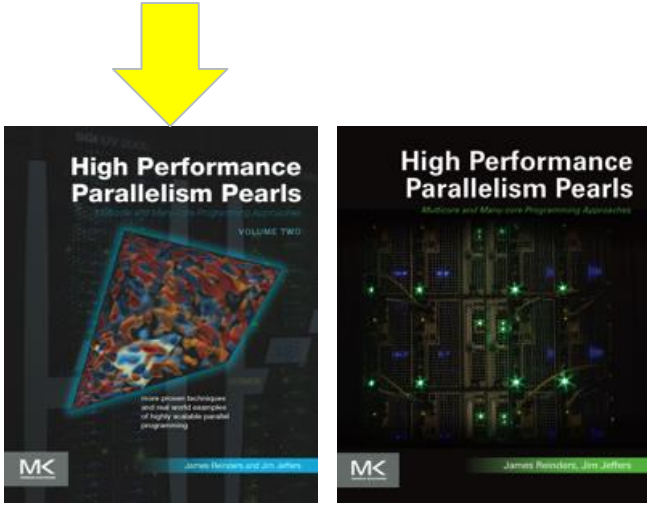
- Same algorithm
- Dynamic instruction count
- Memory access pattern
  - Locality
  - Avoiding TLB misses
  - Threads affinity to benefit from data reuse and locality
- Vectorization
- Right parallel task granularity

easier with OpenMP

easier with OpenCL

As soon as OpenMP and OpenCL implementations have similar characteristics, their performance is similar

Rodinia. We are chapter in the book



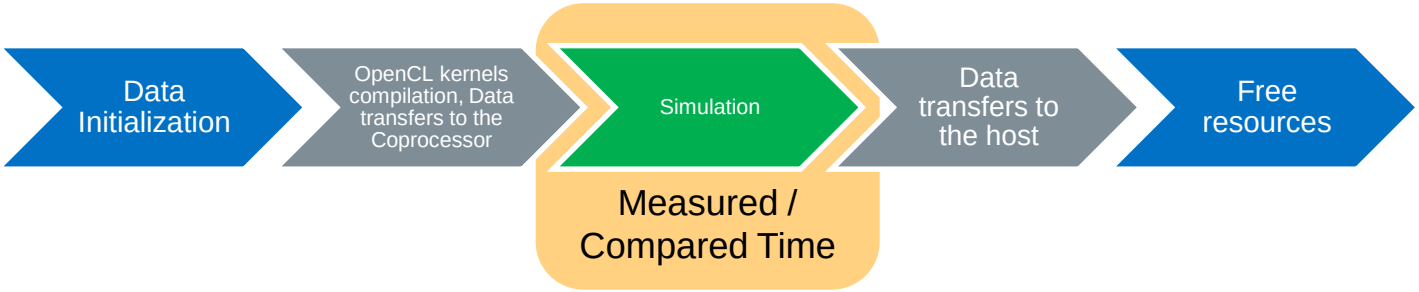
Source code available at
http:// lotsofcores.com

Volume 2 includes the following chapters:

Foreword by Dan Stanzione, TACC
Chapter 1: Introduction
Chapter 2: Numerical Weather Prediction Optimization
Chapter 3: WRF Goddard Microphysics Scheme Optimization
Chapter 4: Pairwise DNA Sequence Alignment Optimization
Chapter 5: Accelerated Structural Bioinformatics for Drug Discovery
Chapter 6: Amber PME Molecular Dynamics Optimization
Chapter 7: Low Latency Solutions for Financial Services
Chapter 8: Parallel Numerical Methods in Finance
Chapter 9: Wilson Dslash Kernel From Lattice QCD Optimization
Chapter 10: Cosmic Microwave Background Analysis: Nested Parallelism In Practice
Chapter 11: Visual Search Optimization
Chapter 12: Radio Frequency Ray Tracing
Chapter 13: Exploring Use of the Reserved Core
Chapter 14: High Performance Python Offloading
Chapter 15: Fast Matrix Computations on Asynchronous Streams
Chapter 16: MPI-3 Shared Memory Programming Introduction
Chapter 17: Coarse-Grain OpenMP for Scalable Hybrid Parallelism
Chapter 18: Exploiting Multilevel Parallelism with OpenMP
Chapter 19: OpenCL: There and Back Again
Chapter 20: OpenMP vs. OpenCL: Difference in Performance?
Chapter 21: Prefetch Tuning Optimizations
Chapter 22: SIMD functions via OpenMP
Chapter 23: Vectorization Advice
Chapter 24: Portable Explicit Vectorization Intrinsics
Chapter 25: Power Analysis for Applications and Data Centers

RotorSim. Experiment Setup

- Two sub-experiments for each benchmark
  - Computation done on Intel Xeon processor
  - Computation offloaded to Intel Xeon Phi coprocessor
- The application measures
  Simulation time by timestamps from `gettimeofday()` function
- Main comparison - by Simulation time



Benchmark Workload

- 12 blocks, each has 96x32x64 grid

- Selected size is a compromise between
  - having enough parallelism (the bigger the better)
  - fitting into Intel Xeon Phi coprocessor memory (limited from the top by ~ 5 GB) and specifics of the implementations

So, we can compare Intel Xeon processor and Intel Xeon Phi coprocessor code behavior and performance

RotorSim Changes

- OpenCL version was already functional when we started, so we worked on the OpenMP version

- OpenMP 4.0 features used:

  #pragma offload target(mic) in(…), out(…) *-> to send data to/from the coprocessor*

  __attribute__ ((target(mic)) *-> to compile functions for the coprocessor*

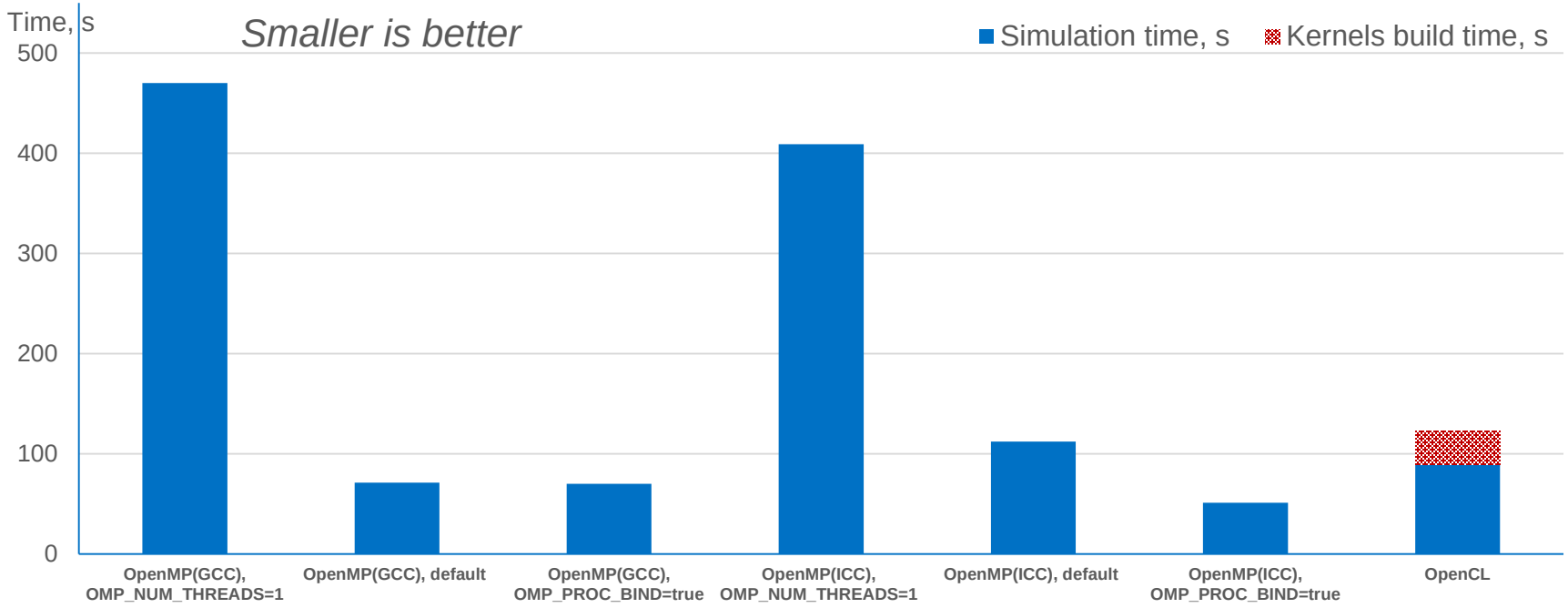
  #pragma omp simd *-> to vectorize hot loops*

- Other OpenMP features used:

  collapse (used collapse(2)) *-> to ensure good granularity*

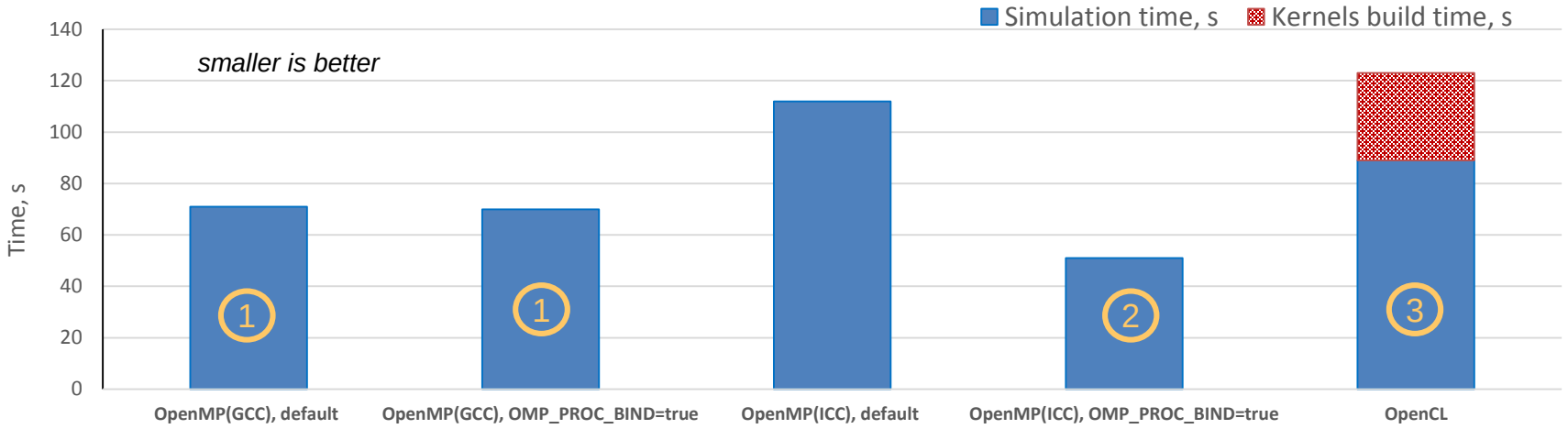
  OMP_PROC_BIND=true *-> to enforce NUMA friendly memory accesses via affinity, as a working set is > L3*

RotorSim on Intel Xeon Processor. Performance Overview



RotorSim on Intel Xeon Processor.

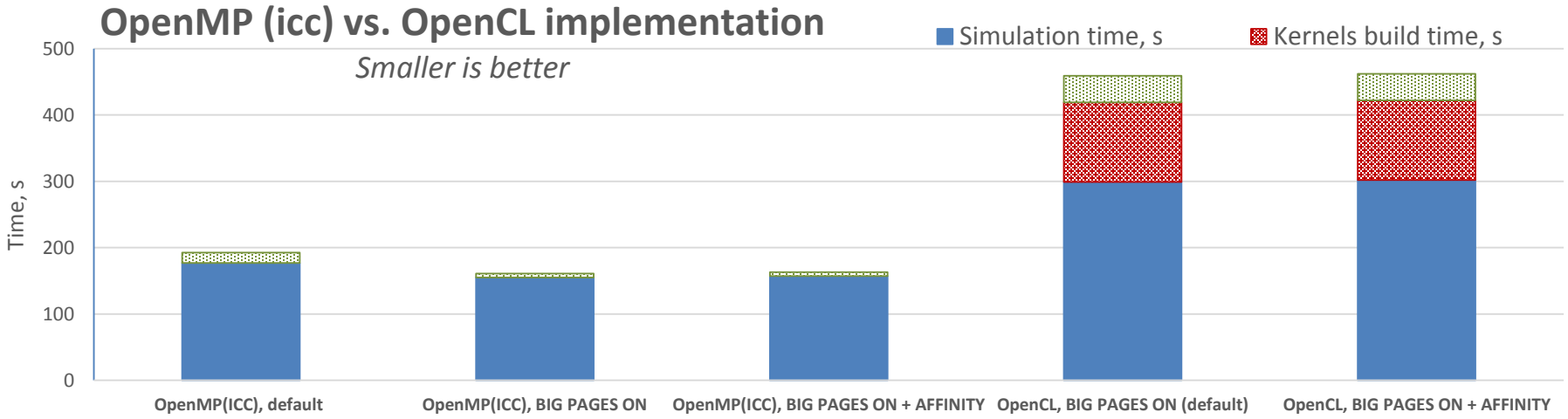
Comparison of parallel implementations performance



1. Performance of the code based on OpenMP (gcc) varies. The best value is referred here.
2. With affinity (respecting NUMA), the OpenMP (icc) based code achieves the highest performance.
3. OpenCL based code has a stable performance less than gcc and icc versions of OpenMP respecting affinity. Based on the Rodinia study, implementing OpenCL code with NUMA affinity doesn't look feasible at the moment

NUMA aware placement improves performance

RotorSim. Performance of parallel versions on Intel Xeon Phi Coprocessor



- Negligible sensitivity to affinity on the Intel Xeon Phi coprocessor
- Speed-up with using big pages, specific to Intel MIC architecture (TLB)
- Per our limited analysis:
  The OpenMP version performance outperforms the OpenCL version due to using a smaller memory footprint;
  The Open CL version memory footprint is big due to implementation specifics

RotorSim. Application Performance Challenges

- Can't afford to redesign the application from scratch
- Easy debugging / ensuring correctness is #1
- Manual optimization is practical on the limited number of hot code regions (if at all)
  For wider coverage, rely on programming model "knobs"
  for good granularity, vectorization, memory accesses, etc.

You might not achieve a peak but reasonably good stable performance is guaranteed.

- In our case, the OpenMP implementation worked well
  - Debugging was doable:
    fixed bugs in the OpenMP version and enabled offload within ~5 weeks of calendar time
  - **Performance on Intel Xeon processor:** OpenMP affinity helps outperform the OpenCL version
  - **Performance on Intel Xeon Phi coprocessor:** the OpenMP version has a smaller memory footprint vs. the OpenCL version (due to the buffer handling in the OpenCL implementation of Rotorsim)
    This seems to be the main reason for performance difference. Need to investigate more

Conclusion

- We don’t have an answer which programming model is better
  - Our experiments achieved higher performance with the OpenMP version… mostly due to more efficient memory accesses (through affinity), we believe.
  - We tried to apply affinity to OpenCL code on Intel Xeon processor but unsuccessfully - there is no feasible solution at the moment
- OpenCL offers out-of-the-box vectorized code and efficient memory accesses on a small scale
- OpenMP requires more efforts for good vectorization, but provides a good control for memory accesses on a big scale and easier to develop and debug applications

Acknowledgement

We wish to acknowledge the following persons:

- Andrey Churbanov for his valuable technical support on optimizing the OpenMP version of RotorSim application.
- Paul Petersen and Larry Meadows for their helps in preparing this presentation.
- And many other reviewers.

Questions?
Thank you!

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804