

# 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.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

## 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



# Optimizing OpenCL Applications on Intel® Xeon Phi™ Coprocessor

IWOCL-2013 Tutorial

Ayal Zaks, Intel

Arik Narkis, Intel (presenter)

[ayal.zaks@intel.com](mailto:ayal.zaks@intel.com)  
[arik.narkis@intel.com](mailto:arik.narkis@intel.com)



# Introduction

- Why Intel® Xeon Phi™ coprocessor?
- Why OpenCL\*?
- What's the challenge?
  
- Production level OpenCL\* on Intel Xeon Phi coprocessor has just been released (May 8<sup>th</sup>, 2013)
  - Subsequent releases are expected to improve features-set and performance

# High Level Outline

Intel® Xeon Phi™ Coprocessor overview

Mapping the OpenCL\* constructs to Xeon Phi

Performance tuning and optimizations

Tools and resources

Summary and Q&As

# Intel® Xeon Phi™ Coprocessor Overview

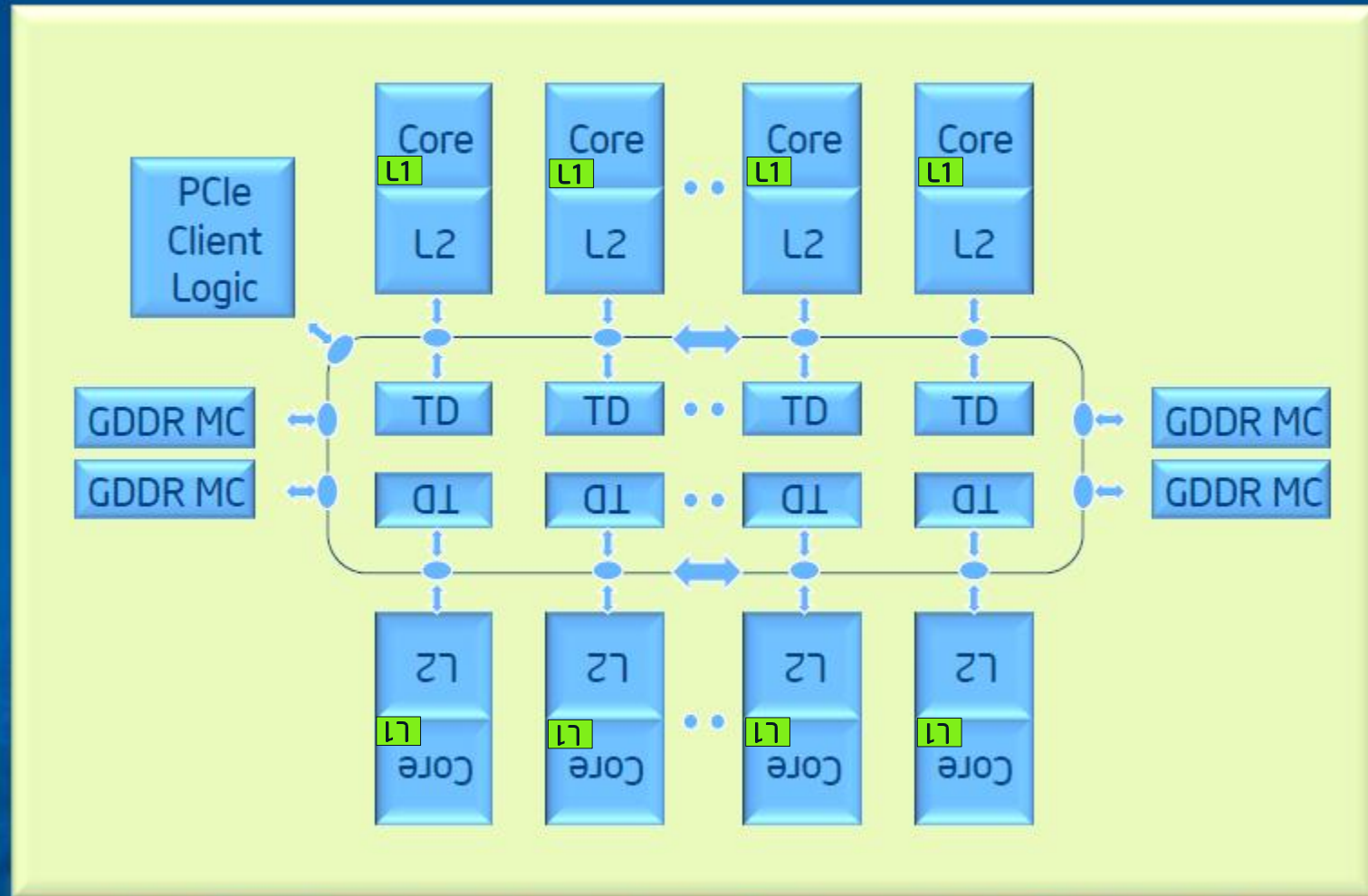


Xeon Phi Developer site: <http://software.intel.com/mic-developer>

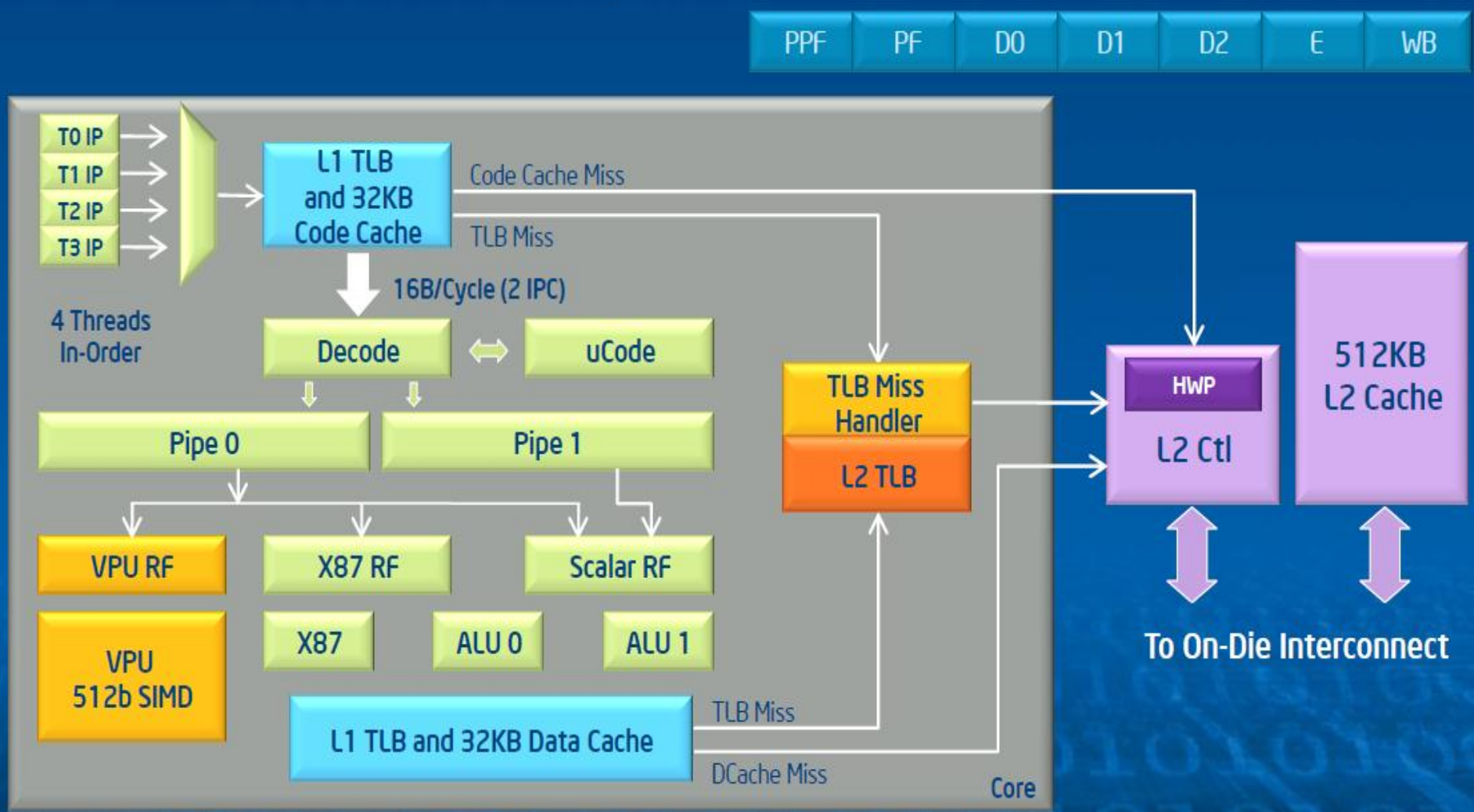
# Intel® Many Integrated Core (Intel MIC) Architecture

- Targeted at highly parallel HPC workloads
  - Physics, Chemistry, Biology, Financial Services
- General Purpose Programming Environment
  - Runs Linux\* (full service, open source OS)
  - Runs applications written in Fortran, C, C++, OpenMP, OpenCL\* ...
  - Runs the x86 ISA + new SIMD extension
  - Supports X86 coherent memory model, IEEE 754
  - x86 collateral (libraries, compilers, Intel® VTune™, debuggers, etc)

# Intel® Xeon Phi™ Coprocessor Micro Architecture



# Intel® Xeon Phi™ Coprocessor – The Core



X86 specific logic < 2% of core + L2 area



**Intel® Xeon Phi™ Coprocessor is an  
x86 based, many-core co-processor  
With wide SIMD vector instruction**

**Moving to OpenCL\* Mapping  
to Xeon Phi ...**

# The Intel® SDK for OpenCL Applications Online Resource

The SDK section of the Intel® Developers Zone is a one-stop shop for resources, support and information for OpenCL\* developers

Dashboard » Tools, SDKs, LIBS » Intel® SDK for OpenCL® Applications XE 2013 Beta

Intel® SDK for OpenCL® Applications XE 2013 Beta

With OpenCL\* 1.2 features on Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors for Linux® OS.

Join an OpenCL training near you

DOWNLOAD

By downloading this package you accept the End User License Agreement

Support

Support Forum  
Frequently Asked Questions  
OpenCL\* at Khronos.org

Product Documents

Release Notes  
Optimization Guide [html] [pdf]  
User Guide [html] [pdf]  
OpenCL\* Code Samples  
OpenCL\* Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor

Tools and Applications

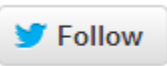
Computing Language Utility (CLU)  
OpenCL\* with Intel® VTune™ Amplifier XE Usage Guide

Intel OpenCL products matrix: [Compare and Download Products](#)

Target Processors	Target Operating System	OpenCL spec version	Target SDK

- ✓ Free Downloads
- ✓ Code Samples
- ✓ Tech Articles
- ✓ Case Studies
- ✓ Forums and Support
- ✓ Beta Programs

[intel.com/software/opencv-xe](http://intel.com/software/opencv-xe)  
[intel.com/software/opencv](http://intel.com/software/opencv)  
[@intelopencv](https://twitter.com/intelopencv)



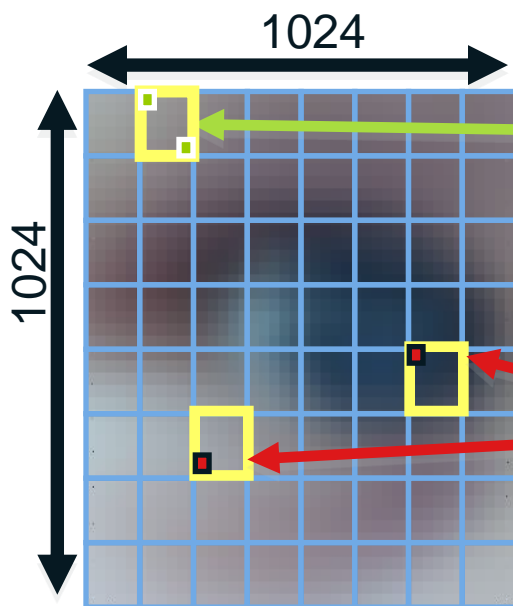
# The NDRange - Example

The NDRange defines a compute task on one of the queues

Global Dimensions: 1024 x 1024 (whole problem space, 1 M work-items)

Local Dimensions: 128 x 128 (work group ... executes together, 16 K WIs)

Workgroups space: 8 x 8 (total 64 WGs)



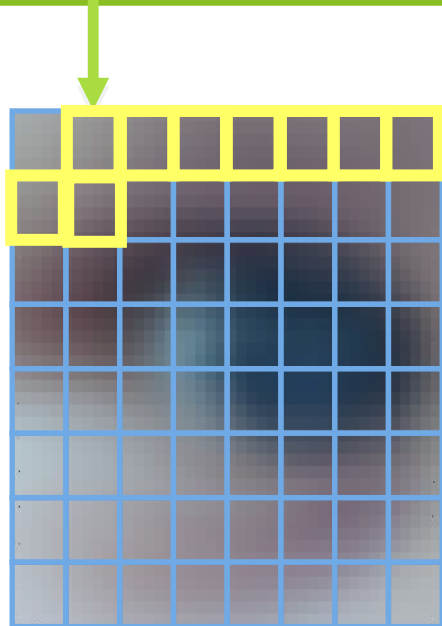
**Synchronization between work-items possible only within a workgroup via local memory / barrier instruction**

**Cannot synchronize outside of a workgroup**

# The NDRange on Intel® Xeon Phi™ Coprocessor

- The workgroup is the smallest task
- Whole workgroups are parallelized on HW threads

**Workgroup[0][1]**



## Xeon Phi

Core 0



Core N-4



Core N-3



Core N-2



Core N-1



# The Work-group

The OpenCL\* compiler creates an optimized routine that executes a WG

```
__Kernel ABC(...)  
for (int i = 0; i < get_local_size(2); i++)  
    for (int j = 0; j < get_local_size(1); j++)  
        for (int k = 0; k < get_local_size(0); k++)  
            Kernel_Body;
```

Dimension zero of the NDRange is the most inner loop

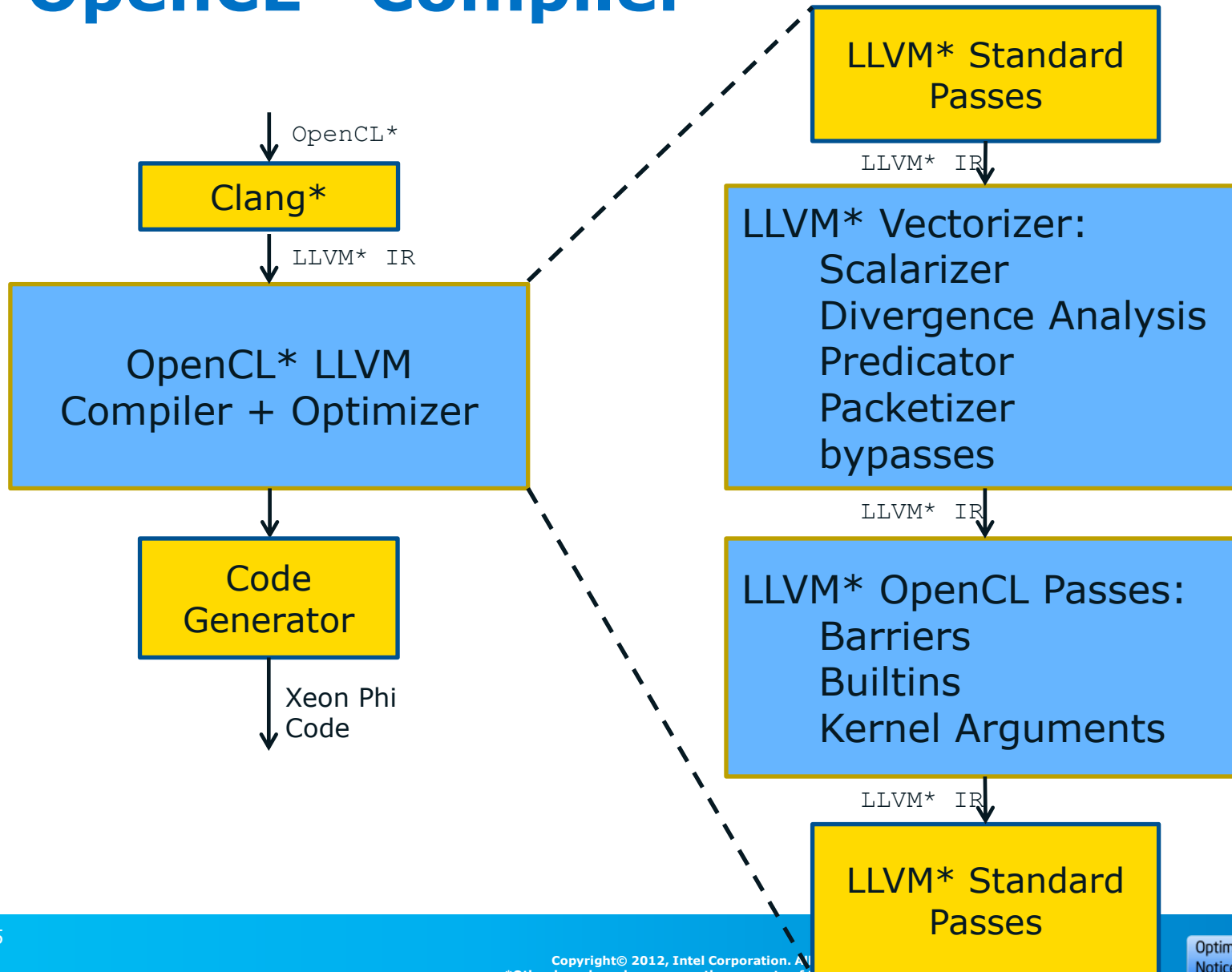
# The Workgroup (cont.)

We haven't utilized the HW vector unit yet

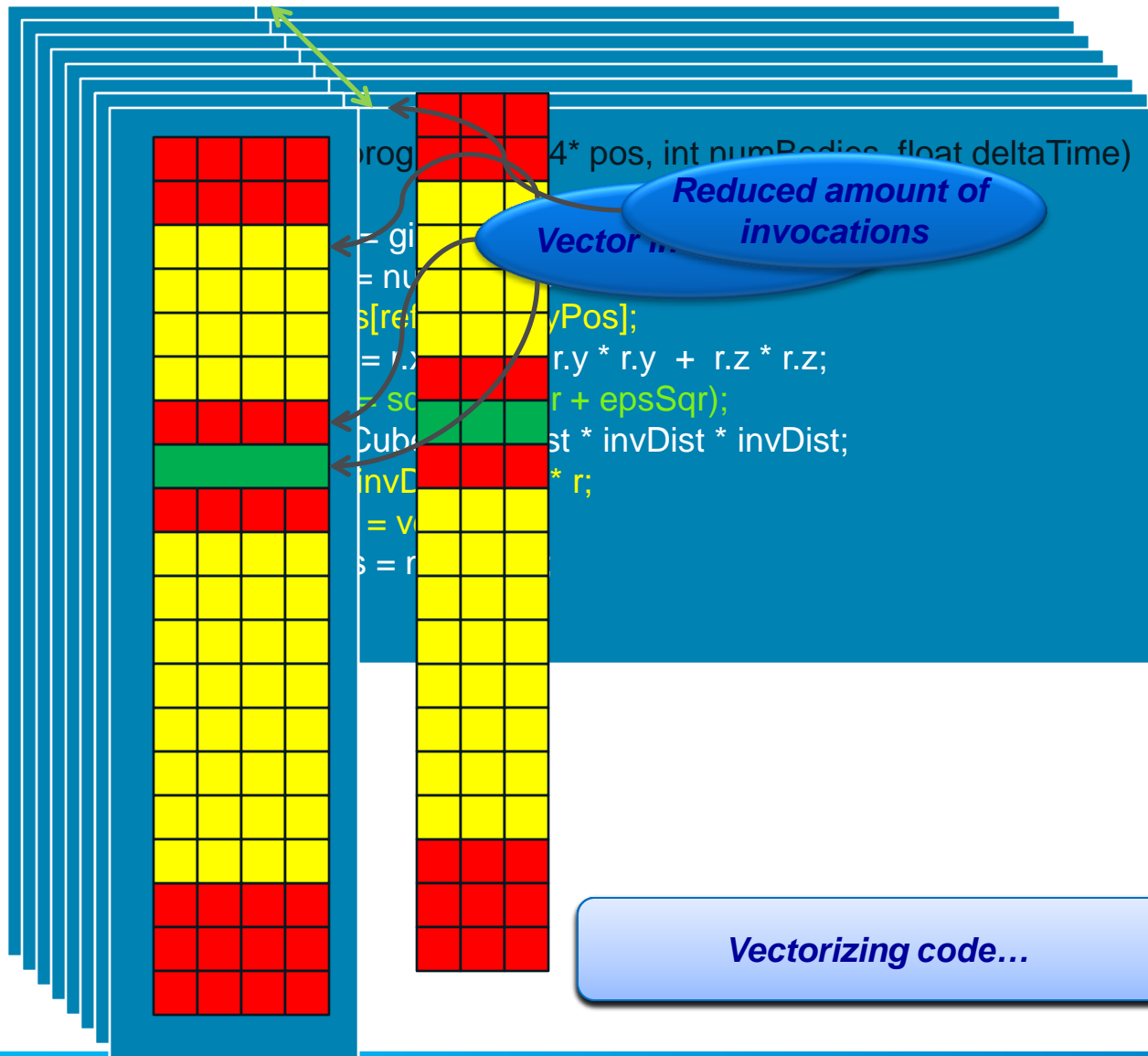
```
__Kernel ABC(...)  
For (int i = 0; i < get_local_size(2); i++)  
    For (int j = 0; j < get_local_size(1); j++)  
        For (int k = 0; k < get_local_size(0); k += VEC_SIZE)  
            Vector_Kernel_Body;
```

- Implicit vectorization over dimension zero of the NDRange
- No reason to vectorize manually

# Intel® Xeon Phi™ Coprocessor OpenCL\* Compiler



# Vectorization example





**Parallelize WGs across the HW threads  
Vectorize WIs across the SIMD unit**

**Moving to Optimizations . . .**

# Host-device efficiency

The PCIe\* is the slowest data channel in the platform

- Transfer reduction
- Implicit transfer elimination
- Overlap compute with transfer

# Host-device efficiency

## Transfer reduction – the obvious

- Use the minimal data type needed for the problem
  - float/double
  - int/long
- Transfer only the data elements needed
  - Array of Structures may include unused fields
- Avoid padding:

```
struct{  
    float a;  
    double b;  
    float c;  
}abc;
```

24 Bytes

```
struct{  
    double b;  
    float a;  
    float c;  
}abc;
```

16 Bytes

# Host-device efficiency

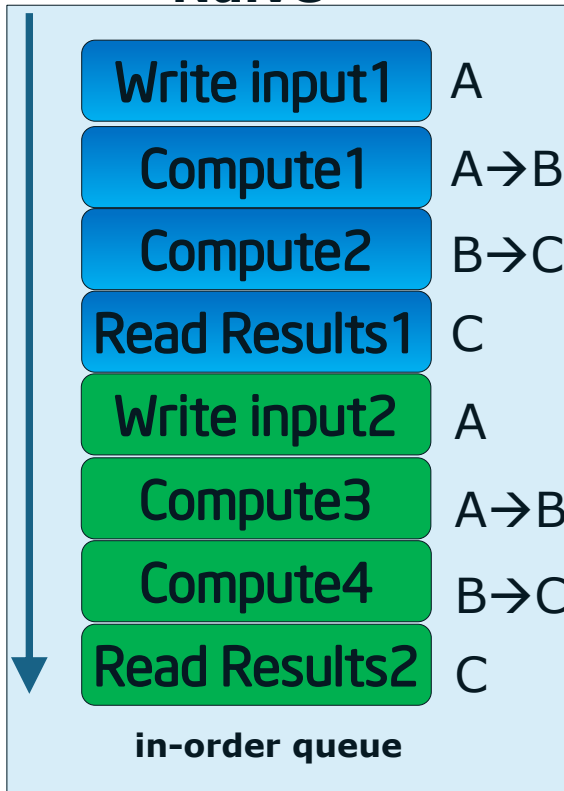
## Avoid implicit buffers transfer

- While mapping a buffer, use the flags wisely:
  - map: `CL_MAP_WRITE_INVALIDATE_REGION`
    - The runtime may not need to copy the buffer over the PCIe to the host
  - map: `CL_MAP_READ`
    - The matching unmap is a NOOP

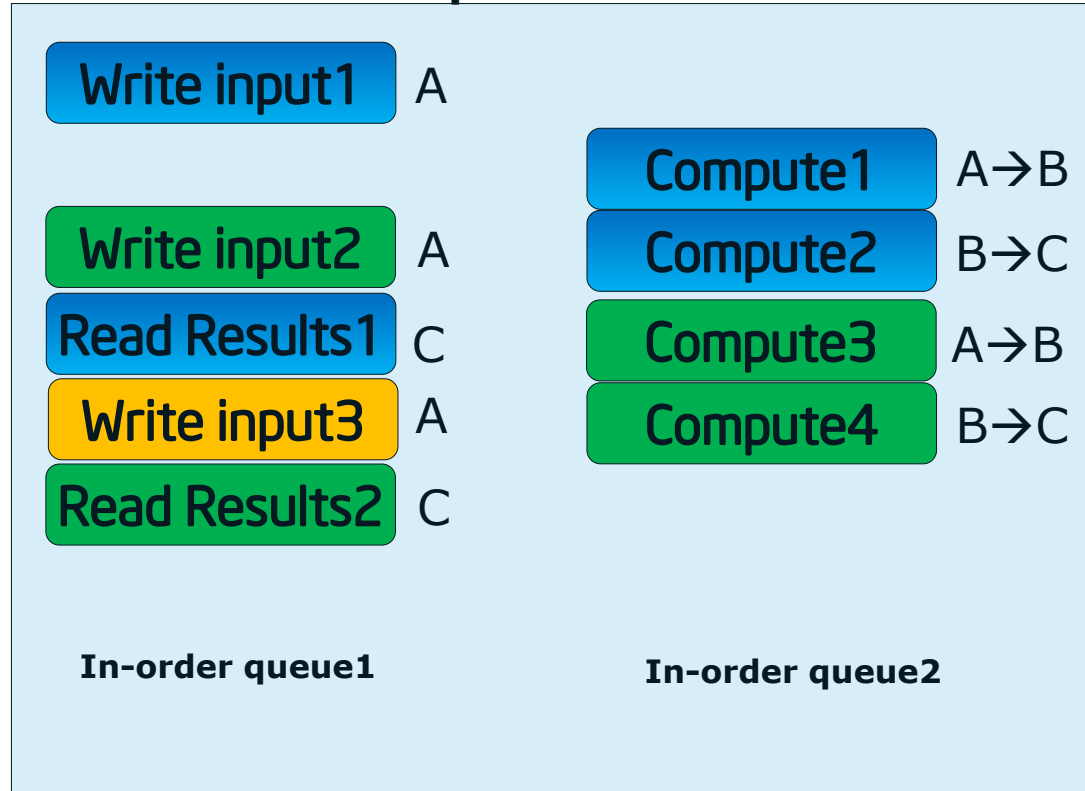
# Host-device efficiency

## Overlap Compute and Transfer

### Naïve



### Optimized



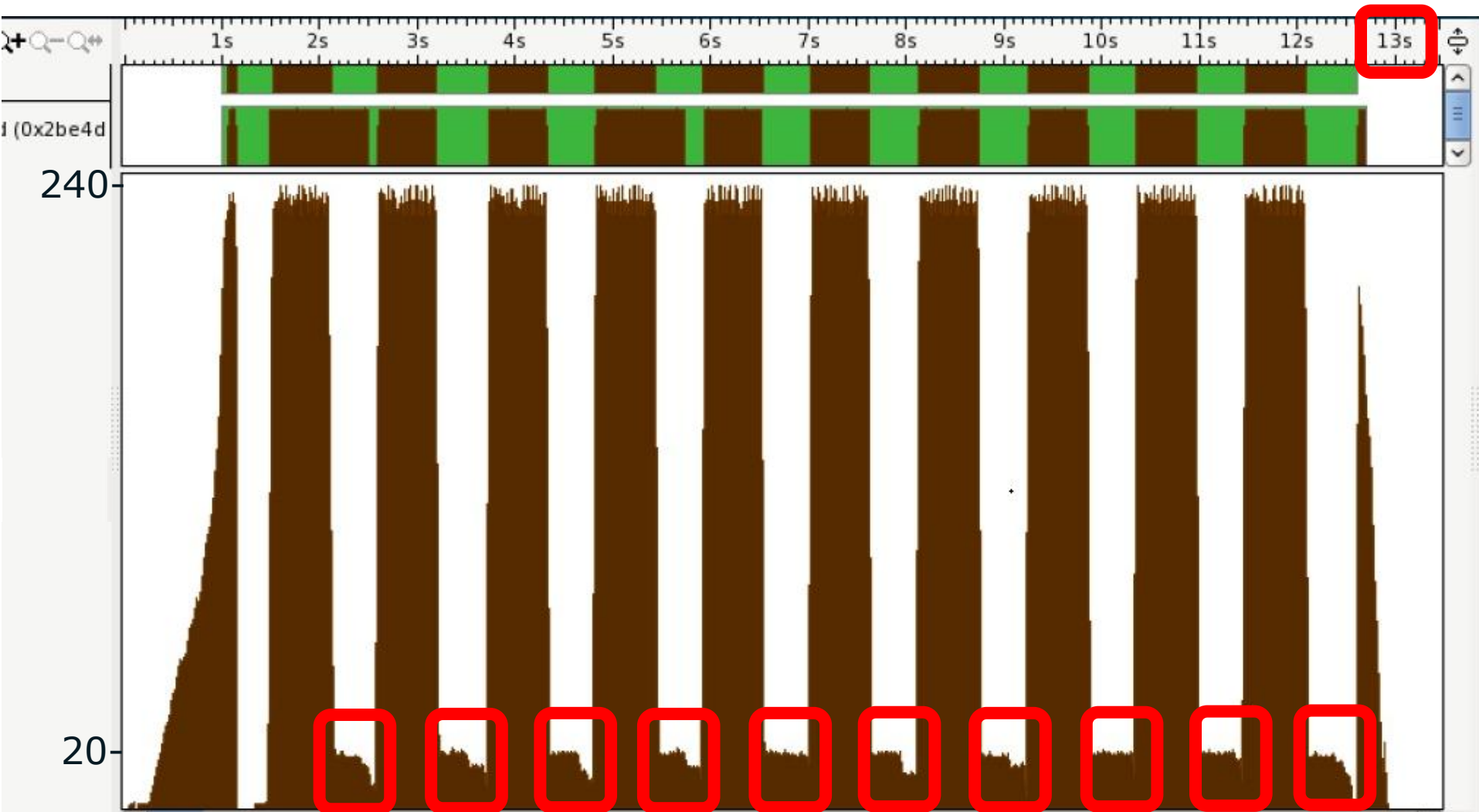
- A, B, C are buffers
- Parallel compute and transfer through 2 in-order queues

# Multi-threading in many core environment

- Core/threads utilization
- The NDRange tail effect (load balancing)
- Task scheduling overhead

# The NDRange Tail Effect

## In-order queue, 260 WGs, 10 repeats



# Feed the Beast

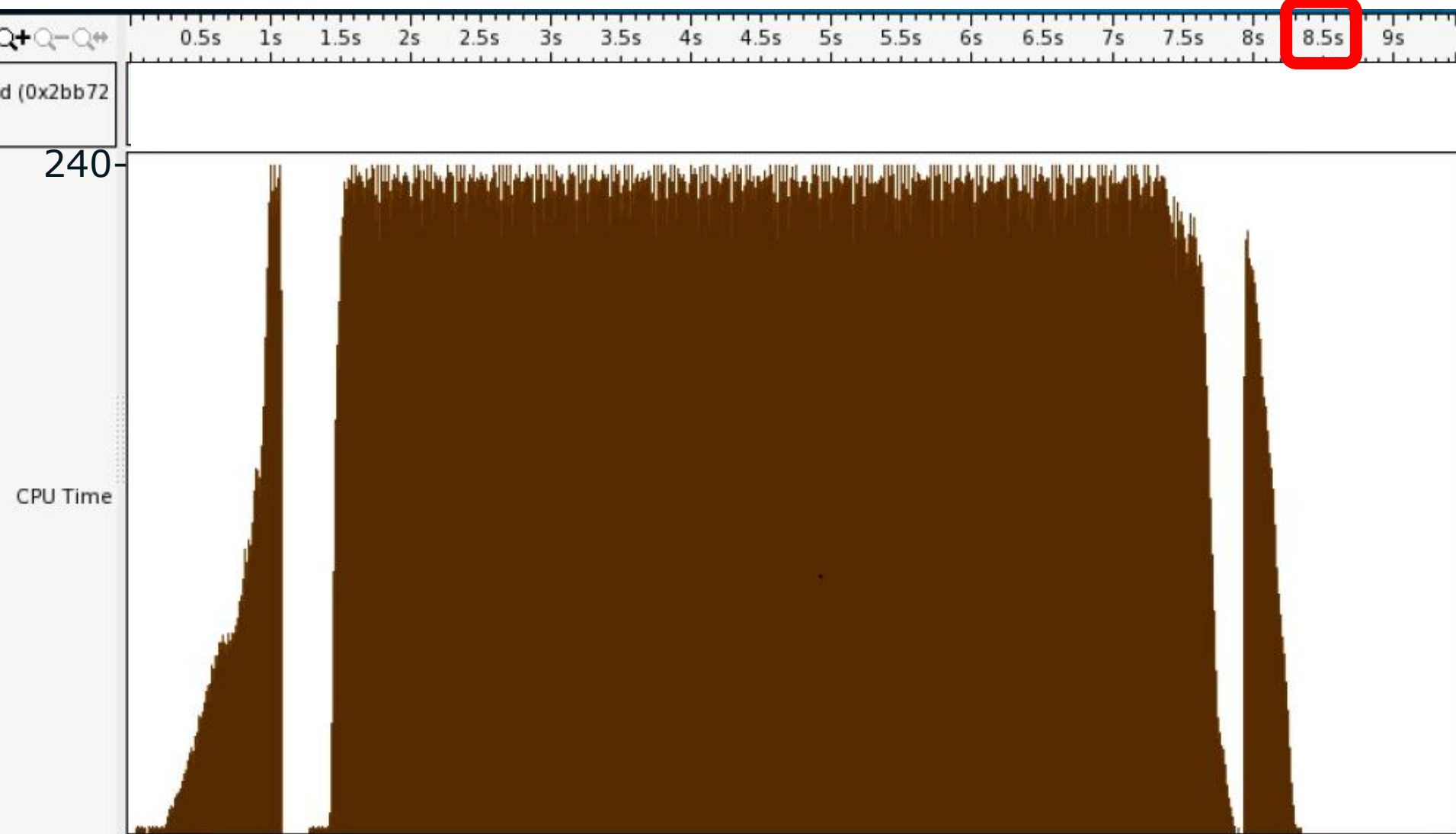
The key:

- The application needs to feed 240 threads for full utilization
- Dependent NDRange don't overlap
- Each NDRange should include enough WGs
  - >1000 WGs should allow dynamic load balancing
  - 240 is the bare minimum
  - 241 would take twice as long – NDRange tail effect

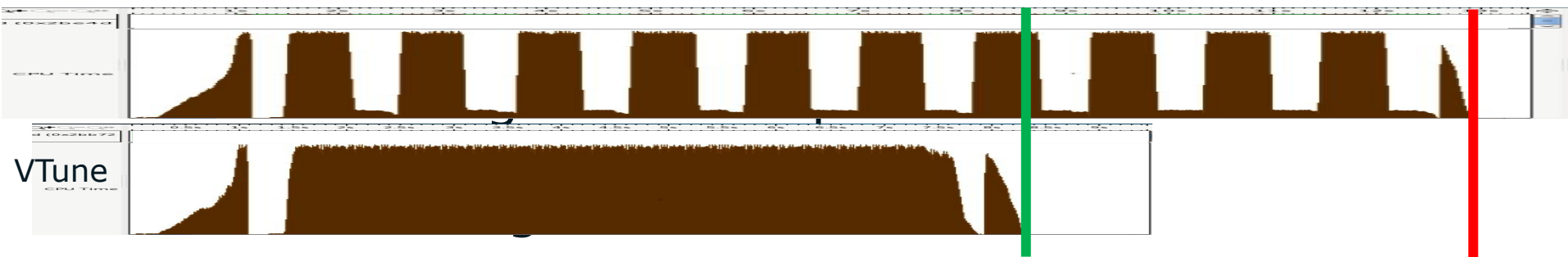


# The NDRange Tail Effect

## Out-of-order queue, 260 WGs, 10 repeats



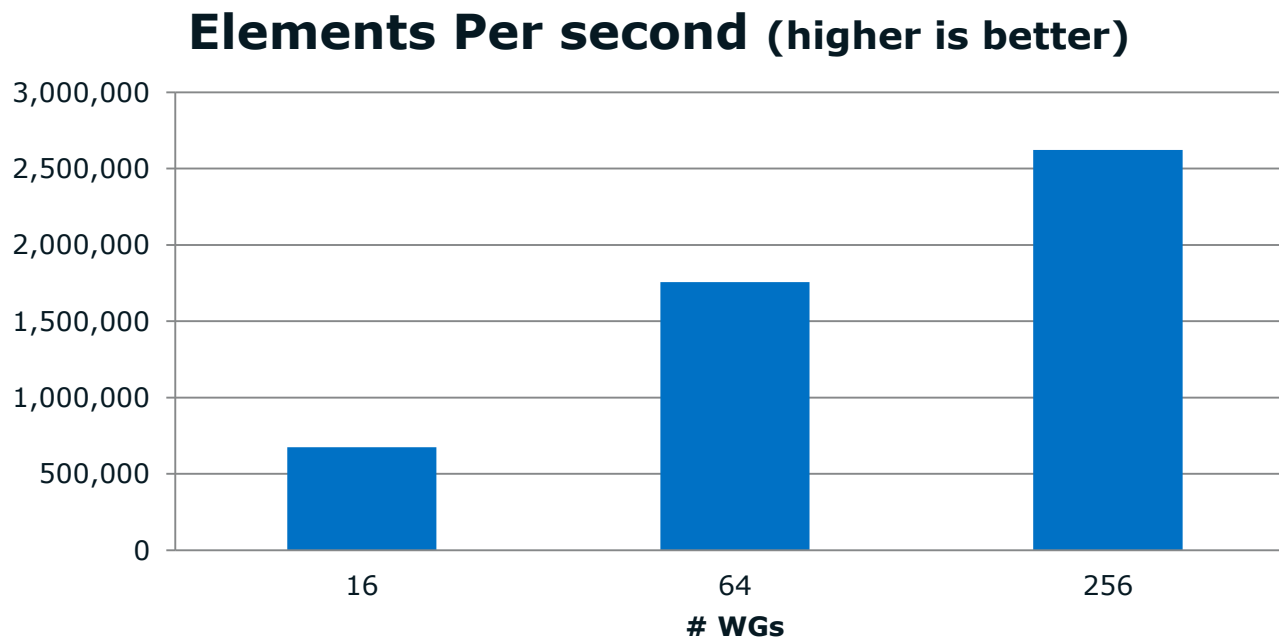
# Hiding the NDRange Tail Effect



- Overlap the tail with the next NDRange
- Use OOO queue

# Performance vs WG Count

- Internal workload example
- Fixed WG size
- Total problem size increases → WG count increases

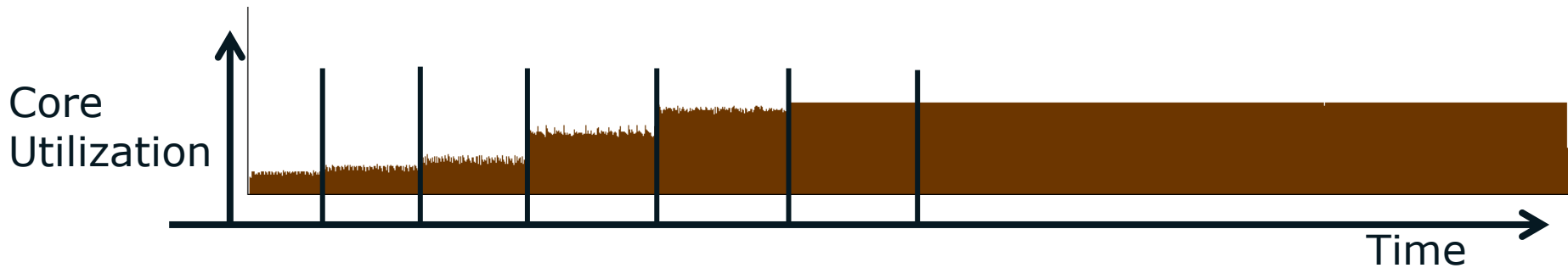


# Under-Utilization Example:

## SG++, Sparse Grid classification benchmark

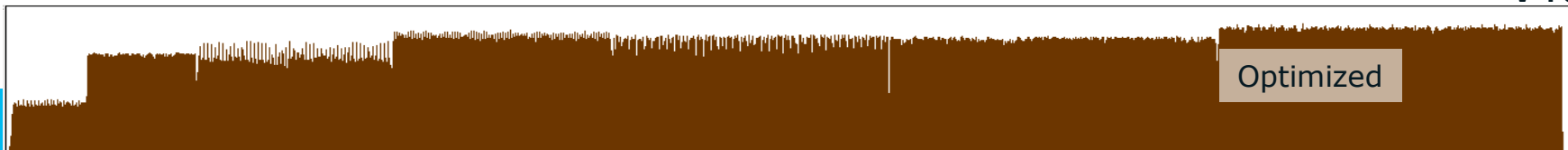
Alex Heinecke, Technical University of Munich

Iterative algorithm



- The input size grows with iterations
- With WG size of 64, there are not enough WGs in the first iterations
- We reduced the WG size to 16 → WG count increased
  - Improved the first iterations utilization
- The kernels include an explicit huge loop

VTune





# Task Scheduling Overhead

- Xeon Phi relies on SW to schedule threads and tasks
- Overhead scheduling 240 threads
  - Noticeable mainly in light-weight WGs
  - We are working to reduce this overhead (not eliminate)
- What is a light-weight WG?
  - Only few computations per work-item
  - Small local\_size

```
__kernel  
void array_mul(  
    __global const float *a,  
    __global const float *b,  
    __global float *c)  
{  
  
    int i = get_global_id(0);  
    c[i] = a[i] * b[i];  
}
```

# Detecting Scheduling Overhead

Grouping: Function / Call Stack

Function / Call Stack	CP. Ti.  	Instructions Retired	CPI Rate	Module
▶ PreScanStoreSumKernel	488.642s	86,030,000,000	6.19	[Dynamic code]
▶ pop	180.101s	20,880,000,000	9.402	libtbb_preview.so.2
▶ UniformAddKernel	163.174s	10,470,000,000	16.98	[Dynamic code]
▶ [vmlinux]	132.046s	5,240,000,000	27.468	vmlinux
▶ [TBB Scheduler Internals]	121.963s	11,100,000,000	11.977	libtbb_preview.so.2
▶ load with acquire	91.046s	430,000,000	230.791	libtbb_preview.so.2
Selected 1 row(s):	488.642s	86,030,000,000	6.191	

Vtune  
view

- The kernels are associated with the "Dynamic Code" module
- Anything else is "overhead"
- 652 sec in kernels
- 524 sec in "overhead"
- if ("overhead" > kernels\_time/5)
  - Investigate the overhead

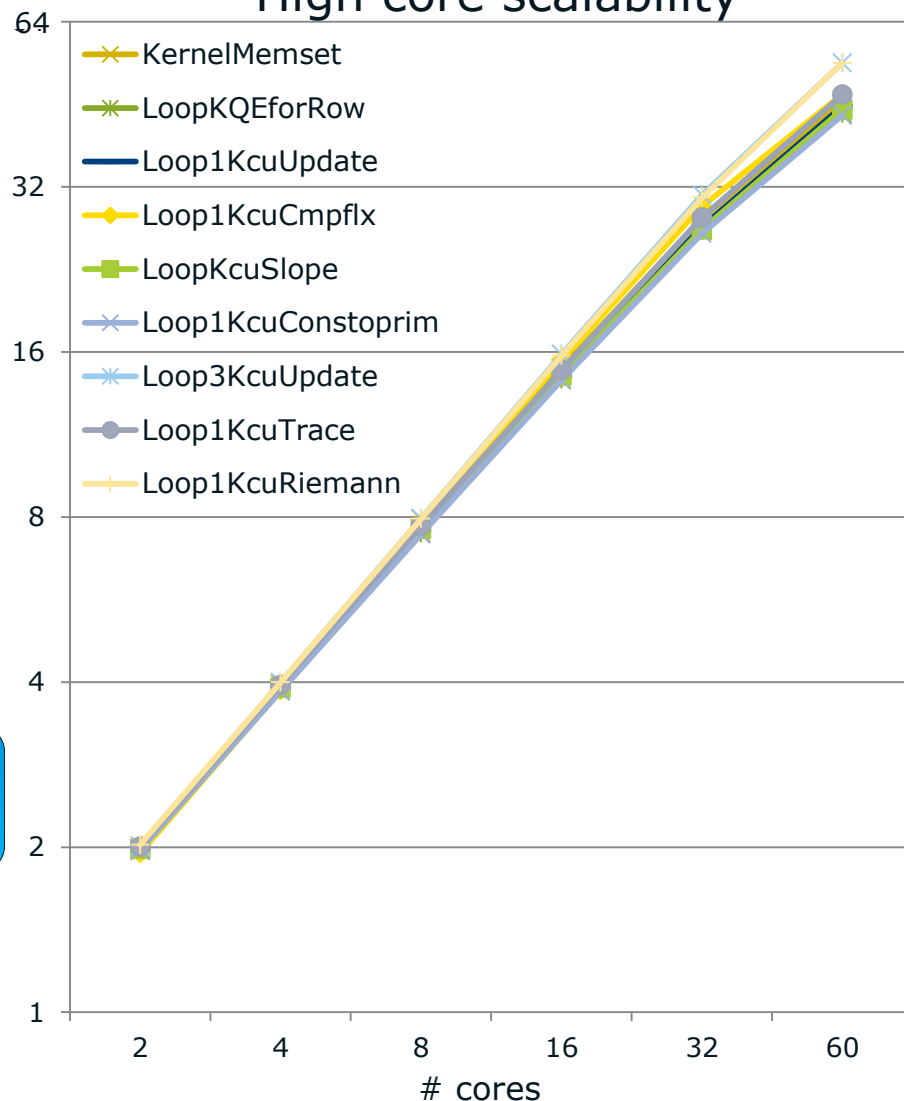
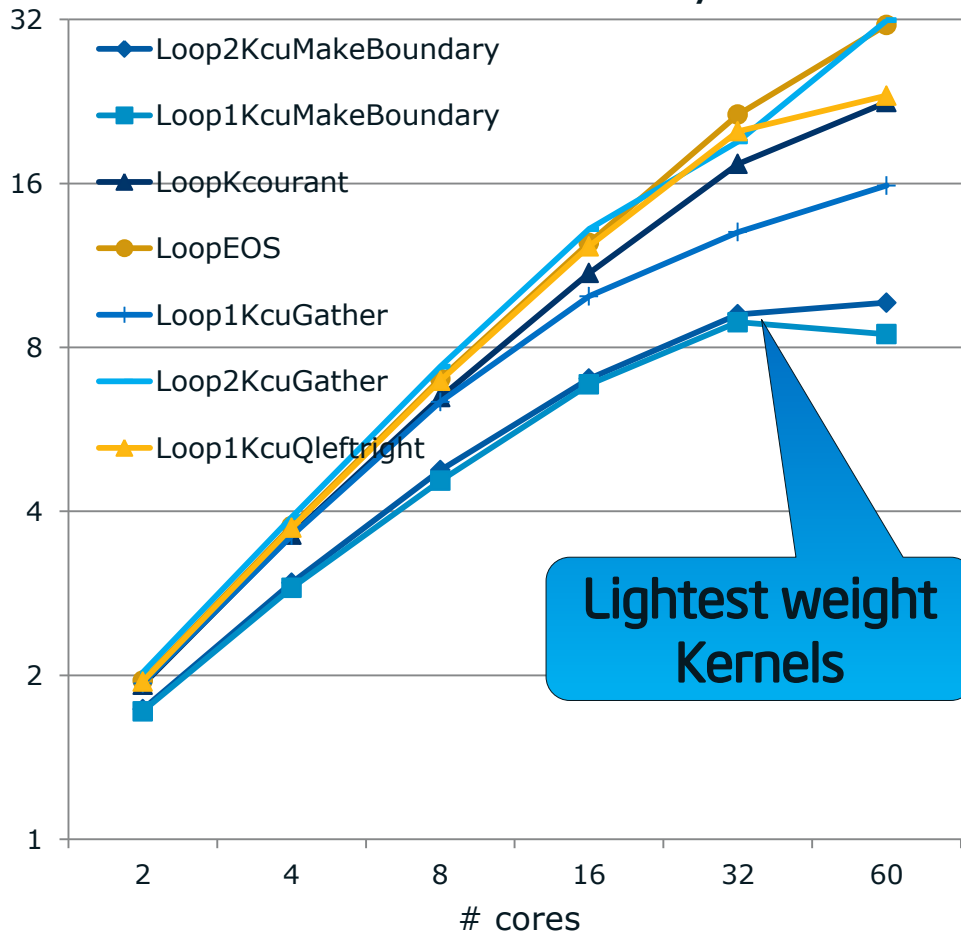
# Scalability Graph - Hydro miniapp

## Guillaume Colin de Verdière, CEA, France

<https://github.com/HydroBench/Hydro>

### High core scalability

### Low core scalability



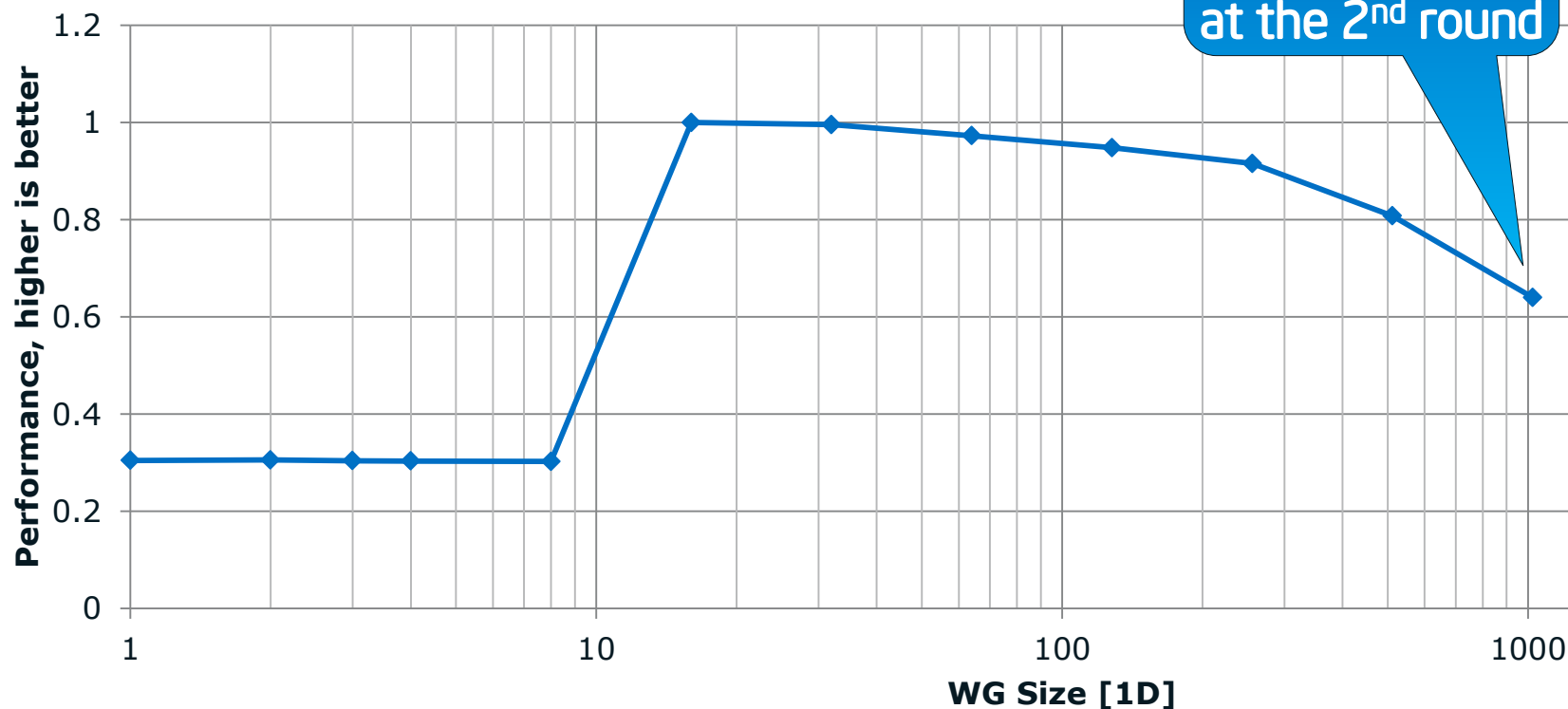
Lightest weight  
Kernels

# Performance Example – BUDE

Simon McIntosh-Smith, University of Bristol

James Price, University of Bristol

293 WGs  
187 idle threads  
at the 2<sup>nd</sup> round



- 300K WI, heavy-weight kernel, diverged branches
- No vectorization with WG size lower than 16
- Load-balancing and NDRange tail impact

32 Measured on pre-production Xeon Phi part



# Workgroup Size Summary

- NDRange with local\_size NULL works well in most cases
- Minimum LOCAL\_SIZE 16 at dimension zero
- LOCAL\_SIZE at dimension zero a multiply of 16
- Total WG\_COUNT higher than 1000

WG_COUNT	Upper bound HW thread Utilization
16	7%
100	42%
Full (240)	100%
240+1	~50%
1,000	~100%
12,000	100%

LOCAL_SIZE (Dimension zero)	Vector Lane Utilization
8	6%
15	6%
16	100%
17	53%
32	100%
1000	89%
1024	100%

# Local memory and barriers avoidance

- Intel® Xeon Phi™ Coprocessor doesn't distinguish between local and global memory
  - It includes coherent x86-like caching system
  - “local memory” is allocated in regular memory
  - Using local memory just adds another memory copy and work-item synchronization (barriers)
- Xeon Phi includes no HW support for Barriers
  - Barriers are emulated by SW
- Recommendation: Avoid using local memory and barriers
  - Doing so would also simplify the code

# Extracted Example from SG++

## Local memory and barriers removal

Alex Heinecke, Technical University of Munich

```
int GIdx = get_global_id(0);
int LIdx = get_local_id(0);
__local double locData[64];
__local double locSource[64 ];

for(int i = 0; i < sourceSize; i+= 64 )
{
    locData[LIdx] = ptrData[i+LIdx];
    locSource[LIdx] = ptrSource[i+LIdx];
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int k = 0; k < 64 ; k++)
    {
        myResult += DoWork(
                        locSource[k],
                        locData[k],
                        ptrLevel[GIdx]
                    );
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
ptrResult[globalIdx] = myResult;
```

```
int GIdx = get_global_id(0);
int LIdx = get_local_id(0);

for(int i = 0; i < sourceSize; i++)
{
```

**Faster on Xeon Phi**

```
myResult += DoWork(
                ptrSource[i],
                ptrData[i],
                ptrLevel[GIdx]
            );
}
ptrResult[globalIdx] = myResult;
```

# Implicit vectorization

- Recap: Implicit vectorization
- Diverged control-flow
- Gather/scatter
- Bounds check (early exit)
- Implicit WI Loop Tail

# Implicit vectorization

## Recap

```
__Kernel ABC(...)  
For (int i = 0; i < get_local_size(2); i++)  
  For (int j = 0; j < get_local_size(1); j++)  
    For (int k = 0; k < get_local_size(0); k += VEC_SIZE)  
      Vector_Kernel_Body;
```

- Implicit vectorization over dimension zero of the NDRange
- Don't vectorize manually!

# Workgroup Vectorization: Diverged Branches

## Uniform Branch

The compiler can prove that all the WIs within the vector take the same branch

```
//isSimple is a kernel argument
int GID = get_global_id(0);
if (isSimple == 0)
    res = buff[GID];
```

## Diverged Branch

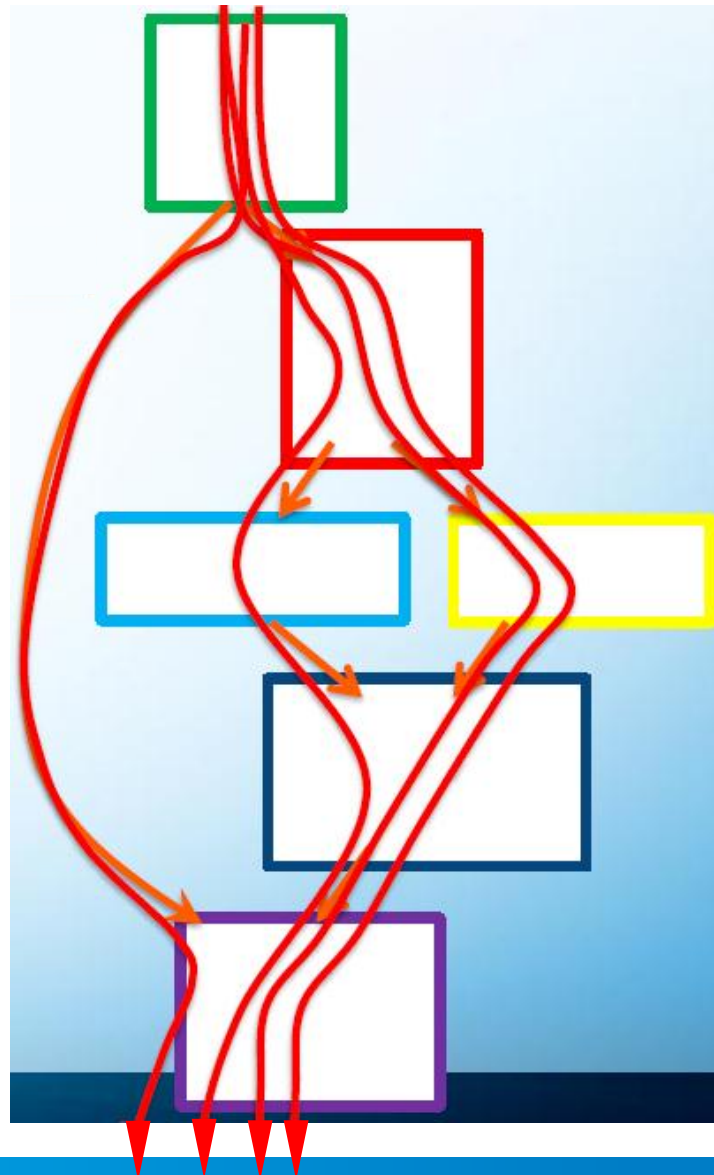
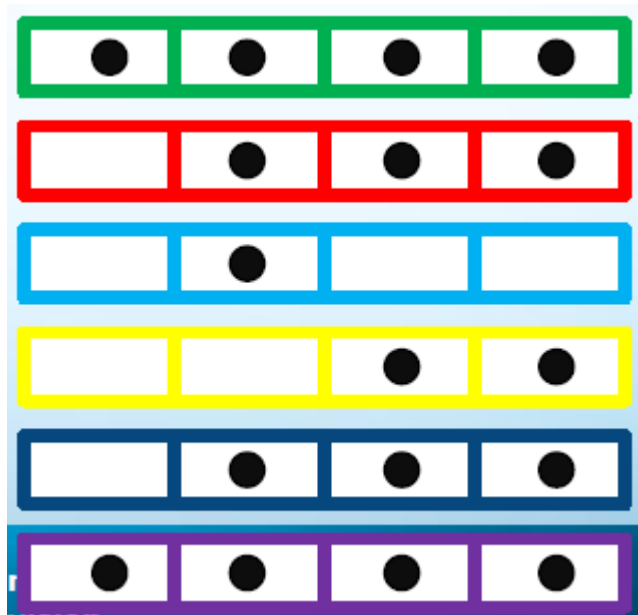
The compiler cannot prove that the branch is uniform

```
int GID = get_global_id(0);
if (GID == 0)
    res = -1;
```

- Branches dependent on `WI_ID[0]` are diverged between work-items
- Simple solution: Don't vectorize

# Predicating (flattening) Diverged Branches

- Predication flattens the control-flow and executes both the „then“ and „else“.
- Diverging CF reduces the utilization of vector instructions.
- Predication adds masking-overhead.



# Workgroup Vectorization: Diverged Branch Predication (if-conversion)

Original Diverged Branch	Automatic Predication (pseudo code)
<pre>int GID = get_global_id(0); if (GID == 0)     res = -1; else{     res = sqrt(buff[GID]);     res += arg1; }</pre>	<pre>int GID = get_global_id(0); mask = (GID == 0); res_then = -1; res_else = sqrt(buff[GID]); res_else += arg1; res = Select(res_then, res_else, mask);</pre>

[http://llvm.org/devmtg/2011-11/Rotem\\_IntelOpenCLSDKVectorizer.pdf](http://llvm.org/devmtg/2011-11/Rotem_IntelOpenCLSDKVectorizer.pdf)

Ralf Karrenberg & Sebastian Hack - Saarland University:

[http://www.cdl.uni-saarland.de/papers/karrenberg\\_wfv.pdf](http://www.cdl.uni-saarland.de/papers/karrenberg_wfv.pdf)

[http://www.cdl.uni-saarland.de/papers/karrenberg\\_opencl.pdf](http://www.cdl.uni-saarland.de/papers/karrenberg_opencl.pdf)



# Workgroup Vectorization: Predication + bypass

Original Diverged Branch	Automatic Predication (pseudo code) + bypass
<pre>int GID = get_global_id(0); if (GID == 0)     res = -1; else{     res = sqrt(buff[GID]);     res += arg1; }</pre>	<pre>int GID = get_global_id(0); mask = (GID == 0); res_then = -1; <b>if(mask != 0)</b>     res_else = sqrt(buff[GID]);     res_else += arg1; <b>endif</b> res = Select(mask, res_then, res_else);</pre>

- Tradeoff between the cost of the branch and the saving
- Included in the current release
- Will be improved in the next release

# Implicit vectorization

## Diverged branches

- Diverged branches add performance penalty
  - Masks management
  - Low lane utilization
  - Expensive memory accesses
- Avoid branches
  - Algorithmic changes
  - Use uniform iteration space
  - Kernel specialization

# Implicit Vectorization Example

## Use uniform iteration space to avoid branches

```
#define INPUT_SIZE...
#define OUTPUT_SIZE (INPUT_SIZE/2)
float inp[INPUT_SIZE];
float out[OUTPUT_SIZE];

global_size[0] = INPUT_SIZE;
```

```
#define INPUT_SIZE...
#define OUTPUT_SIZE (INPUT_SIZE/2)
float inp[INPUT_SIZE];
float out[OUTPUT_SIZE];

global_size[0] = INPUT_SIZE/2;
```

```
int Gidx = get_global_id(0);
float sum;

if (Gidx % 2 == 0)
{
    sum = inp[Gidx]+inp[Gidx+1];
    out[Gidx/2] = sum;
}
```

```
int Gidx = get_global_id(0);
float sum;

sum = inp[2*Gidx]+inp[2*Gidx+1];
out[Gidx] = sum;
```

\*Local size is even number

# Implicit Vectorization Example

## Construct the NDRange space to avoid branches on ID0

```
global_size[] = {K, L};  
local_size[] = {M, N};
```

```
//switch the implicit loops  
global_size[] = {L, K};  
local_size[] = {N, M};
```

```
int Gid0 = get_global_id(0);  
int Gid1 = get_global_id(1);
```

```
float res;
```

```
if (!isError(buff1[Gid0])) //Diverged
```

```
{  
    res= Compute(  
                buff1[Gid0],  
                buff2[Gid1]  
                )  
}
```

```
int Gid0 = get_global_id(0);  
int Gid1 = get_global_id(1);
```

```
float res;
```

```
if (!isError(buff1[Gid1])) //Uniform
```

```
{  
    sum = Compute(  
                buff1[Gid1],  
                buff2[Gid0]  
                )  
}
```

# Implicit Vectorization

## Diverged branch – Dynamic uniformity matters

Consider an unavoidable diverged branch

```
if (buff[id0] > 0)
{
    // compute positive number
}
else{
    // compute negative number
}
```

Both **then** and **else** include bypass

Few input scenarios:

- buff[] is entirely positive
  - Randomly spread values
  - Sorted smallest to largest
  - Each chunk of 1024 elements is sorted
- 
- Dynamic uniformity improves vector lane utilization
  - In some cases, (partial) sorting can be beneficial

# Gather and Scatter Operations

- The compiler generates scatter/gather on non-consecutive memory accesses
- Gather and Scatter instructions use int32 indices
- `get_global_id()` is the source of indices
- Guess what?
  - `size_t get_global_id (uint dimindx)`
  - `size_t` is unsigned int64 on Xeon Phi
- The compiler needs to safely cast uint64 to int32
  - Or give-up using gather or scatter

# Helping the Compiler generate Gather and Scatter Operations

- Cast IDs to signed int
- Avoid pointers manipulations
  - ~~myBuff = buff + arg;~~
- Use array notations
  - Buffer[id]
- Indirect memory access is hard to track
  - ~~Buffer[A[id]]~~

# Bounds Check: Early-Exit and Late-Start Optimization

Original kernel

```
__kernel
void abc(...)
{
    size_t id = get_global_id(0);
    if(id > LAST_ID) //Diverged
        return;

    // Rest of kernel
}
```

Pseudo naïve generated code

```
void abc(...)
{
    for (int k = sgid; k <= lgid; k+= VEC_SIZE)
    {
        if(id > LAST_ID)
            return;
    }
    // Rest of vectorized and MASKED kernel
}
```

Pseudo optimized generated code

```
void abc(...)
{
    for (int k = sgid; k <= MIN(lgid, LAST_ID); k+= VEC_SIZE)
    {
        // Rest of vectorized kernel (NON-MASKED)
    }
}
```



# Early-Exit and Late-Start Optimization

## What's the problem?

Original kernel

```
__kernel
void abc(...)
{
    int id = get_global_id(0);

    if(id > LAST_ID)
        return;

    // Rest of kernel
}
```

- What's the semantics of this kernel?
- Which work-items should reach beyond the "return"?
- $0 \leq ID \leq LAST\_ID$
- What about  $ID == 0x80000002$ ?
- The IF condition doesn't define suffix

Recommendations:

- Use ID bounds check only when required
- Keep the ID bounds check `size_t`

```
__kernel
void abc(...)
{
    size_t id = get_global_id(0);

    if(id > LAST_ID)
        return;

    int_id = (int)id;
    // Rest of kernel
}
```

# Implicit WI Loop Tail

```
void myKernel(...)
{
    int k;
    for (int k = 0; k < get_local_size(0); k+= VEC_SIZE)
        Vector_Kernel_Body;
    k -= VEC_SIZE;
    for (; k < get_local_size(0); k++)
        Scalar_Kernel_Body;
}
```

- The tail is executed in scalar loop
- WG of size  $2*VEC\_SIZE$  executes faster than  $2*VEC\_SIZE-1$
- It's harder with "barriers"
  - Kernels with barriers execute vectorized only if WG size is divisible by  $VEC\_SIZE$
- Recommendation: favor `local_size[0]` divisible by  $VEC\_SIZE$

# Cache optimizations

- The memory subsystem is often the bottleneck
- In-order execution implies greater sensitivity to memory latencies
- Generic guidelines valid to Intel® Xeon Phi™ coprocessor too:
  - Reduce data size
  - Improve temporal and spatial locality
  - Apply tiling/blocking techniques to allow data re-use from caches

# Blocking Example

```
for (i1 = 0; i1 < N; i1 ++){
  for (i2=0; i2 < N; i2++) {
    OUT[i1] += compute(data[i1], data[i2]);
  }
}
```

Blocking reduces GDDR traffic significantly for a class of algorithms

```
for (i2 = 0; i2 < N; i2 += BLOCK_SIZE) {
  for (i1=0; i1 < N; i1 ++){
    for (i22=0; i22 < BLOCK_SIZE; i22 ++){
      OUT[i1] += compute(data[i1], data[i2 + i22]);
    }
  }
}
```

How large should BLOCK\_SIZE be?

The largest such that four blocks stay in the L2 cache

See our OpenCL GEMM sample

# Data layout and memory access pattern

- Data access pattern impacts the performance greatly
- Consecutive access is usually the fastest
- AOS/SOA tradeoffs

2:50

# Consecutive Access Within the WG Row/Column major

```
__kernel
void myKernel(...)
{
    int k = get_global_id(0);
    int i = get_global_id(1);
    A[i * ROW_ZISE + k] += B[k * ROW_ZISE + i];
}
```

Real kernel

Consecutive access

$A[i * ROW\_ZISE + k] += B[k * ROW\_ZISE + i];$

Strided access

```
void myKernel(...)
{
    int I, k;
    for (int i = 0; i < get_local_size(1); i++)
        for (int k = 0; k < get_local_size(0); k += VEC_SIZE)
        {
            A[i * ROW_ZISE + k]16 += B[k * ROW_ZISE + i]gather_16;
        }
}
```

Pseudo generated code

Recommendation: Prefer row major consecutive memory access

# Consecutive Access Within the WG

## 1D strided access

```
__kernel                                     Real kernel
void myKernel(...)
{
    int k = get_global_id(0);
    A[k] += B[5 * k]; Strided access
}
```

Consecutive access

```
void myKernel(...)                               Pseudo generated code
{
    int k;
    for (int k = 0; k < get_local_size(0); k += VEC_SIZE)
    {
        A[k]16 += B[5 * k]gather_16;
    }
}
```

Recommendation: Prefer consecutive access along dimension zero

# SoA vs AoS Data Layouts

SOA:

```
double POSITION_X[SIZE_OF_BUFFER];  
double POSITION_Y[SIZE_OF_BUFFER];  
double POSITION_Z[SIZE_OF_BUFFER];
```

SOA:

- Consecutive access translates to plain vector load/store
- May access to many pages simultaneously

AOS:

```
typedef struct{  
    double X;  
    double Y;  
    double Z;  
}POS;  
POS POSITION[SIZE_OF_BUFFER];
```

AOS:

- Consecutive access translates to strided gather/scatter
- Minimal simultaneous pages access

- SOA usually faster for consecutive access pattern
- AOS usually faster for random sparse access pattern
  - Random access translates to random gather for both
  - In random access, spatial locality much better with AOS



# Data Prefetching – Intel® Xeon Phi™ Coprocessor HW

- Data prefetching is critical
- L1 Data Cache – 32K per core
- L2 Data/Instruction cache – 512K per core
- HW Data prefetching to L2 cache
- SW Prefetching
  - Instructions(\*) for prefetching to the L1D and L2 caches
  - One cache line prefetch *or* gather prefetch
  - Prefetch in exclusive mode or not
- Prefetch instruction won't cause a page-fault!

Processor events for measuring prefetch effectiveness

\* <http://software.intel.com/sites/default/files/forum/278102/327364001en.pdf>

3:00

# SW Prefetching

## Auto-prefetching

- Identify strided memory access within a loop
- Estimate loop iteration duration
- Don't overload HW resources
- Insert prefetches to bring data on time to L2 and L1 caches
- Support vectorized code including gather/scatter operations

## Manual Prefetching

- When future iteration accesses are not predictable
- For non strided access
- For scalar code
- Accesses that progress in an outer loop
- Whenever auto-prefetching didn't happen

# How Can I help Prefetching?

- Prefer consecutive memory accesses along the inner most loop (implicit dimension zero or explicit kernel inner loop)
- Avoid pointer manipulations
- Process the data directly at the global buffers
- Use the “prefetch” built-in for your key kernel inputs and outputs
  - Important especially when the access pattern is not regular
  - Better batch few prefetch instructions together
  - As a start – add “prefetches” for the current iteration

# Controlling Auto-Prefetching

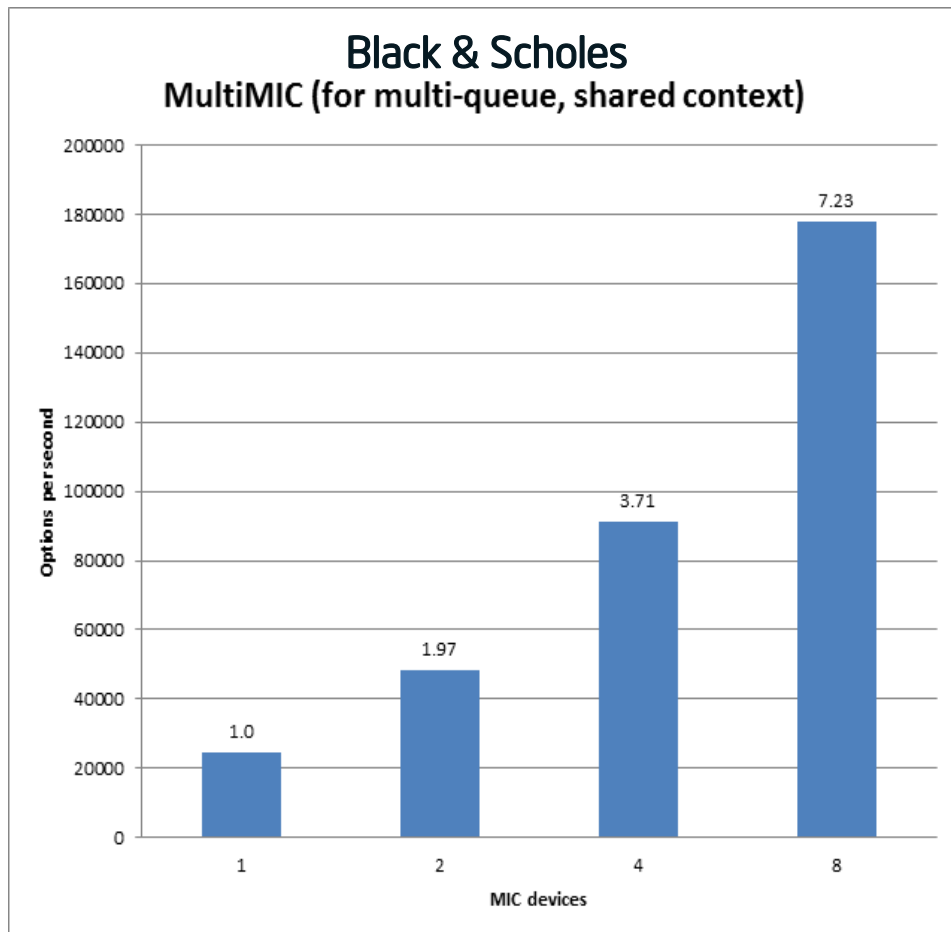
## Intel® Xeon Phi™ Specific

New clBuildProgram switch: *-auto-prefetch-level=[0-3]*

- 0: Disable SW auto-prefetching
  - 1: Limited SW auto-prefetching (linear address only)
  - 2: Safe SW auto-prefetching: 1 + masked memory access <default>
  - 3: Advance SW auto-prefetching: 2 + scatter/gather
- 
- Controls per kernel compilation
  - When Vtune hot-spot on scatter/gather instructions
    - Try using auto-prefetch level 3
  - When Vtune hot-spot on prefetch instructions
    - Try using auto-prefetch level 1
  - If these don't help, then add prefetch instructions manually based on Vtune's top memory accesses

# Multi Xeon-Phi Devices

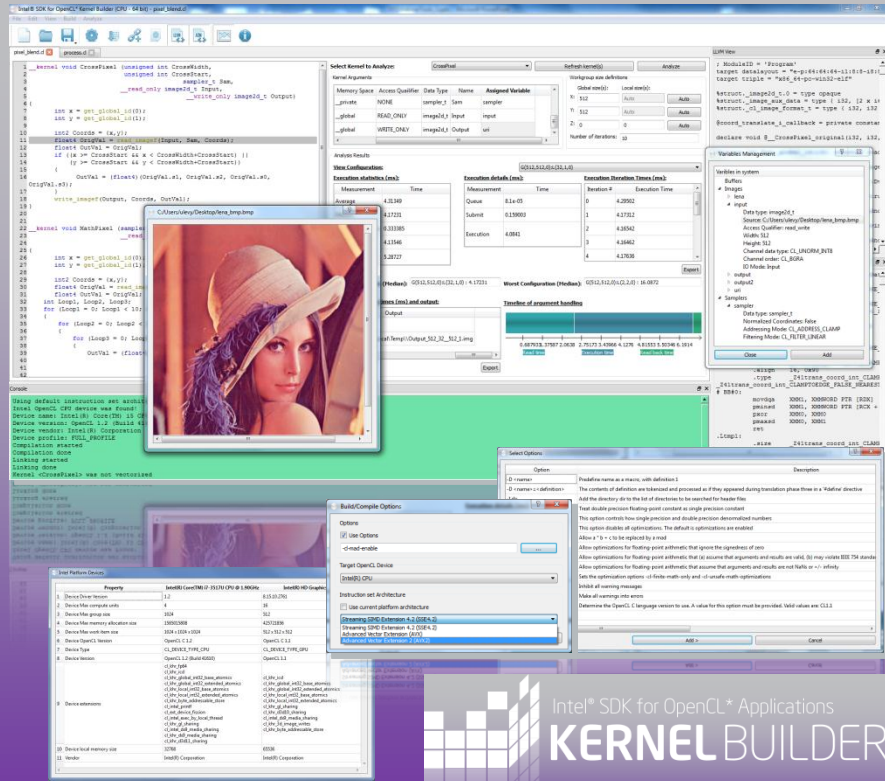
- Multi Xeon-Phi has just been introduced
- Optimized for shared-context
- Multi-applications
  - Each on a separate Xeon Phi
- Cluster with OpenCL
  - Nothing specific to OpenCL



# Moving to Tools . . .

# Kernel Builder

## OpenCL\* Kernels Design and Optimization Tool

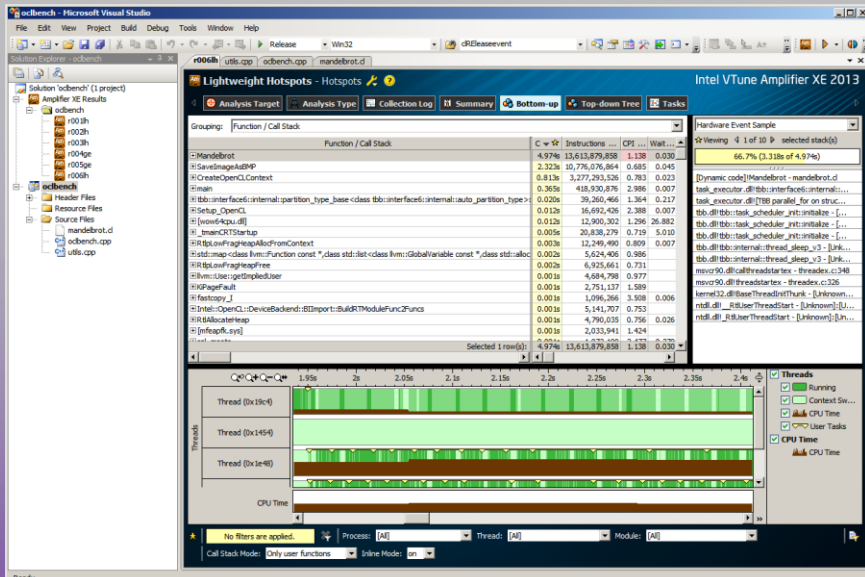


- Dynamic performance analysis & design tool with Offline Compilation support
- Assign variables to the kernel and test its correctness
- Analyze kernel performance based on:
  - group sizes
  - Optimization build switches
  - device used
- Supports MIC, CPU and GPU
- Available on Windows and Linux

Easy development of OpenCL\* Kernels for all Intel devices

<http://software.intel.com/en-us/articles/webinar-creating-and-optimizing-opengl-applications-with-intel-opengl-tools>

# Analyze OpenCL\* Applications with Intel® VTune™ Amplifier XE



## Universal Profiling Tool

- Easy, low-overhead Hotspots analysis
- Focused analysis: u-arch, parallelism, memory
- Interactive source/assembly
- Filter, group and sort your data
- Smooth Visual Studio\* integration
- Windows, Linux. Java, .NET, OpenCL\*, ...

## Special OpenCL\* support

- Understand how your kernel performs and why
- Optimize according to guidelines available with the Performance Optimization Guide



<http://software.intel.com/en-us/intel-vtune-amplifier-xe>



# Intel® VTune™ Amplifier XE Process/Module view

The screenshot displays the Intel VTune Amplifier XE interface. At the top, the title bar reads "Knights Corner Platform - Lightweight Hotspots 1 Hotspots viewpoint (change)". Below the title bar, there are several tabs: "Analysis Target", "Analysis Type", "Collection Log", "Summary", and "Bottom-up". The "Bottom-up" tab is selected and highlighted with a red box.

Below the tabs, there is a "Grouping:" dropdown menu with the text "Process / Module / Function / Thread / Call Stack" selected and highlighted with a red box. Below this, a table lists various processes and modules with their respective CPU times. The table has columns for "Process / Module / Function / Thread / Call Stack", "CPU Time", and "Task Time".

Process / Module / Function / Thread / Call Stack	CPU Time	Task Time
mic_server	2963.53s	
[Dynamic code]	2098.35s	
libtbb_preview.so.2	433.99s	
mic_server	167.15s	
_ocl_svml_b2.so.3.0	120.55s	
vmlinux	109.862s	3,600s
libc-2.14.90.so	20.645s	4,880s
libtbbmalloc.so.2	7.078s	
libpthread-2.14.90.so	5.456s	880s
sep3_8	0.221s	
libcoi_device.so	0.147s	
libstdc++.so.6.0.16	0.074s	
micscif	0s	80s
ramoops	27.576s	2,560s
coi_daemon	17.548s	1,520s
<b>Selected 1 row(s):</b>	<b>2963.539s</b>	<b>711,520s</b>

Below the table, there is a "Thread" section with a list of threads and a corresponding CPU time graph. The graph shows CPU time usage over a 70-second period. A red box highlights a significant period of high CPU activity between approximately 15s and 35s. The "CPU Time" label is also highlighted with a red box.

At the bottom of the interface, there are filter controls: "No filters are applied.", "Process: Any Process", "Thread: Any Thread", and "Module: A".

# Intel® VTune™ Amplifier XE Top-Down View (from all modules)

Call Stack	CPU Time: Total	Module	Instructions Retired: Total	CPI Rate: Total
<b>Total</b>	<b>3009.401s</b>		<b>100.0%</b>	<b>4.562</b>
Loop1KcuRiemann	600.406s	[Dynamic code]	36.6%	2.486
Loop1KcuTrace	339.760s	[Dynamic code]	9.8%	5.266
Loop3KcuUpdate	316.461s	[Dynamic code]	2.6%	18.264
tbb::internal::custom_	260.055s	libtbb_preview.so.2	7.3%	5.401
Loop1KcuConstoprime	154.323s	[Dynamic code]	4.9%	4.757
Loop1KcuQleftright	149.677s	[Dynamic code]	4.9%	4.677
LoopKcuSlope	136.922s	[Dynamic code]	7.0%	2.966
tbb::internal::generic_s	122.912s	libtbb_preview.so.2	2.1%	8.682
_ocl_svml_b2_sqrt16	119.816s	_ocl_svml_b2.so.3.0	6.2%	2.917
Loop1KcuCmpflx	104.774s	[Dynamic code]	3.6%	4.413
Loop2KcuGather	87.373s	[Dynamic code]	1.9%	6.890
Loop1KcuUpdate	75.134s	[Dynamic code]	1.7%	6.839
tasklet_hi_action	75.134s	vmlinux	0.1%	169.833
Loop1KcuGather	70.636s	[Dynamic code]	0.8%	13.686
LoopKComputeDeltat	57.143s	[Dynamic code]	2.6%	3.341
schedule	23.521s	vmlinux	0.3%	10.290
Selected 1 row(s):	3009.401s		100.0%	4.562

# Recommendations Summary

- Provide enough WGs to allow high core utilization
- Avoid light-weight kernels
- Avoid branches, especially diverging branches
- Use OOO queues
  - Parallel compute and transfer
  - More load-balancing
- Linear access is the fastest
- Use simple addressing []
- Prefer row major consecutive access
- Add the “prefetch” built-in when auto-prefetch is not enough

# Credits

Intel:

Anat Shemer

Maxim Shevtsov

Mikhail Letavin

Dmitry Budnikov

Adir Deri

Yariv Aridor

Evgeny Fiksman

Ohad Shacham

Mohammed Agabaria

Uri Levy

SG++: Alex Heinecke, Technical University of Munich

Hydro: Guillaume Colin de Verdière, CEA, France

BUDE: Simon McIntosh-Smith, University of Bristol

James Price, University of Bristol

# Resources

VISUAL COMPUTING  
**SOURCE**

DASHBOARD  
Getting Started

LEARN  
Case Studies  
Tech Articles  
Videos  
Events

TOOLS, SDKs, LIBs  
Product Detail

SAMPLES

FORUMS

BLOGS

RESEARCH  
Adv. Rendering

Intel AppUp

Dashboard > Tools, SDKs, LIBs > Intel® SDK for OpenCL® Applications XE 2013

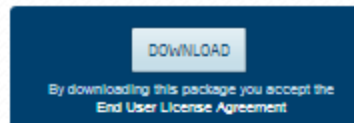
▼ | בחר שפה



Tweet 28

+1 6

Follow Product Mgr



## Introducing Intel® SDK for OpenCL® Applications XE 2013

Develop highly parallel applications using OpenCL® 1.2 for Intel® Xeon® processors, and Intel Xeon Phi™ coprocessors

The new Intel® SDK for OpenCL® Applications XE 2013 includes certified OpenCL 1.2 support for Intel Xeon processors, and Intel Xeon Phi coprocessors for Linux® operating systems. Targeted at developers of highly parallel applications including High Performance Compute (HPC), workstations, and data analytics, the new SDK broadens the parallel programming options on Intel architecture and allows developers to maximize data parallel application performance on Intel Xeon Phi coprocessors.

The Intel SDK for OpenCL Applications XE 2013 includes an OpenCL runtime API for Intel Xeon processors and Intel Xeon Phi coprocessors as well as tools, optimization guides, samples, and training content.

For Intel Xeon Phi coprocessor support, you must install Intel® Manycore Platform Software Stack (Intel® MPSS) Update 3 or higher. Available at <http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss>

## About the Intel® Xeon Phi™ Coprocessor

The Intel Xeon Phi coprocessor is the first product based on Intel Many Integrated Core Architecture (Intel® MIC architecture), and it targets highly parallel segments such as oil exploration, scientific research, financial analyses, and climate simulation. Intel MIC architecture combines many Intel CPU cores onto a single chip. Developers interested in programming these cores can use standard programming methods. The same OpenCL source code written for Intel Xeon processors can be reused on Intel Xeon Phi coprocessors with minimal modifications.

Intel OpenCL products matrix: [Compare and Download Products](#)

Target Processors	Target Operating System	OpenCL spec version	Target SDK

## Product Documents

- [Product Brief \[pdf\]](#)
- [Release Notes](#)
- [Installation Notes](#)
- [User Guide \[html\] \[pdf\]](#)
- [Optimization Guide \[html\] \[pdf\]](#)

## Support

- [Support Forum](#)
- [Frequently Asked Questions](#)

## Training

- [OpenCL® Code Samples](#)
- [OpenCL® Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor](#)
- [Workshop: Optimizing OpenCL applications for Intel® Xeon Phi™ Coprocessor](#)
- [Xeon Phi developer quick start guide](#)
- [Optimize with Intel VTune Amplifier XE](#)

# Resources

VISUAL COMPUTING  
**SOURCE**

DASHBOARD  
Getting Started

LEARN  
Case Studies  
Tech Articles  
Videos  
Events

TOOLS, SDKs, LIBs  
Product Detail

SAMPLES


FORUMS

BLOGS

RESEARCH  
Adv. Rendering

Intel AppUp

Dashboard > Tools, SDKs, LIBs > Intel® SDK for OpenCL® Applications XE 2013

▼ | בחר שפה 



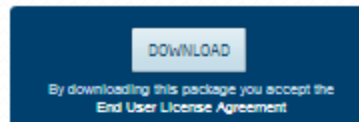
Tweet

28

+1

6

Follow Product Mgr



## Introducing Intel® SDK for OpenCL® Applications XE 2013

Develop highly parallel applications using OpenCL® 1.2 for Intel® Xeon® processors, and Intel Xeon Phi™ coprocessors

The new Intel® SDK for OpenCL® Applications XE 2013 includes certified OpenCL 1.2 support for Intel Xeon processors, and Intel Xeon Phi coprocessors for Linux® operating systems. Targeted at developers of highly parallel applications including High Performance Compute (HPC), workstations, and data analytics, the new SDK broadens the parallel programming options on Intel architecture and allows developers to maximize data parallel application performance on Intel Xeon Phi coprocessors.

The Intel SDK for OpenCL Applications XE 2013 includes an OpenCL runtime API for Intel Xeon processors and Intel Xeon Phi coprocessors as well as tools, optimization guides, samples, and training content.

For Intel Xeon Phi coprocessor support, you must install Intel® Manycore Platform Software Stack (Intel® MPSS) Update 3 or higher. Available at <http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss>

### About the Intel® Xeon Phi™ Coprocessor

The Intel Xeon Phi coprocessor is the first product based on Intel Many Integrated Core Architecture (Intel® MIC architecture), and it targets highly parallel segments such as oil exploration, scientific research, financial analyses, and climate simulation. Intel MIC architecture combines many Intel CPU cores onto a single chip. Developers interested in programming these cores can use standard programming methods. The same OpenCL source code written for Intel Xeon processors can be reused on Intel Xeon Phi coprocessors with minimal modifications.

Intel OpenCL products matrix: [Compare and Download Products](#)

Target Processors	Target Operating System	OpenCL spec version	Target SDK

### Product Documents

- [Product Brief \[pdf\]](#)
- [Release Notes](#)
- [Installation Notes](#)
- [User Guide \[html\] \[pdf\]](#)
- [Optimization Guide \[html\] \[pdf\]](#)

### Support

- [Support Forum](#)
- [Frequently Asked Questions](#)


### Training

- [OpenCL® Code Samples](#)
- [OpenCL® Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor](#)
- [Workshop: Optimizing OpenCL applications for Intel® Xeon Phi™ Coprocessor](#)
- [Xeon Phi developer quick start guide](#)
- [Optimize with Intel VTune Amplifier XE](#)

Copyright© 2012, Intel Corporation. All rights reserved.

\*Other brands and names are the property of their respective owners.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos

Optimization  
Notice 



# Resources



## Product Documents

[Product Brief \[pdf\]](#)

[Release Notes](#)

[Installation Notes](#)

[User Guide \[html\] \[pdf\]](#)

[Optimization Guide \[html\] \[pdf\]](#)

## Support

[Support Forum](#)

[Frequently Asked Questions](#)

## Training

[OpenCL\\* Code Samples](#)

[OpenCL\\* Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor](#)

[Workshop: Optimizing OpenCL applications for Intel® Xeon Phi™ Coprocessor](#)

[Xeon Phi developer quick start guide](#)

[Optimize with Intel VTune Amplifier XE](#)

# Thank You!

# Questions?



# 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.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

## 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

