C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators
Power-Aware Computing – Performance of Application Kernels

hyPACK-2013
Mode 3 : Intel Xeon Phi Coprocessors

Lecture Topic :
Intel Xeon-Phi : Tuning & Performance

***Venue** : CMSD, UoHYD ; **Date** : October 15-18, 2013*

An Overview of Intel Xeon Phi – Tuning & Perf.

Lecture Outline

Following topics will be discussed

- Understanding of Intel Multi-Core Systems with Intel Xeon Phi Programming from Performance Point of View

Intel Xeon Phi Coprocessor : Prog. Env & Tips for obtaining Performance (Part-I)

Xeon Phi : Programming Environment

- **Shared Address Space Programming (Offload, Native, Host)**

  OpenMP, Intel TBB, Cilk Plus, Pthreads

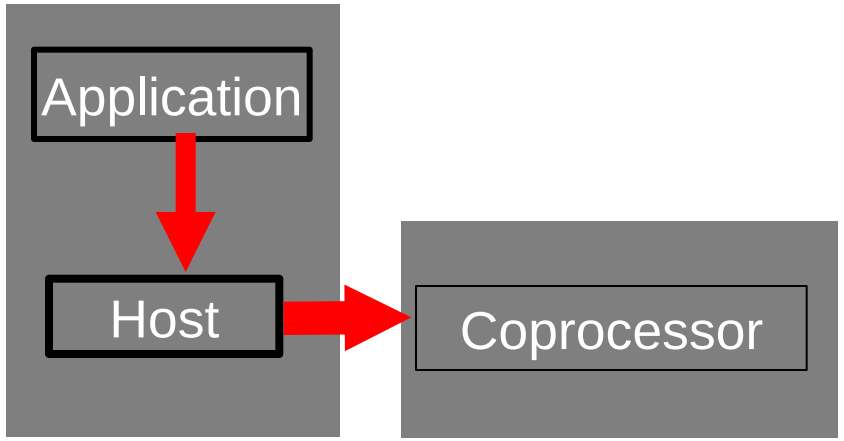
- **Message Passing Programming**

  (Offload – MIC Offload /Host Offload)

  (Symmetric & Coprocessor /Host)

- **Hybrid Programming**

  (MPI – OpenMP, MPI Cilk Plus
  MPI-Intel TBB)



Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon-Phi : Shared Address Space Prog.

- **Rule of thumb :** An application must scale well past one hundred threads on Intel Xeon processors to profit from the possible higher parallel performance offered with e.g. the Intel Xeon Phi coprocessor.

- The scaling would profit from utilizing the highly parallel capabilities of the MIC architecture, you should start to create a simple performance graph with a varying number of threads (from one up to the number of cores)

Intel Xeon-Phi : Shared Address Prog.

- **What we should know from programming point of view :** We treat the coprocessor as a 64-bit x86 **SMP-on-a-chip** with an high-speed bi-directional **ring** interconnect, (up to) **four** hardware threads per core and **512-bit SIMD** instructions.

- With the available number of cores, we have easily 200 hardware threads at hand on a single Intel Xeon coprocessor.

- Resource availability and Memory access is an important for threading on all 60 Cores.

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon-Phi : Programming Env.

Keys to Productive Performance

- Choose the right Multi-core centric or Many-core centric model for your application
- Vectorize your application (today)
  - Use the Intel vectorizing compiler
- Parallelize your application (today)
  - with MPI (or other multi-process model)
  - With threads (via Intel ® Cilk TM Plus, OpenMP*, Intel ® Threading Building Blocks, Pthreads, etc.)
- Go asynchronous to overlap computation and communication

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon-Phi : Performance-Tips

Performance on Xeon Phi using different prog.

- **What we should know from programming point of view :** We treat the coprocessor as a 64-bit x86 **SMP-on-a-chip** with an high-speed bi-directional **ring** interconnect, (up to) **four** hardware threads per core and **512-bit SIMD** instructions.

- With the available number of cores, we have easily 200 hardware threads at hand on a single coprocessor.

Intel Xeon System & Xeon-Phi

Performance on Xeon Phi using different prog.

About Hyper-Threading

- hyper-threading hardware threads can be switched off and can be ignored.

About Threading on Xeon-Phi Coprocessor

- The multi-threading on each core is primarily used to hide latencies that come implicitly with an in-order microarchitecture. Unlike hyper-threading these hardware threads cannot be switched off and should never be ignored.
- In general a minimum of **three** or **four** active threads per cores will be needed.

Summary: Tricks for Performance

Performance on Xeon Phi using different prog.

- Use asynchronous data transfer and double buffering offloads to overlap the communication with the computation
- Optimizing memory use on Intel MIC architecture target relies on understanding access patterns
- Loop Optimizations may benefit performance

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi Coprocessor :Native Compilation

To achieve good Performance - Following information should be kept in mind.

- Data should be **aligned to 64 Bytes (512 Bits)** for the MIC architecture, in contrast to 32 Bytes (256 Bits) for AVX and 16 Bytes (128 Bits) for SSE.
- Due to the large SIMD width of 64 Bytes **vectorization is even more important for the MIC architecture than for Intel Xeon!**
- The MIC architecture offers **new instructions** like
  - **gather/scatter,**
  - **fused multiply-add,**
  - **masked vector instructions etc.**

  which allow more loops to be parallelized on the coprocessor than on an **Intel Xeon based host**.

Intel Xeon Phi Coprocessor : Native Compilation

To achieve good Performance - Following information should be kept in mind.

Use pragmas like

- `#pragma ivdep,`
- `#pragma vector always,`
- `#pragma vector aligned,`
- `#pragma simd`

etc. to achieve autovectorization.

Autovectorization is enabled at default optimization level `-O2`. Requirements for vectorizable loops can be found references.

Intel Xeon Phi Coprocessor : Native Compilation

To achieve good Performance - Following information should be kept in mind.

- Let the compiler generate vectorization reports using the compiler option **`-vecreport2`** to see if loops were vectorized for MIC (Message "*MIC* Loop was vectorized" etc).

- The options –**`opt-report-phase hlo`** (High Level Optimizer Report) or

  –**`opt-report-phase ipo_inl`** (*Inlining* report) may also be useful.

Intel Xeon Phi Coprocessor :Native Compilation

To achieve good Performance - Following information should be kept in mind.

- Explicit vector programming is also possible via Intel Cilk Plus language extensions (C/C++ array notation, vector elemental functions, ...) or the new SIMD constructs from OpenMP 4.0 RC1.
- Vector elemental functions can be declared by using `__attributes__((vector))`. The compiler then generates a vectorized version of a scalar function which can be called from a vectorized loop.

Intel Xeon Phi Coprocessor : Native Compilation

To achieve good Performance - Following information should be kept in mind.

- One can use intrinsics to have full control over the vector registers and the instruction set.
- Include `<immintrin.h>` for using intrinsics.
- **Hardware prefetching** from the L2 cache is enabled per default.
- In addition, **software prefetching** is on by default at compiler optimization level `-O2` and above. Since Intel Xeon Phi is an `inorder` architecture, care about prefetching is more important than on `out-of-order` architectures.

Intel Xeon Phi Coprocessor : Native Compilation

To achieve good Performance - Following information should be kept in mind.

- The compiler prefetching can be influenced by setting the compiler switch **`-opt-prefetch = n`**.

- Manual prefetching can be done by using intrinsics **`(_mm_prefetch`**`()` or

  pragmas (#**`pragma prefetch var`**).

Intel Xeon Phi Coprocessor : Prog. Env & Tips for obtaining Performance (Part-II)

Optimization Framework

A collection of methodology and tools that enable the developers to express parallelism for Multicore and Manycore Computing

Objective: Turning unoptimized program into a scalable, highly parallel application on multicore and manycore architecture

Step 1: Leverage Optimized Tools, Library

Step 2: Scalar, Serial Optimization /Memory Access

Step 3: Vectorization & Compiler

Step 4: Parallelization

Step 5: Scale from Multicore to Manycore

Source : References & Intel Xeon-Phi; http://www.intel.com/

A Family of Parallel Programming Models Developer Choice

Step 1 :

Intel® Cilk™ Plus	Intel® Threading Building Blocks	Domain-Specific Libraries	Established Standards	Research and Development
C/C++ language extensions to simplify parallelism	Widely used C++ template library for parallelism	Intel® Integrated Performance Primitives	Message Passing Interface (MPI)	Intel® Concurrent Collections
			OpenMP* (offload TR coming "real soon")	Intel® Offload Extensions
		Intel® Math Kernel Library		Intel® SPMD Parallel Compiler (ispc)
			Coarray Fortran	
Open sourced	Open sourced		OpenCL*	
Also an Intel product	Also an Intel product			

Applicable to Multicore and Manycore Programming

Source : References & Intel Xeon-Phi; http://www.intel.com/

Objective of Scalar and Serial Optimization

Step 2 :

- Obtain the most efficient implementation for the problem at hand
- Identify the opportunity for vectorization and parallelization
- Create Base to account for vectorization and parallelization gains
  - Avoid situation when vectorized, slower code was parallelized and create a false impression of performance gain

Source : References & Intel Xeon-Phi; http://www.intel.com/

Algorithmic Optimizations

- Elevate constants out of the core loops
  - Compiler can do it, but it need your cooperation
  - Group constants together
- Avoid and replace expensive operations
  - divide a constant can usually be replace by multiplying its reciprocal
- Strength reduction in hot loop
  - People like inductive method, because it's clean
  - Iterative can strength reduce the operation involved
  - In this example, exp() is replace by a simple multiplication

```
const double      dt = T / (double)TIMESTEPS;
    const double     vDt = V * sqrt(dt);
    for(int i = 0; i <= TIMESTEPS; i++){
     double price = S * exp(vDt * (2.0 * i -
                TIMESTEPS));
          cell[i] = max(price - X, 0);
                    }
```

```
const double factor = exp(vDt * 2);
  double        price = S * exp(-
        vDt(2+TIMESTEPS));
for(int i = 0; i <= TIMESTEPS; i++){
        price = factor * price;
     cell[i] = max(price - X, 0);
                }
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Use Compiler Optimization Switches

Optimization Done	Linux*
Disable optimization	-O0
Optimize for speed (no code size increase)	-O1
Optimize for speed (default)	-O2
High-level loop optimization	-O3
Create symbols for debugging	-g
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen -prof-use
Optimize for speed across the entire program	-fast (same as: -ipo –O3 -no-prec-div -static -xHost)
OpenMP 3.0 support	-openmp
Automatic parallelization	-parallel

Source : References & Intel Xeon-Phi; http://www.intel.com/

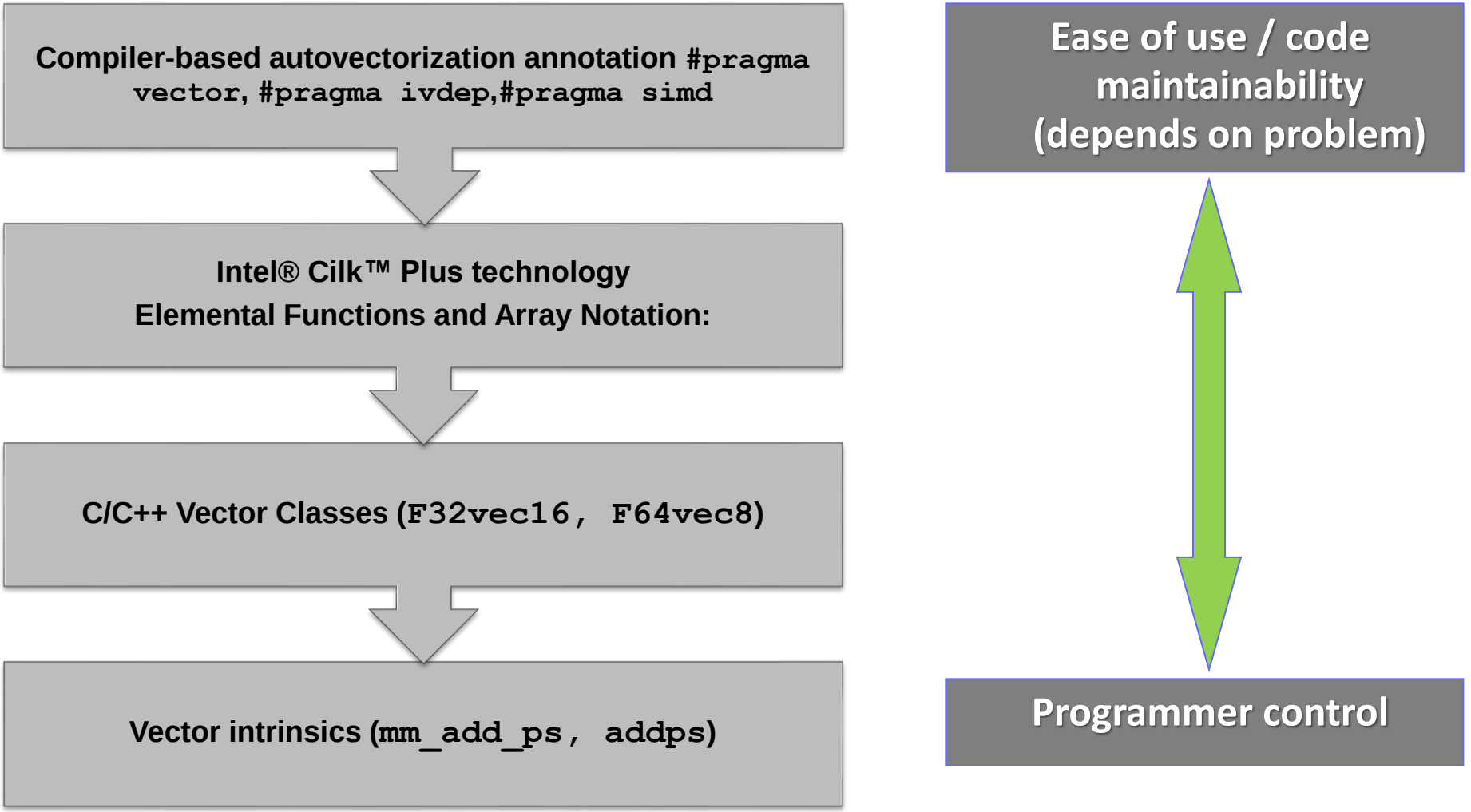
Vectorization and SIMD Execution

Step 3 :

- **SIMD**
  - Flynn's Taxonomy: Single Instruction, Multiple Data
  - CPU perform the same operation on multiple data elements
- **SISD**
  - Single Instruction, Single Data
- **Vectorization**
  - In the context of Intel® Architecture Processors, the process of transforming a scalar operation (SISD), that acts on a single data element to the vector operation that that act on multiple data elements at once(SIMD).
  - Assuming that setup code does not tip the balance, this can result in more compact and efficient generated code
  - For loops in "normal" or "unvectorized" code, each assembly instruction deals with the data from only a single loop iteration

Source : References & Intel Xeon-Phi; http://www.intel.com/

SIMD Abstraction – Options Compared



Source : References & Intel Xeon-Phi; http://www.intel.com/

Get Your Code Vectorized by Intel Compiler

- Data Layout, AOS -> SOA
- Data Alignment (next slide)
- Make the loop innermost
- Function call in treatment
  - Inline yourself
  - inline! Use `__forceinline`
  - Define your own vector version
  - Call vector math library - SVML
- Adopt jumpless algorithm
- Read/Write is OK if it's continuous
- Loop carried dependency

Array of Structures		
S0	X0	T0
S1	X1	T1
…	…	…

Structure of Arrays		
S0	S1	…
X0	X1	…
S0	S1	…

Not a true dependency

```
for(int i = TIMESTEPS; i > 0; i--)
        #pragma simd
        #pragma unroll(4)
  for(int j = 0; j <= i - 1; j++)
cell[j]=puXDf*cell[j+1]+pdXDf*cell[j];
CallResult[opt] = (Basetype)cell[0];
```

A true dependency

```
for (j=1; j<MAX; j++)
a[j] = a[j] + c * a[j-n];
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Options for Parallelism on Intel® Architecture

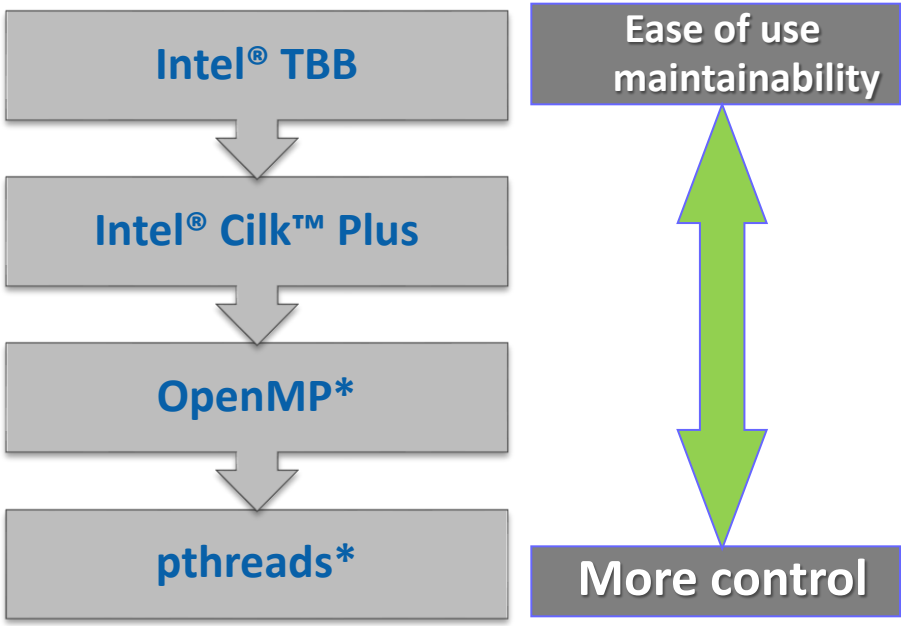
Step 4 :

- C++ template Library of parallel algorithms, containers
- Load balancing via work stealing

- Keyword extension of C/C++, Serial equivalence via compiler
- Load balancing via work stealing

- Well known industry standard
- Best suited when resource utilization is known at design time

- Time-tested industry standard for Unix-like
- Common denominator or other high level threading libraries



- What's available on Intel® host processor are also available on Intel® target coprocessor
- Many others (boost) are ported to the coprocessor
- Choose the best threading model your problem dictates

Source : References & Intel Xeon-Phi; http://www.intel.com/

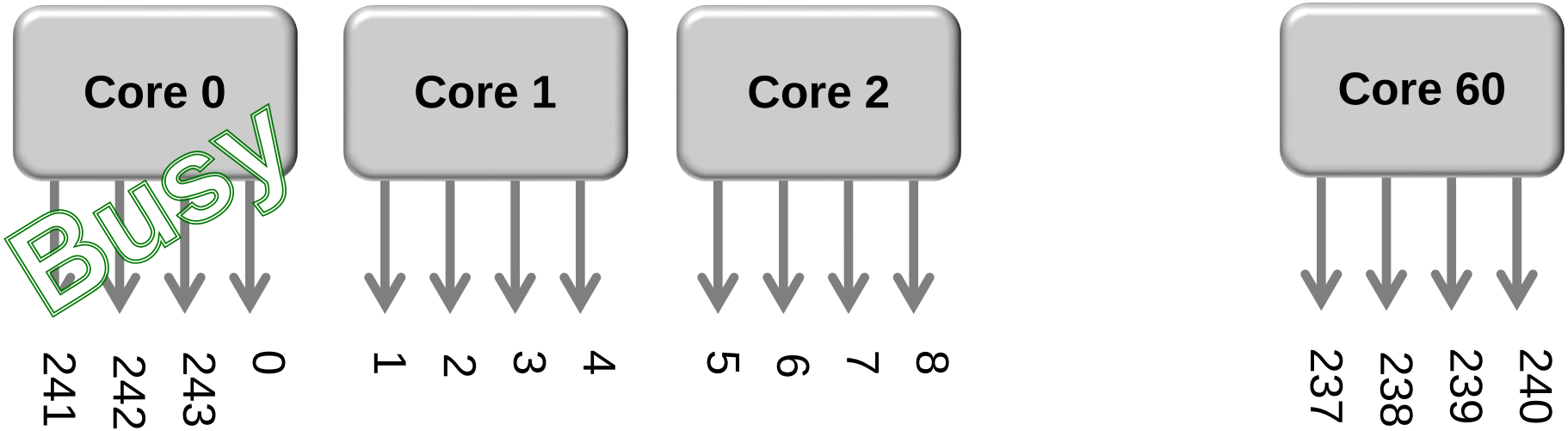
Options for Parallelism – pthreads*

- POSIX* Standard for thread API with 20 years history
- Foundation for other high level threading libraries
- Independently exist on the host and Intel® MIC
- No extension to go from the host to Intel® MIC
- Advantage: Programmer has explicit control
  - From workload partition to thread creation, synchronization, load balance, affinity settings, etc.
- Disadvantage: Programmer has too much control
  - Code longevity
  - Maintainability
  - Scalability

Source : References & Intel Xeon-Phi; http://www.intel.com/

Thread Affinity using pthreads*

- Partition the workload to avoid load imbalance
  - Understand static vs. dynamic workload partition
- Use pthread API, define, initialize, set, destroy
  - Set CPU affinity with `pthead_setaffinity_np()`
  - Know the thread enumeration and avoid core 0
  - Core 0 boots the coprocessor, job scheduler, service interrupts

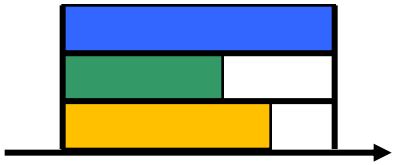


Source : References & Intel Xeon-Phi; http://www.intel.com/

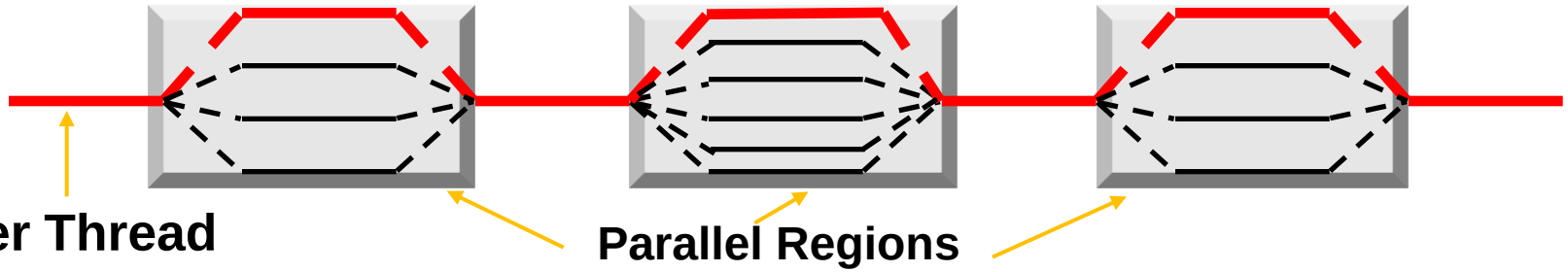
Options for Parallelism – OpenMP*

- Compiler directives/pragmas based threading constructs
  - Utility library functions and Environment variables
- Specify blocks of code executing in parallel

```
#pragma omp parallel sections
{
    #pragma omp section
    task1();
    #pragma omp section
    task2();
    #pragma omp section
    task3();
```



- Fork-Join Parallelism:
  - Master thread spawns a team of worker threads as needed
  - Parallelism grow incrementally



Source : References & Intel Xeon-Phi; http://www.intel.com/

OpenMP* Performance, Scalability Issues

- Manage Thread Creation Cost
  - Create threads as early as possible, Maximize the work for worker threads
  - IA threads take some time to create, But once they're up, they last till the end
- Take advantage of memory locality, use NUMA memory manager
  - Allocate the memory on the thread that will access them later on.
  - Try not to allocate the memory the worker threads use in the main thread
- Ensure your OpenMP* program works serially, compiles without openmp*
  - Protect OpenMP* API calls with _OPENMP,
  - Make sure serial works before enable OpenMP* (e.g. compile with –openmp)
- Minimize the thread synchronization
  - use local variable to reduce the need to access global variable

Source : References & Intel Xeon-Phi; http://www.intel.com/

```
#pragma omp parallel for
for (int k = 0; k < RAND_N; k++)
   h_Random[k] = cdfnorminv ((k+1.0)/(RAND_N+1.0));

#pragma omp parallel for
for(int opt = 0; opt < OPT_N; opt++)
{
   CallResultList[opt]     = 0;
   CallConfidenceList[opt] = 0;
}
```

```
#pragma omp parallel
{
   #ifdef _OPENMP
   int threadID = omp_get_thread_num();
   #else
   int threadID = 0;
   #endif

   float *CallResult = (float *) scalable_aligned_malloc
                                  (mem_size, SIMDALIGN);
   float *PutResult  = (float *) scalable_aligned_malloc
                                  (mem_size, SIMDALIGN);

}
```

```
#ifdef _OPENMP
int ThreadNum = omp_get_max_threads();
omp_set_num_threads(ThreadNum);
#else
int ThreadNum = 1;
#endif
```

Scale from Multicore to Manycore

Step 5 :

A Tale of Two Architectures

	Intel® Xeon® processor	Intel® Xeon Phi™ Coprocessor
Sockets	2	1
Clock Speed	2.6 GHz	1.1 GHz
Execution Style	Out-of-order	In-order
Cores/socket	8	Up to 61
HW Threads/Core	2	4
Thread switching	HyperThreading	Round Robin
SIMD widths	8SP, 4DP	16SP, 8DP
Peak Gflops	692SP, 346DP	2020SP, 1010DP
Memory Bandwidth	102GB/s	320GB/s
L1 DCache/Core	32kB	32kB
L2 Cache/Core	256kB	512kB
L3 Cache/Socket	30MB	none

Source : References & Intel Xeon-Phi; http://www.intel.com/

Assessing potential

- **Threads**
  - Code analysis – loop nesting, iteration counts, determinism
  - Intel Vtune™ Amplifier timeline analysis – existence of applciation serialization
  - Performance vs. threads – knee of the curve
- **Vectorization**
  - VTune Amplifier hot spots and compiler VEC reports
  - HW PerfMon-based evaluation
  - Performance vs. vectorization on/off
- **Bandwidth**
  - HW PerfMon-based evaluation

Source : References & Intel Xeon-Phi; http://www.intel.com/

More on Thread Affinity

- Bind the worker threads to certain processor core/threads
- Minimizes the thread migration and context switch
- Improves data locality; reduce coherency traffic
- Two components to the problem:
  - How many worker threads to create?
  - How to bind worker threads to core/threads?
- Two ways to specify thread affinity
  - Environment variables OMP_NUM_THREADS, KMP_AFFINITY
  - C/C++ API: `kmp_set_defaults("KMP_AFFINITY=compact")`
    `omp_set_num_threads(244);`
    - Add to your source file#include <omp.h>
    - Compiler with –openmp
    - Use libiomp5.so

Source : References & Intel Xeon-Phi; http://www.intel.com/

The Optimal Thread Number

- Intel MIC maintains 4 hardware contexts per core
  - Round-robin execution policy,
  - Require 2 threads for decent performance
  - Financial algorithms takes all 4 threads to peak
- Intel Xeon processor optionally use HyperThreading
  - Execute-until-stall execution policy
  - Truly compute intensive ones peak with 1 thread per core
  - Finance algorithms likes HyperThreading, 2 threads per core
- Use OpenMP application with NCORE number of cores
  - **Host only:** 2 x ncore (or 1x if HyperThreading disabled)
  - **MIC Native:** 4 x ncore
  - **Offload:** 4 x (ncore-1) OpenMP runtime avoids the core OS runs

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi Coprocessor : Prog. Env & Tips for obtaining Performance (Part-III)

Intel Xeon Phi Coprocessor : Prog. Env &

Use Compiler Optimization Switches

Optimization Done	Linux*
Disable optimization	-O0
Optimize for speed (no code size increase)	-O1
Optimize for speed (default)	-O2
High-level loop optimization	-O3
Create symbols for debugging	-g
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen; -prof-use
Optimize for speed across the entire program	-fast (same as: -ipo –O3 -no-prec-div -static -xHost)
OpenMP 3.0 support	-openmp
Automatic parallelization	-parallel

Source : References & Intel Xeon-Phi; http://www.intel.com/

Prog.API - Multi-Core Systems with Devices

Performance: Intel Xeon-Phi Coprocessor

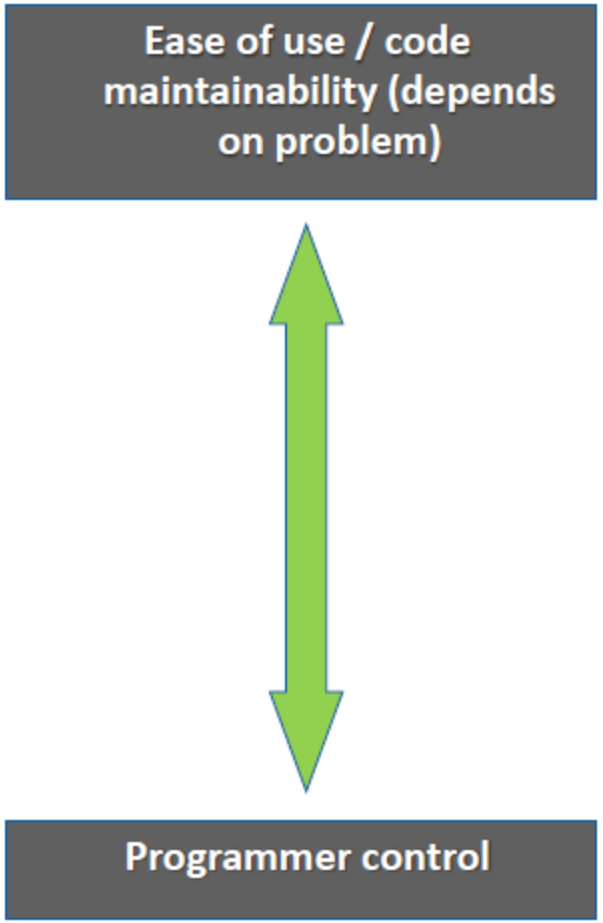
- Vectorization is key for performance
  - Sandybridge, MIC, etc.
  - Compiler hints
  - Code restructuring
- Many-core nodes present scalability challenges
  - Memory contention
  - Memory size limitations

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon-Phi : Prog. Env. Perf Issuses

Options for Vectorization : Use Tools

- Intel® Math Kernel Library
- Array Notation: Intel® Cilk™ Plus
- Automatic vectorization
- Semiautomatic vectorization with annotation: `#pragma vector`, `#pragma ivdep`, and `#pragma simd`
- C/C++ Vector Classes (`F32vec16`, `F64vec8`)
- Vector intrinsics (`_mm_add_ps`, `addps`)



Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Coprocessors – Intel Compiler's Offload Programs

Optimised Offloaded Code

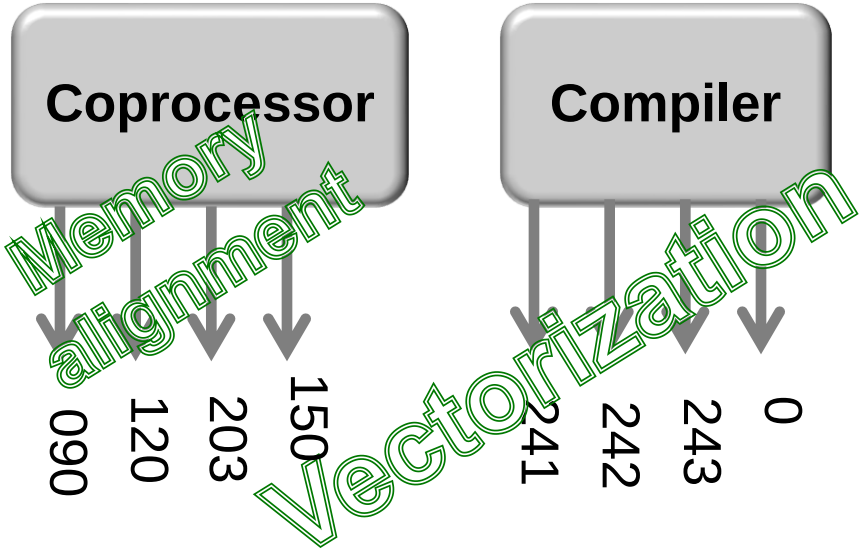
Tuning & Performance :

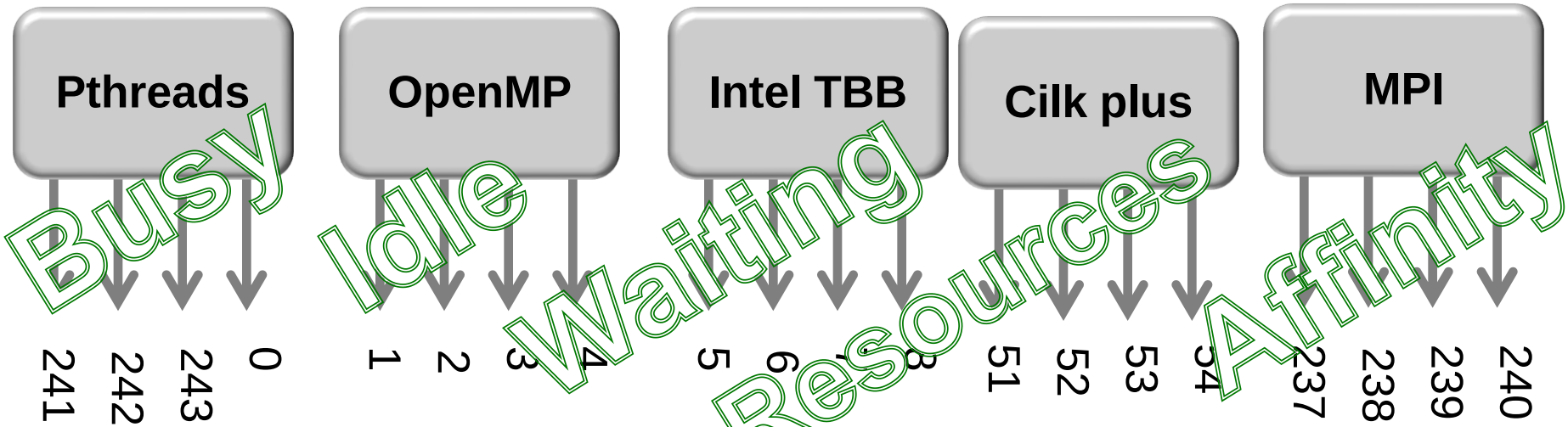
- Using intrinsics with manual data prefetching and register blocking can still considerably increase the performance.

- Try to get a suitable vectorization and write cache and register efficient code, i.e. values stored in registers should be reused as often as possible in order to avoid cache and memory access.

Intel Xeon Phi Prog. : Tools to Measure Overheads

- **Quantification of Overheads : Use Tools on Intel Xeon Phi**
- Prog.on Shared Address Space Platforms (UMA/NUMA)
  - Data Parallel Fortran 2008, Pthreads, OpenMP, Intel TBB Cilk Plus
  - Explicit Message Passing - MPI – Cluster of Message Passing Multi-Core systems

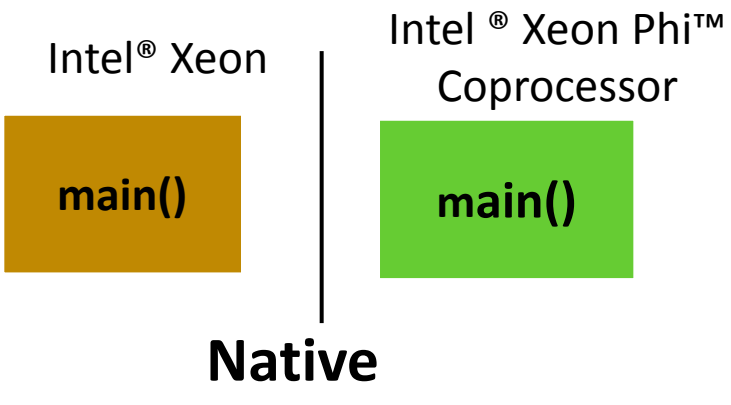


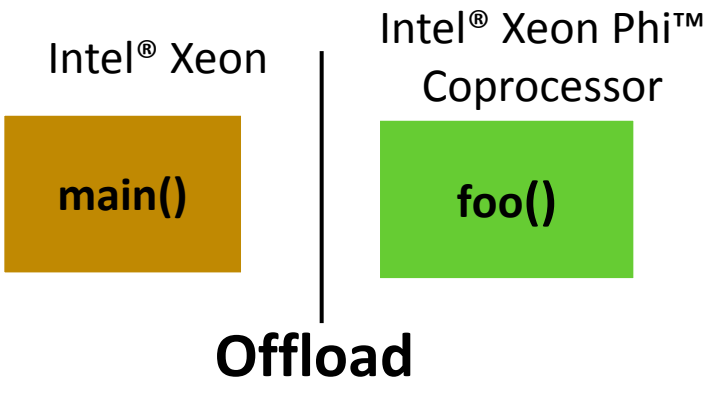


Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon & Xeon Phi : Execution Modes

- **Quantification of Overheads – Explicit / Implicit Data Transfer – Using Offload**





- Card is an SMP machine running Linux
- Separate executables run on both MIC and Xeon
  - e.g. Standalone MPI applications
- No source code modifications most of the time
  - Recompile code for Xeon Phi™ Coprocessor
- Autonomous Compute Node (ACN)

- "main" runs on Xeon
- Parts of code are offloaded to MIC
- Code that can be
  - Multi-threaded, highly parallel
  - Vectorizable
  - Benefit from large memory BW
- Compiler Assisted vs. Automatic
  - #pragma offload (…)

Intel Xeon-Phi : Programming Env.

Pros:

- Compilation with an additional Intel compiler flag (`-mmic`);
- Scalability tests: fast and smooth;
- Quick analysis with Intel tools (VtuneT, Itac Intel Trace Analyzer and Collector;
- Porting time: one day with validation of the numerical result;
- expert developer of FARM, with good knowledge of the Intel Compiler, But with only a basic knowledge of MIC.
- Best scalability with OpenMP and Hybrid.

Source : References & Intel Xeon-Phi; http://www.intel.com/

Xeon Phi : Programming Environment

Porting on MIC : Issues to be addressed

- `MPI Init` routine problem: increasing CPU time for increasing number of processes; Same problem when using two MICs together;
- Detailed analysis of OpenMP threads & thread affinity and Memory available per thread
- Execution time depends strongly from code vectorization, so compiler vectorization for data parallel and task parallel constructs
- code re-structure and memory access pattern are a key point to have a vectorizable satisfactory overall Performances.

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Performance Issues

Factors to work around

- Limited problem size or limited exposure
  - Inherent lack of available parallelism
  - Parallelism not adequately exposed by programmer
- Excessive synchronization
  - Inhibits harvesting thread parallelism
- ISA-specific issues
  - Data structures excessively rely on scatter/gather
  - Use of 64b integer indices and 64 INT←→ FP conversion
- Offload overhead
  - Excessive communication/computation ratio, unhidden communication
- Memory footprint and working set size
  - Limited to 8GB, unless you “overlay,” e.g. with offload

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Performance Issues

Prefetch on Intel Multicore and Many-core

- **Objective:** Move data from memory to L1 or L2 Cache in anticipation of CPU Load/Store
- More import on in-order Intel Xeon Phi Coprocessor
- Less important on out of order Intel Xeon Processor
- Compiler prefetching is on by default for Intel® Xeon Phi™ coprocessors at –O2 and above
- Compiler prefetch is not enabled by default on Intel® Xeon® Processors
  - Use external options `-opt-prefetch[=n] n = 1.. 4`
- Use the compiler reporting options to see detailed diagnostics of prefetching per loop
  - Use `-opt-report-phase hlo -opt-report 3`

Intel Xeon Phi : Performance Issues

Automatic Prefetches

Loop Prefetch

- Compiler generated prefetches target memory access in a future iteration of the loop
- Target regular, predictable array and pointer access

Interactions with Hardware prefetcher

- Intel® Xeon Phi™ Comprocessor has a hardware L2 prefetcher
- If Software prefetches are doing a good job, Hardware prefetching does not kick in
- References not prefetched by compiler may get prefetched by hardware prefetcher

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Performance Issues

Explicit Prefetch

- **Use Intrinsics**
  - `_mm_prefetch((char *) &a[i], hint);`
    See xmmintrin.h for possible hints (for L1, L2, non-temporal, …)
  - But you have to specify the prefetch distance
  - Also gather/scatter prefetch intrinsics, see zmmintrin.h and compiler user guide, e.g. _mm512_prefetch_i32gather_ps
- **Use a pragma / directive (easier):**
  - **#pragma prefetch a [:hint[:distance]]**
  - You specify what to prefetch, but can choose to let compiler figure out how far ahead to do it.
- **Use Compiler switches:**
  - `-opt-prefetch-distance=n1[,n2]`
  - specify the prefetch distance (how many iterations ahead, use n1 and prefetches inside loops. n1 indicates distance from memory to L2.

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Performance Issues

Streaming Store

- Avoid read for ownership for certain memory write operation
- Bypass prefetch related to the memory read
- Use `#pragma vector nontemporal(v1,…)` to drop a hint to compiler
- Without Streaming Stores 448 B[illegible]

  - With Streaming Stores, 320 Bytes read/write per iteration
  - Relief Bandwidth pressure; improve cache utilization
  - –vec-report6 displays the compiler action

```
for (int chunkBase = 0; chunkBase < OptPerThread; chunkBase += CHUNKSIZE)
{
#pragma simd vectorlength(CHUNKSIZE)
#pragma simd
#pragma vector aligned
#pragma vector nontemporal (CallResult, PutResult)
      for(int opt = chunkBase; opt < (chunkBase+CHUNKSIZE); opt++)
      {
         float CNDD1;
         float CNDD2;
         float CallVal =0.0f, PutVal  = 0.0f;
         float T = OptionYears[opt];
         float X = OptionStrike[opt];
         float S = StockPrice[opt];
                        ……

         CallVal  = S * CNDD1 - XexpRT * CNDD2;
         PutVal  = CallVal  +  XexpRT - S;
         CallResult[opt] = CallVal ;
         PutResult[opt] = PutVal ;
      }
}
```

```
bs_test_sp.c(215): (col. 4) remark: vectorization support: streaming store was generated for CallResult.
bs_test_sp.c(216): (col. 4) remark: vectorization support: streaming store was generated for PutResult.
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Performance Issues

Data Blocking

- Partition data to small blocks that fits in L2 Cache
  - Exploit data reuse in the application.
  - Ensure the data remains in the cache across multiple uses
  - Using the data in cache remove the need to go to memory
  - Bandwidth limited program may execute at FLOPS limit
- Simple case of 1D
  - Data size DATA_N is used WORK_N times from 100s of threads
  - Each handles a piece of work and have to traverse all data

Without Blocking

- 100s of thread pound on different area of DATA_N
- Memory interconnet limit the performance

```
#pragma omp parallel for
for(int wrk = 0; wrk < WORK_N; wrk++)
{
     initialize_the_work(wrk);
     for(int ind = 0; ind < DATA_N; ind++)
     {
        dataptr  datavalue = read_data(dataind);
        result = compute(datavalue);
        aggregate = combine(aggregate, result);
     }
     postprocess_work(aggregate);
}
```

With Blocking

- Cacheable BSIZE of data is processed by all 100s threads a time
- Each data is read once kept reusing until all threads are done with it

```
for(int BBase = 0; BBase < DATA_N; BBase += BSIZE)
{
#pragma omp parallel for
    for(int wrk = 0; wrk < WORK_N; wrk++)
    {
         initialize_the_work(wrk);
         for(int ind = BBase; ind < BBase+BSIZE; ind++)
         {
            dataptr  datavalue = read_data(ind);
            result = compute(datavalue);
            aggregate[wrk] = combine(aggregate[wrk], result);
          }
          postprocess_work(aggregate[wrk]);
    }
}
```

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Xeon Phi : Performance Issues

Memory Alignment

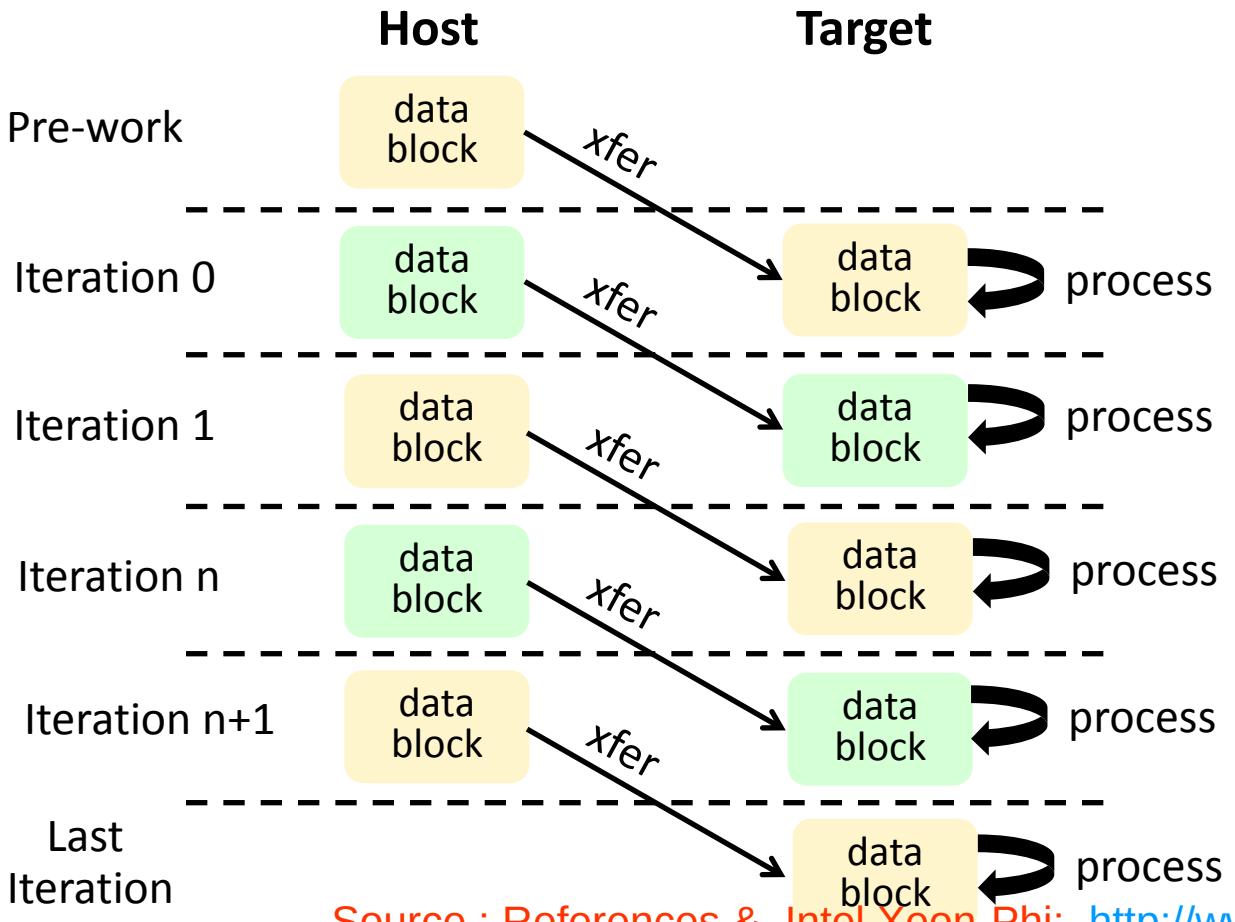
- Allocated memory on heap
  - `_mm_malloc(int size, int aligned)`
  - `scalable_aligned_malloc(int size, int aligned)`
- Declarations memory:
  - `__attribute__((aligned(n))) float v1[];`
  - `__declspec(align(n)) float v2[];`
- Use this to notify compiler
  - `__assume_aligned(array, n);`
- Natural boundary
  - Unaligned access can fault the processor
- Cacheline Boundary
  - Frequently accessed data should be in 64
- 4K boundary
  - Sequentially accessed large data should be in 4K boundary

Instruction	Length	Alignment
SSE	128 Bits	16 Bytes
AVX	256 Bits	32 Bytes
IMCI	512 Bits	64 Bytes

Source : References & Intel Xeon-Phi; http://www.intel.com/

Double Buffering Example

- Transfer and work on a dataset in small pieces
- While part is being transferred, work on another part!

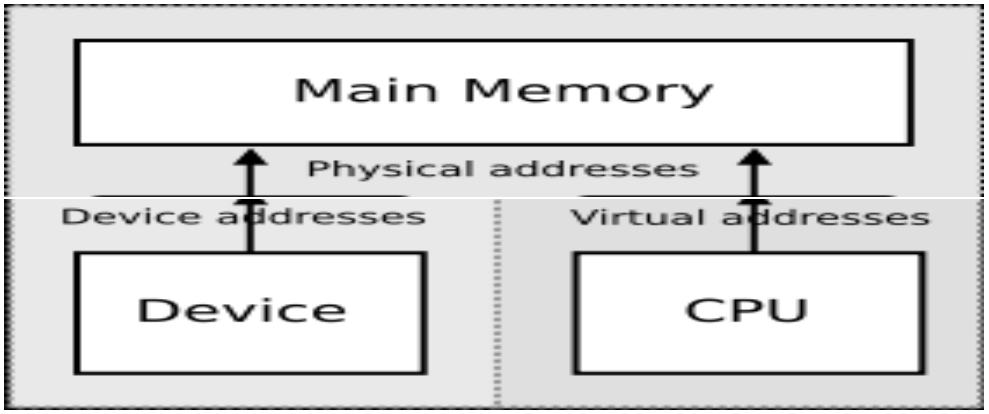


Source : References & Intel Xeon-Phi; http://www.intel.com/

Memory Mapping

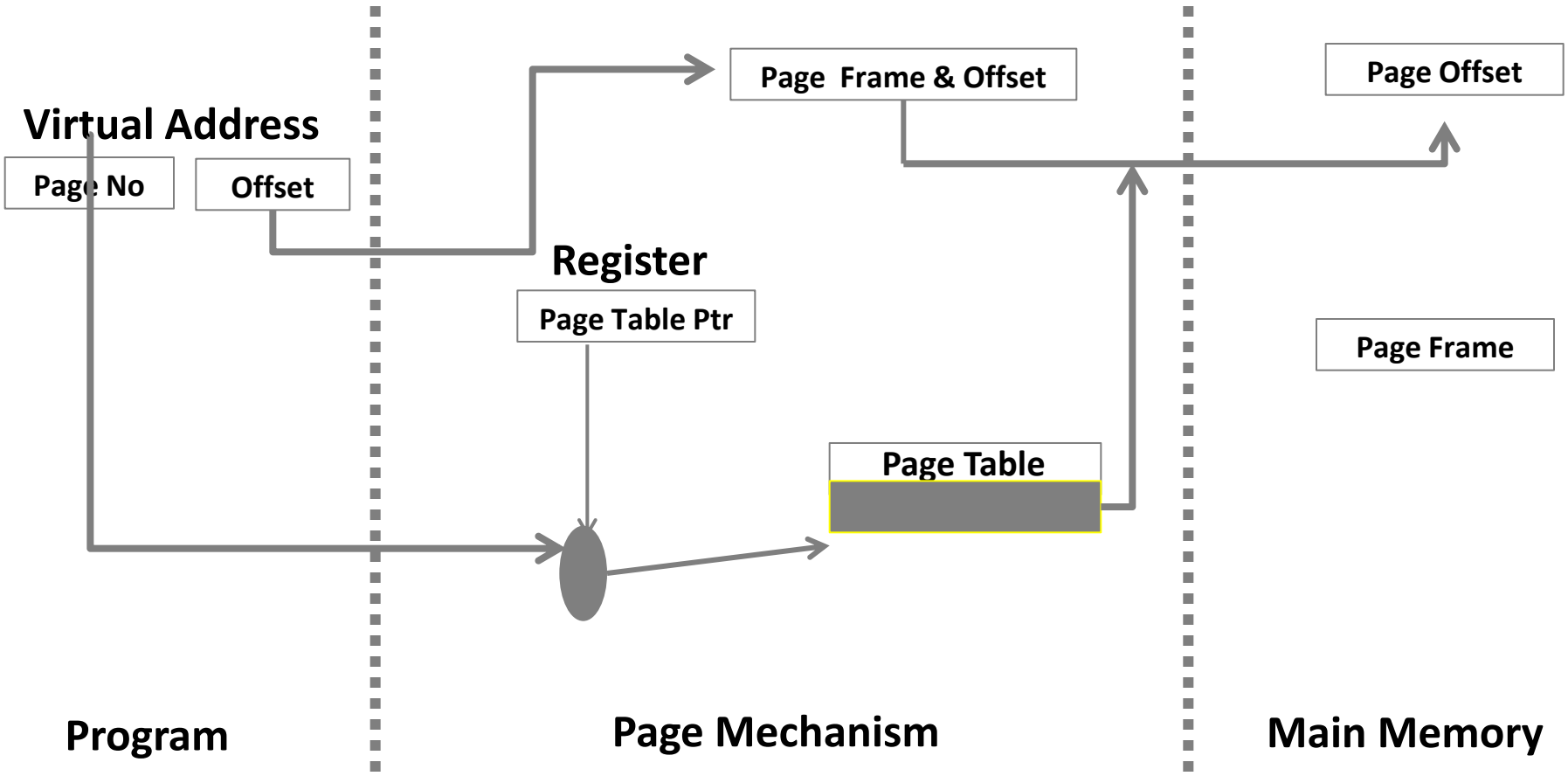
Implementation: Matrix into Matrix Multiplication using mmap (Assume that Matrix Size A = 1,00,000 Real float and Matrix Size B = 1,00,000 Real float)

- Translation of address issued by some device (e.g., CPU or I/O device) to address sent out on memory bus (physical address)
- Mapping is performed by memory management units



Computing – Enabling Huge Memory – Implementation using Memory Mapping (mmap)

Address Mapping Function (Review)



Memory – Huge Pages and Pre-faulting

- IA processors support multiple page sizes; commonly 4K and 2MB
- *Some* applications will benefit from using huge pages
  - Applications with sequential access patterns will improve due to larger TLB "reach"
- TLB miss vs. Cache miss
  - TLB miss means walking the 4 level page table hierarchy
    - Each page walk could result in additional cache misses
  - TLB is a scarce resource and you need to "manage" them well
- On Intel® Xeon Phi™ Coprocessor
  - 64 entries for 4K, 8 entries for 2MB
  - Additionally, 64 entries for second level DTLB.
    - Page cache for 4K, L2 TLB for 2MB pages
- Linux supports huge pages – CONFIG_HUGETLBFS
  - 2.6.38 also has support for Transparent Huge Pages (THP)
- Pre-faulting via MAP_POPULATE flag to mmap()

Intel Xeon Phi : The Intel Composer XE 2013

- The Intel Composer XE – Development tool and SDK suite available for developing Intel Xeon Phi
  - It includes C/C++ Fortran Complier
  - It includes runtime libraries like OpenMP, thread etc. Debuging tool and math kernel library (MKL)
  - Supports various parallel programming models fro Intel Xeon Phi such as Intel Cilk Plus, Intel Threading Building blocks (TBB), OpenMP and Pthread
  - It includes Intel MKL

Source : References & Intel Xeon-Phi; http://www.intel.com/

Intel Trace Analyzer and Collector (ITAC)

❖ **Intel MPI, Intel Trace Analyzer and Collector(ITAC) on MIC**

- Intel Trace Collector gathers information from running programs into a trace file, and the Intel Trace Analyzer allows the collected data to be viewed and analyzed after a run.
- The Intel Trace Analyzer and Collector support processors and coprocessors.
- The Trace Collector can integrate information from multiple sources including an instrumented Intel MPI Library and PAPI.
- Trace file from an application running on the **host system** and **coprocessor** simultaneously can be generated
- Generate trace file only **on Coprocessor** system

Source : References & Intel Xeon-Phi; http://www.intel.com/

An Overview of Prog. Env on Intel Xeon-Phi

Summary

- An Overview of Intel Xeon-Phi Coprocessor Architecture & Software Environment is discussed
- Programming paradigms on Intel Xeon-Phi Coprocessor are discussed
- Tips for Tuning & Performance Issues on Intel Xeon-Phi Coprocessor are discussed

Thank You

Any questions ?