

Quick Start Guide Execution Modes for Intel® Xeon Phi™ Coprocessors

**Arturo Argueta, Roberto Camacho Barranco,
Esthela Gallardo and Pat Teller
UTEP Technology Insertion Project
Leonardo Fialho and Jim Browne
UT-Austin Technology Insertion Project**



Motivation

An application can be executed on a system that incorporates both CPUs and Intel® Xeon Phi™ coprocessors in four ways.

- **Native Mode – Intel Xeon Phi coprocessors only**
- **CPU mode – CPUs only**
- **Offload mode – Both: functional partitioning**
- **Symmetric mode – Both: data partitioning**

Performance can differ drastically by mode.

Many factors to consider!

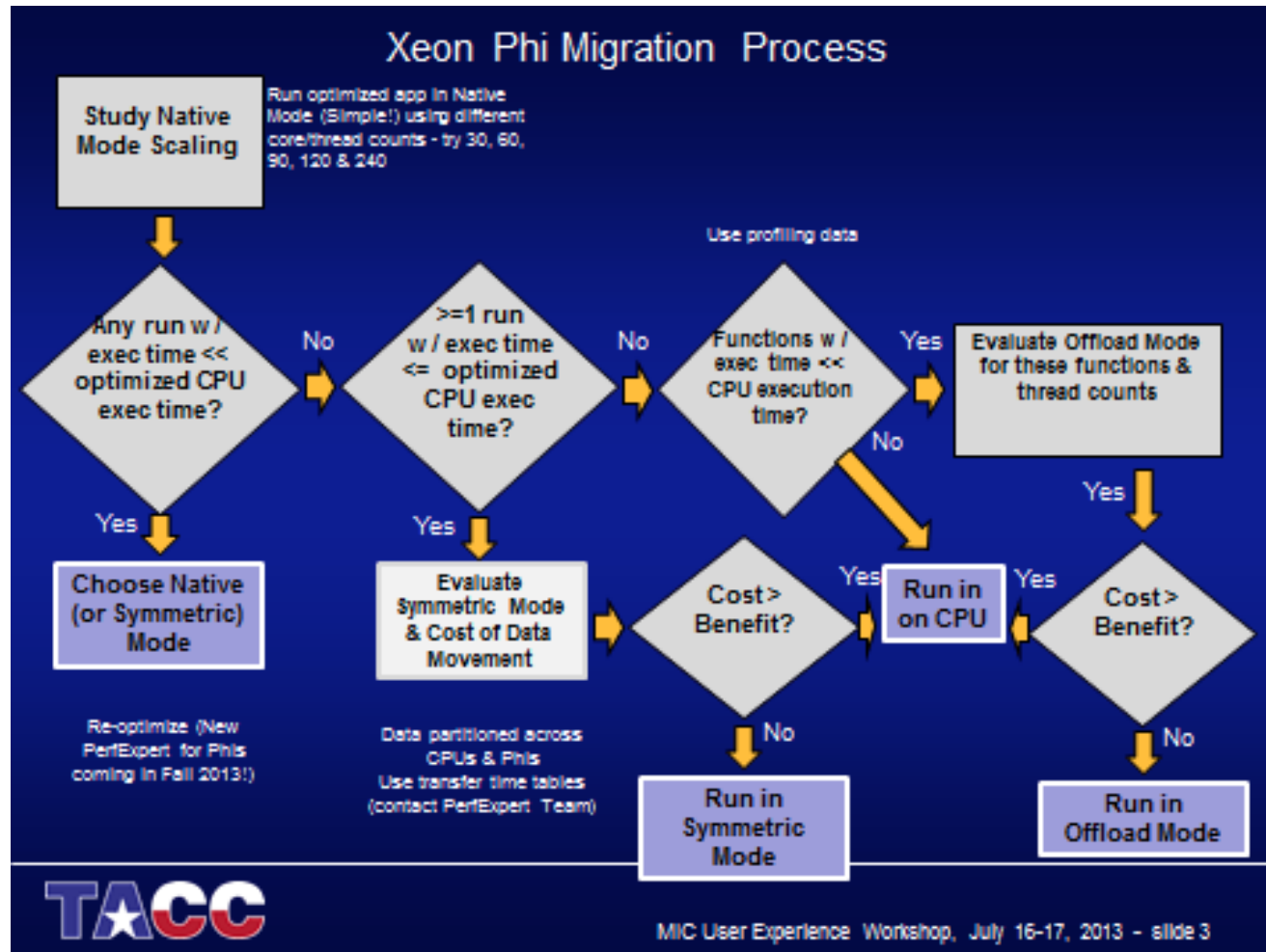
How to choose execution mode?

Strategy

- 1. Organize as systematic workflow.**
- 2. Specify entire workflow of four steps (four experiments) in two pages**
- 3. Make each step describable in two pages**
- 4. Provide examples for each step**



Workflow



Optimize Code for CPU Execution

Why?

It has been established that code that is well optimized for Sandy Bridge CPUs is often well optimized for Intel® Xeon Phi™ coprocessors and vice versa.

How?

- Use [PerfExpert](#) to analyze your code for: (i) vectorization and (ii) the *optimal scaling*, i.e., the optimal number of tasks/threads to execute per node.
- Profile the execution of the optimized code to determine the execution time of functions/procedures; using a significant fraction of the total execution time at the optimal scale.

Analysis for Native Mode

Follow instructions in Quick Start Guide for Native Mode Execution

- 1. Add OpenMP pragmas for each loop nest in each important function determined by profiling .**
- 2. Compile OpenMP Code for Intel® Xeon Phi™ coprocessors**
- 3. Run on run on Xeon Phi Coprocessors for 60, 120 and 240 threads**
- 4. If the Phi-only runs give significantly better performance than the CPU-only runs then either Native or Symmetric Mode will be the best choice**
- 5. Re-optimize (PerfExpert) your code on the Intel Xeon Phi coprocessor and collect run times of significant functions.**

Analysis For Symmetric Mode

- If best Intel® Xeon Phi™ coprocessor-only and Sandy Bridge cpu-only executions are about the same speed, consider Symmetric Mode.
 - Prepare a symmetric mode code by following the *Quick Start Guide for Symmetric Mode Execution*
 - Symmetric Mode is beneficial if the speedup in execution time by using both CPUs and Intel Xeon Phi coprocessors exceeds the cost of moving data between CPU and Intel Xeon Phi coprocessor memories.
 - See the *Quick Start Guide for Symmetric Mode Execution* to estimate the cost of data movement

Analysis of OffLoad Mode

- If there are no important functions that are much faster on the Intel® Xeon Phi™ coprocessor, execute on the CPUs.
- If there are important functions which execute much faster on the Intel Xeon Phi coprocessors then evaluate Offload Mode following the example given in the *Quick Start Guide for Offload Mode Execution*
 - Assign these functions to the Intel Xeon Phi coprocessor and determine the cost of data movement by following the example in *Offload Mode: Estimation of Execution Time and Cost of Data Movement*.
- If the time saved in execution time exceeds the cost of data movement, then try Offload Mode.



Offload Mode: Estimation of Execution Time and Cost of Data Movement

- To obtain the execution time of offloaded code and the amount of data (number of bytes) transferred and the execution time cost of data movement, use the following command: `export OFFLOAD_REPORT = 2`
- For more information regarding OFFLOAD_REPORT see: <http://software.intel.com/en-us/node/463216>. For more information on how to set the OFFLOAD_REPORT variable see: <http://software.intel.com/en-us/node/463218>
- For the example code, the median value to transfer a byte was 2.12762E-09 sec.

Quick Start Guides

- The QSGs for Native, Symmetric and OffLoad modes each has a complete example including the compiler options and pragmas, OpenMP pragmas, etc.
- The QSGs can be downloaded from the TACC web site:
<https://www.tacc.utexas.edu/research-development/tacc-projects/mic-development>

Future Work

- **Revise and edit to make simpler and shorter**
- **Respond to suggestions for improvement**
- **Update data movement cost estimates as MPI improves.**
- **Update examples as compilers change**
- **Develop a script or use a workflow tool to automate the full four experiment process and returning a recommendation.**
- **Anyone interested in helping with this project?**



Combining Compiler Analyses and Runtime Measurement to Enhance Vectorization

Ashay Rane – UT TACC

Rakesh Krishnaiyer, Chris J Newburn and
Zakhar Matveev - Intel

Jim Browne, and Leo Fialho – UT TACC

Motivation

- Chip and node architectures and modern languages are both very complex. Best (most efficient) executables are thus complex to write.
- Compiler static analyses alone cannot generate optimal executable
- Optimization based on runtime measurements is much more effective when guided by static analysis
- Vectorization is becoming a critical factor for both execution speed and power consumption.



Approach

- Integrate compiler static analyses with runtime measurement and analyses to generate “best possible” optimization process.
- Automate this process as much as possible.

Workflow

1. Profile application for hotspots using production inputs.
2. Parse compiler vectorization reports to find why hot spot loops are not fully vectorized and what information the compiler needs.
3. Instrument hot-loops that are not fully vectorized for information compiler could not determine
4. Gather measurements, analyze results and generate recommendations for enhancing vectorization.
5. Verify validity of the recommended changes.
6. Implement changes, measure performance gains.

Automated step

Manual step



Tool (MACVEC) workflow



Causes for Poor Vectorization

Compiler could not determine hence compiler assumed:

- **Inter-iteration (vector dependence) dependence.**
- **Varying trip count (non-countable loop).**
- **Temporal array references.**
- **Mis-aligned loads and stores.**
- **Access strides longer than one element for one or more structured variables**
- **Unknown branch probabilities**



Measurement Overhead

Measurements

Overhead

Loop trip counts

1.08x

Array access strides

1.05x

Alignment of arrays

1.12x

Overlapping pointers

1.07x

Branch path outcomes

1.07x

Strides

2x-4x

Potential Benefit

Rodinia Benchmarks for Accelerator Performance

Application	Time
heartwall	07.43%
euler	12.42%
kmeans	19.54%
backprop	32.52%
leukocyte	35.01%
lavaMD	37.42%
srad_v1	48.45%
pre_euler_double	71.60%
pre_euler	75.94%
euler_double	78.99%
streamcluster	85.58%

Fraction of Execution time spent in incompletely vectorized loops.



Examples – Unvectorized Loops

Example: Rodinia LavaMD.

- Hot function `kernel_cpu(box* b, fp* qv, ...)` defined in `kernel_cpu.c`.
- Compiler does not know caller arguments when compiling `kernel_cpu.c`.
- Assumes pointers `b` and `qv` may overlap in memory.
- Concludes existence of vector dependence.



Example – Poorly Vectorized Loop

Example: NAS CG.

- Unknown loop trip count: for ($k = \text{rowstr}[j]$; $k < \text{rowstr}[j+1]$; $k++$) { }
- Double indirection in loop body: $\text{suml} += a[k] * p[\text{colidx}[k]]$;
- Compiler generates gather/scatter instructions for each iteration.

NAS CG Loops

- `#pragma omp for`
- `for (j = 0; j < lastrow - firstrow + 1; j++) {`
- `suml = 0.0;`
- `for (k = rowstr[j]; k < rowstr[j+1]; k++) {`
- `suml = suml + a[k]*p[colidx[k]];`
- `}`
- `q[j] = suml;`
- `}`

NAS CG Loop

- `#pragma omp for`
- `for (j = 0; j < lastrow - firstrow + 1; j++) {`
- `for (k = rowstr[j]; k < rowstr[j+1]; k++) {`
- `colidx[k] = colidx[k] - firstcol;`
- `}`
- `}`



Examples of ineffective vectorization

- Example: NBody.
- Operates on dynamically allocated (malloc()ed) arrays
- Memory allocator may allocate objects in any way that it desires.
- Compiler cannot guarantee alignment of objects to cache-line boundary.



Implementation

- Hotspot measurements and performance benchmarking on
- Binaries compiled using Intel 14.0 and GCC compilers.
- Source code instrumentation using the Rose source-to-source compiler.
- All measurements on Intel Xeon (Sandy Bridge) and
- Intel Xeon Phi (Knights Corner).
- Recommendations are based on source code analysis,
- Hence recommendations are independent of target compiler.

Dynamic profiling measurements

- Loop trip counts.
- Array access strides.
- Alignment of arrays.
- Iteration Dependence - Overlapping pointers.
- Reuse distance per data structure.
- Branch path outcomes.



Rule Based Recommendations

Stride

Precondition: All arrays aligned to cache-line boundaries, Loop is vectorizable.

Recommendation: `#pragma vector aligned` .

Non-temporal stores

Precondition: Low reuse for specific array, Loop is vectorizable.

Recommendation: `#pragma vector nontemporal` .

Streaming stores

Precondition: Arrays are written but never read back., Arrays are accessed with unit stride, no mask register., Low reuse for specific array.

Recommendation: `-opt-streaming-stores=always` .

Rule Based Recommendations

Pointer-overlap checks

Precondition: Span of memory accessed using pointers does not overlap with other pointer accesses.

Recommendation: restrict keyword.

Branch path analysis

Precondition: Branch evaluates to always true or always false.

Recommendation: builtin_expect() .

Stride

Precondition: Code to be compiled for Intel Xeon Phi.,
Fixed-length strides that are more than 4 cache lines apart.

Recommendation: #pragma prefetch array , -opt-gather-scatter-unroll .

Validation

Validation Applications	Xeon	Xeon Phi
Nbody	0.93x	1.45x
Stream Copy	1.06x	1.00x
Stream Scale	1.41x	1.32x
Stream Add	1.30x	1.29x
Stream Triad	1.29x	1.30x

Case Studies

Benchmark Codes	Xeon	Xeon Phi
NAS CG	1.06x	2.18x
LavaMD - Rodinia	2.19x	8.99x
SRAD - Rodinia	0.99x	1.09x

LavaMD – 8.99x with V14 of Intel compiler, 5x with V15 compiler.

Case Studies

Benchmark Codes	Xeon	Xeon Phi
NAS CG	1.06x	2.18x
LavaMD - Rodinia	2.19x	8.99x
SRAD - Rodinia	0.99x	1.09x

LavaMD – 8.99x with V14 of Intel compiler, 5x with V15 compiler.

Case Studies

Applications	Xeon	Xeon Phi
LBM	1.06x	1.20x
Lulesh	1.03x	1.00x
MILC	1.10x	1.60x

Performance improvement for MILC is for matrix computations only.

Safety of Recommendations

- Are recommendations independent of standard compiler optimizations?
- Will recommendations be applicable across multiple program inputs?
 - Seven of the nine recommendations are guaranteed to be safe.
 - $O(1)$ runtime checks guarantee safety for remaining recommendations.
 - Source-code level measurements ensure that measurements are valid across compilers.



Future Work

- **Package MACVEC as a module for Stampede**
- **Make MACVEC open source downloadable**
- **Complete automation of vectorization optimizations by integration into PerfExpert**
- **Provide version of MACVEC which reports compiler messages, tells user what to check and provides recommendation.**
- **Add additional analyses, set associativity conflicts, etc.**
- **Provide Fortran version**



Collaborators

- We are soliciting collaborators who would like to apply MACVEC to their codes on Stampede
- Contact – browne@cs.utexas.edu

Acknowledgement

This project was funded by Intel and by the NSF through the Stampede grant.

LavaMD Loop

- for (k=0; k<(1+box[l].nn); k++)
- {
- // neighbor box - get pointer to the right box
- if(k==0){
- pointer = 1;
- // set first box to be processed to home box
- }
- else{
- pointer = box[l].nei[k-1].number; ////
- //remaining boxes are neighbor boxes
- }
- // neighbor box - box parameters
- first_j = box[pointer].offset;
- // neighbor box - distance, force, charge and type parameters
- rB = &rv[first_j];
- qB = &qv[first_j];
- // Do for the # of particles in home box



LavaMD Loop

```
• for (i=0; i<NUMBER_PAR_PER_BOX; i=i+1){  
•  
•  
• // do for the # of particles in current (home or neighbor) box  
• for (j=0; j<NUMBER_PAR_PER_BOX; j=j+1){  
•  
• // // coefficients  
• r2 = rA[i].v + rB[j].v - DOT(rA[i],rB[j]);  
• u2 = a2*r2;  
• vij= exp(-u2);  
• fs = 2.*vij;  
• d.x = rA[i].x - rB[j].x;  
• d.y = rA[i].y - rB[j].y;  
• d.z = rA[i].z - rB[j].z;  
• fxij=fs*d.x;  
• fyij=fs*d.y;  
• fzij=fs*d.z;  
•  
•
```



LavaMD Loop

- // forces
- - $fA[i].v += qB[j]*vij;$
 - $fA[i].x += qB[j]*fxij;$
 - $fA[i].y += qB[j]*fyij;$
 - $fA[i].z += qB[j]*fzij;$
-
- - } // for j
-
- - } // for i
-
- - } // for k
-