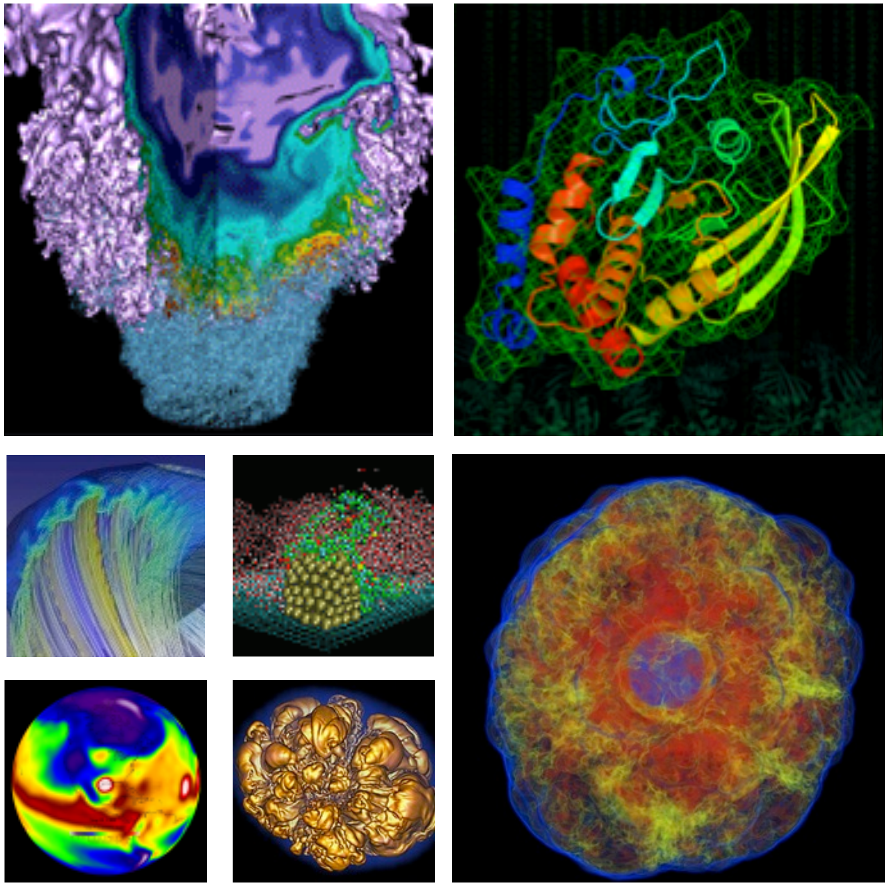
Enabling Application Portability across HPC Platforms:

An Application Perspective



Alice KONIGES[1], Tim MATTSON[2], Yun (Helen) HE[1], Richard GERBER[1]

1) Lawrence Berkeley National Laboratory, USA
2) Intel

September, 2015

Disclaimer

- The views expressed in this talk are those of the speakers and not their employers.
- I work with very smart people. Anything stupid I say is mine … don't blame my collaborators.

I work in Intel's research labs. I don't build products. Instead, I get to poke into dark corners and think silly thoughts… just to make sure we don't miss any great ideas.

Hence, my views are by design <u>far</u> "off the roadmap".

- This presentation is a "conversation" between two talks .. One from NERSC and one from me. Just to be clear, when a slide comes from "my talk" I always indicate that fact by putting a picture of me in a kayak on the slide in question.

Cori: A pre-exascale supercomputer for the Office of Science workload

- **System will begin to transition the workload to more energy efficient architectures**
- **Will showcase technologies expected in exascale systems**
  - Processors with many ‘slow’ cores and longer vector units
  - Deepening memory and storage hierarchies

Image source: Wikipedia

System named after Gerty Cori, Biochemist and first American woman to receive the Nobel prize in science.

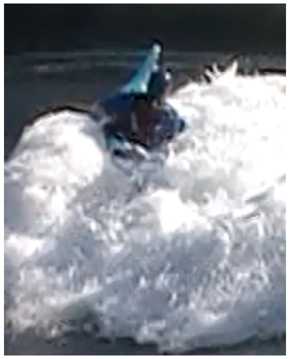
Cori: A pre-exascale supercomputer for the Office of Science workload

- **System will begin to transition the workload to more energy efficient architectures**
- **Will showcase technologies expected in exascale systems**

Image source: Wikipedia

System named after Gerty Cori, Biochemist and first American woman to receive the Nobel prize in science.

It is so nice that they named their machine after a chemist. Chemists rule!!!!

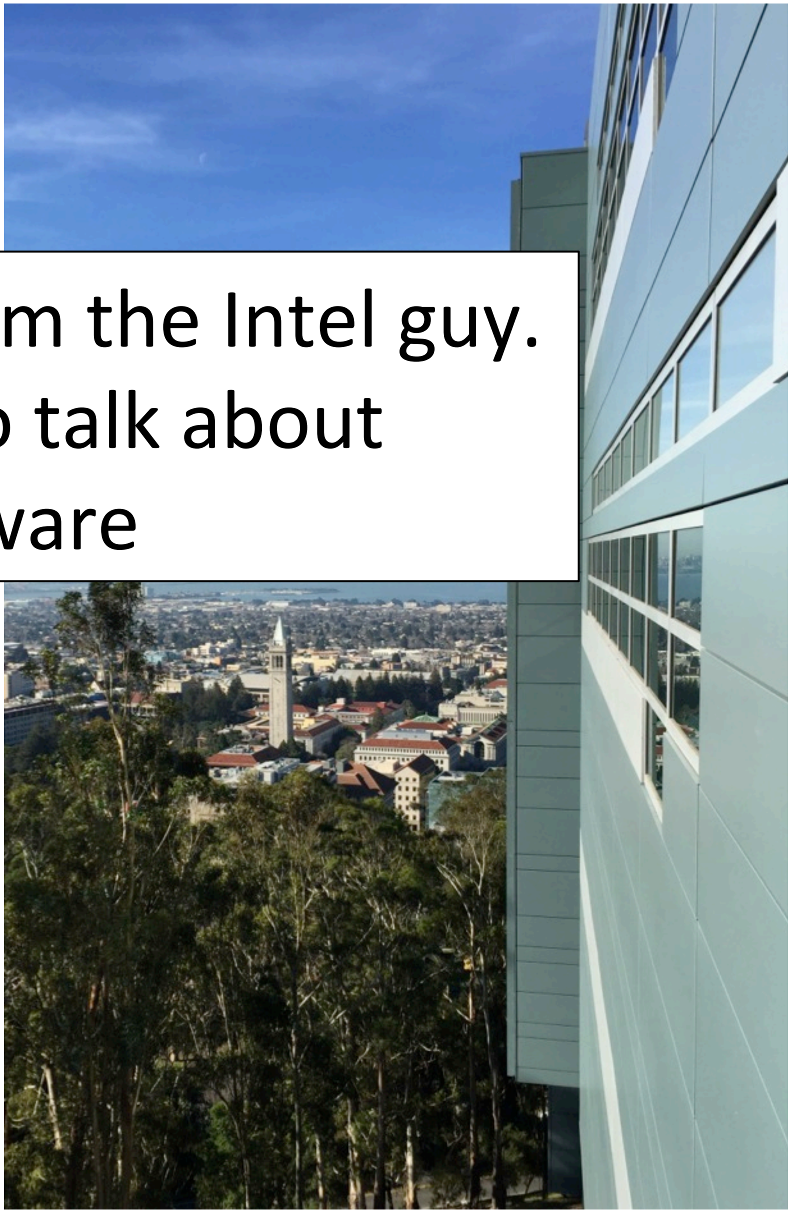


Cori Configuration – and a new home

- **Over 9,300 Knights Landing compute nodes**
  - Self-hosted, up to 72 cores, 16 GB high bandwidth memory
- **1,600 Haswell compute nodes as a data partition**
- **Aries Interconnect**
- **Lustre File system**
  - 28 PB capacity, >700 GB/sec I/O bandwidth
- **Delivery in two phases, summer 2015 and 2016 into new CRT facility**

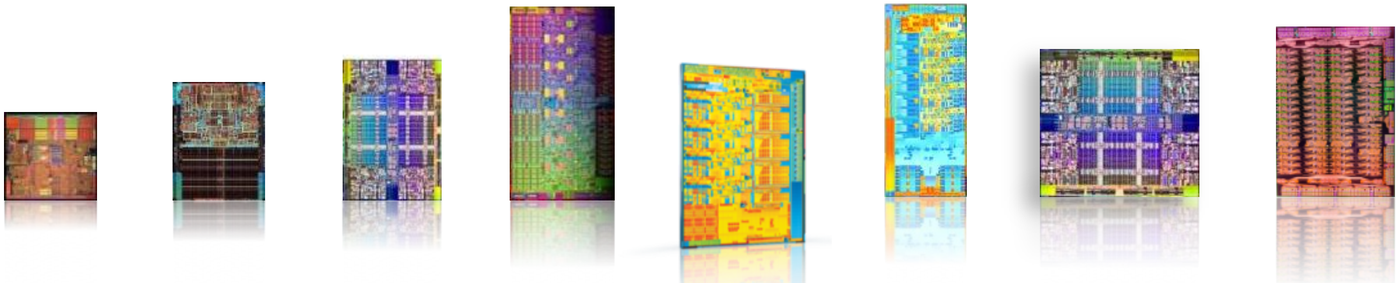
Cori Configuration – and a new home

- **Over 9,300 Knights Landing compute nodes**
  - Self-hosted, up to 72 cores, 16 GB high bandw
- **1,600 Haswe a data partit**
- **Aries Interc**
- **Lustre File system**
  - 28 PB capacity, >700 GB/sec I/O bandwidth
- **Delivery in two phases, summer 2015 and 2016 into new CRT facility**



Wait a minute. I'm the Intel guy. It's my job to talk about hardware

Increasing parallelism in Xeon and Xeon Phi

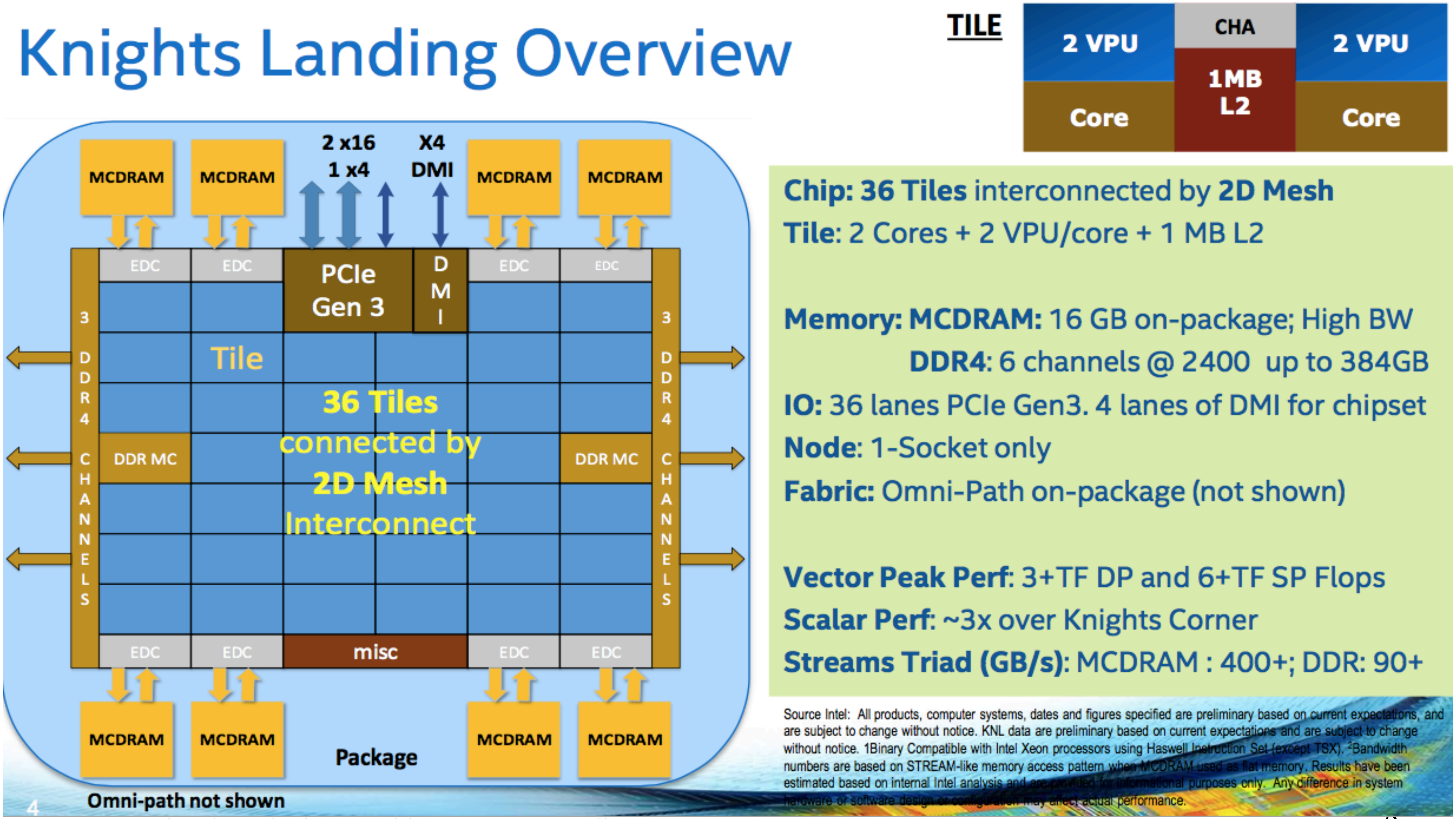


	Intel® Xeon® processor 64-bit series	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EX	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor **Knights Landing**[1]
Core(s)	1	2	4	6	8	12	18	61	*60+*
Threads	2	2	8	12	16	24	36	244	4x #cores
SIMD Width	128	128	128	128	256	256	256	512	*2x512*

**Product specification for launched and shipped products available on ark.intel.com.*

[1] *Not launched.*

Lots of cores with in package memory



Knights Landing Overview

Chip: 36 Tiles interconnected by **2D Mesh**

Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW

DDR4: 6 channels @ 2400 up to 384GB

IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset

Node: 1-Socket only

Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops

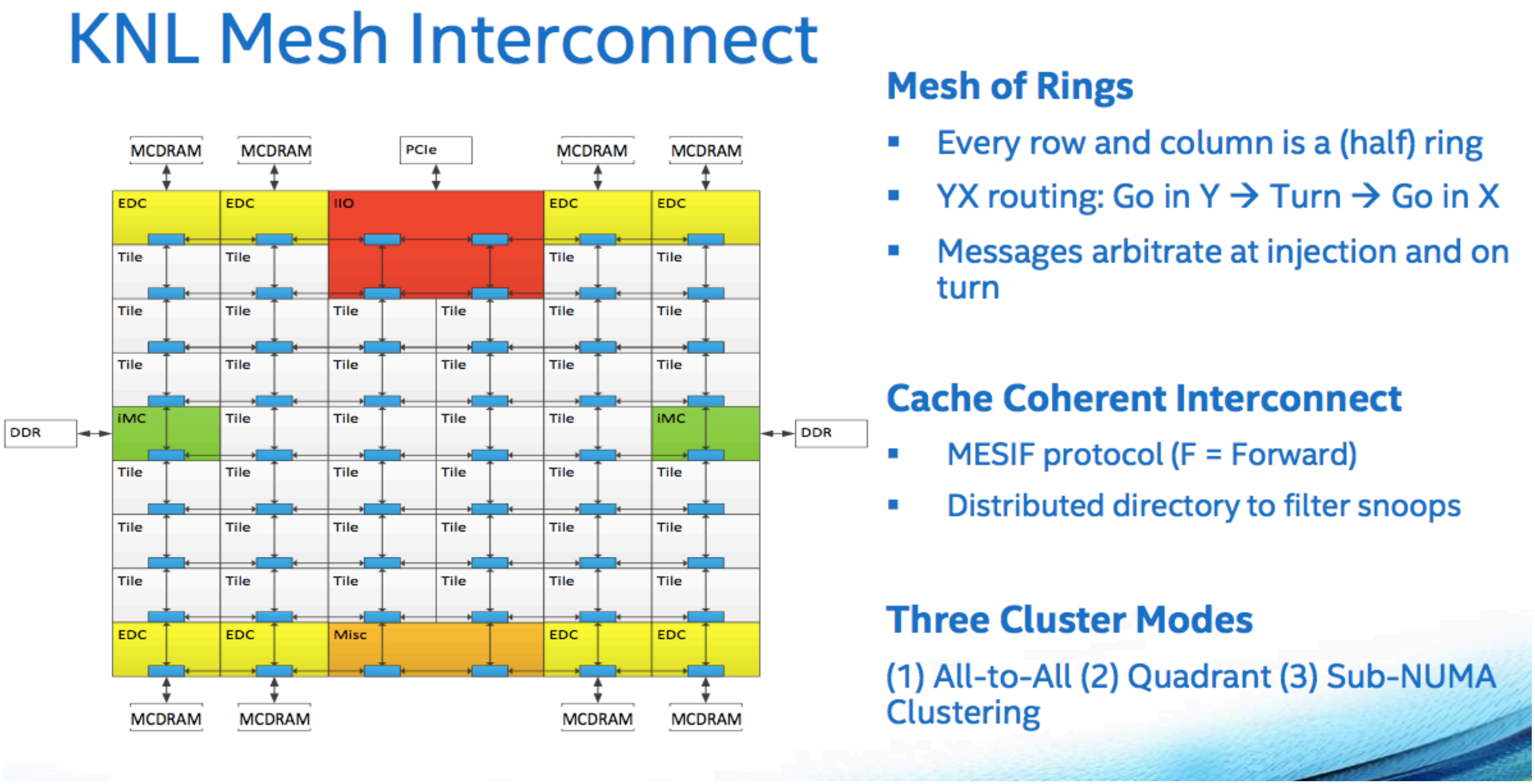
Scalar Perf: ~3x over Knights Corner

Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). [2]Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

Source: Avinash Sodani, Hot Chips 2015 KNL talk

Connecting tiles



KNL Mesh Interconnect

Mesh of Rings

- Every row and column is a (half) ring
- YX routing: Go in Y → Turn → Go in X
- Messages arbitrate at injection and on turn

Cache Coherent Interconnect

- MESIF protocol (F = Forward)
- Distributed directory to filter snoops

Three Cluster Modes

(1) All-to-All (2) Quadrant (3) Sub-NUMA Clustering

Source: Avinash Sodani, Hot Chips 2015 KNL talk

Network interface Chip in the package …

KNL w/ Intel® Omni-Path

Omni-Path Fabric integrated *on package*

First product with integrated fabric

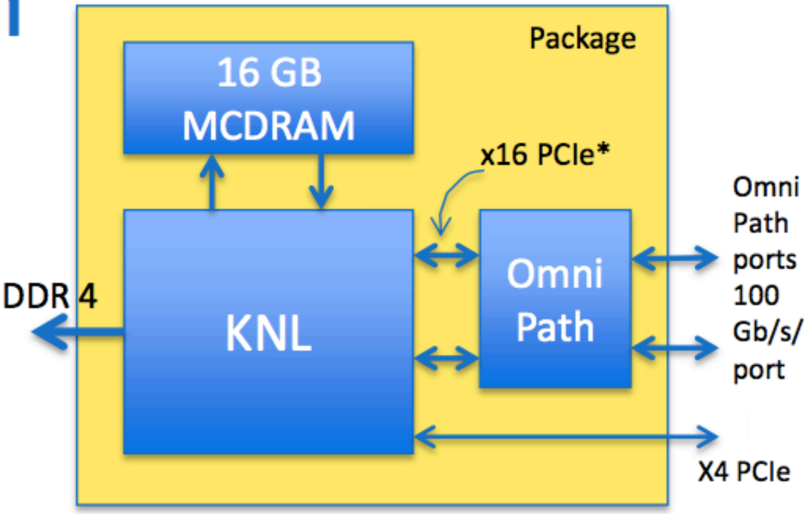
Connected to KNL die via 2 x16 PCIe* ports

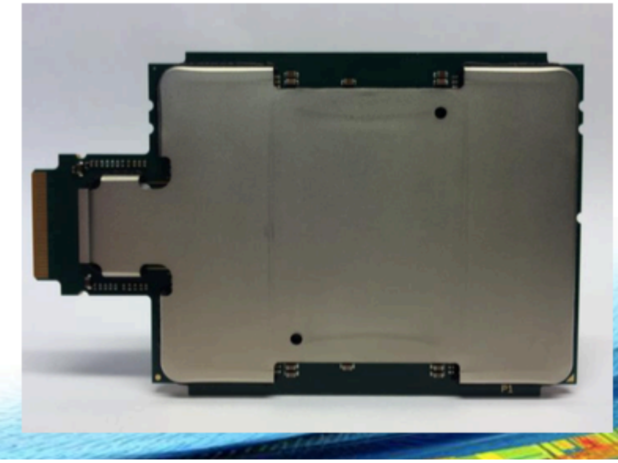
Output: 2 Omni-Path ports

- 25 GB/s/port (bi-dir)

Benefits

- Lower cost, latency and power
- Higher density and bandwidth
- Higher scalability





*On package connect with PCIe semantics, with MCP optimizations for physical layer

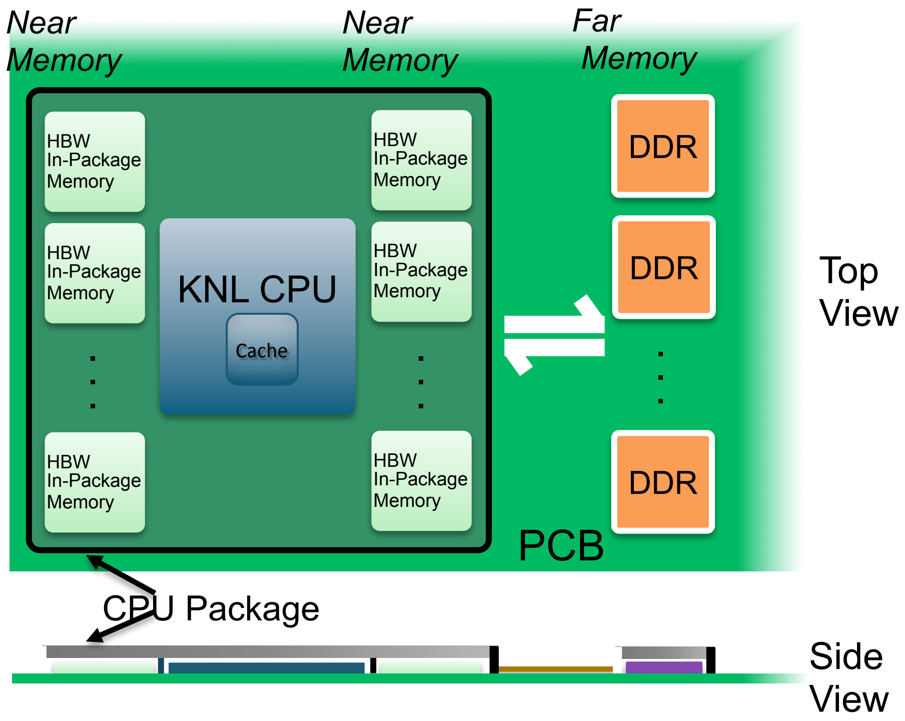
Source: Avinash Sodani, Hot Chips 2015 KNL talk

Knights Landing Integrated On-Package Memory

Cache Model Let the hardware automatically manage the integrated on-package memory as an “L3” cache between KNL CPU and external DDR

Flat Model Manually manage how your application uses the integrated on-package memory and external DDR for peak performance

Hybrid Model Harness the benefits of both cache and flat models by segmenting the integrated on-package memory

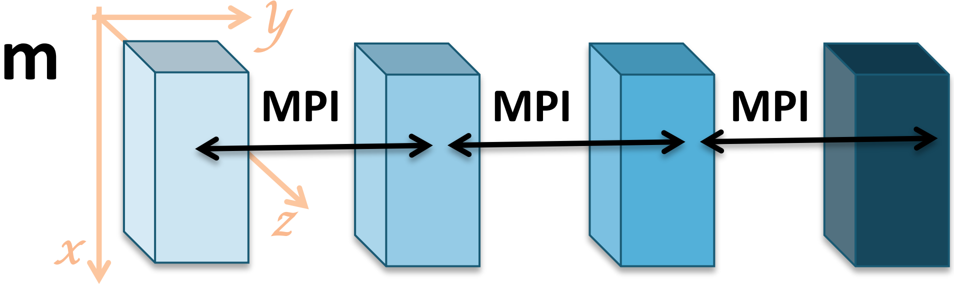


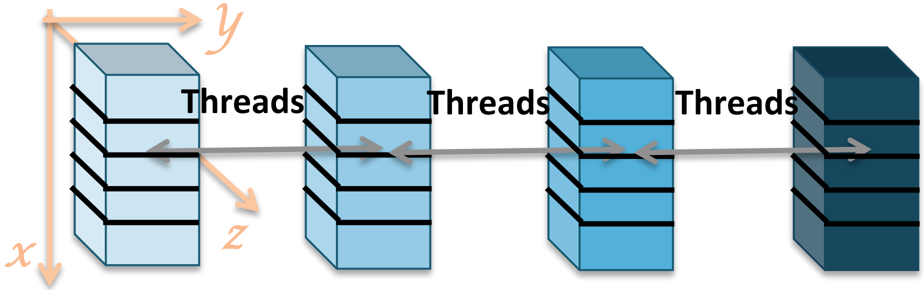
Maximum performance through higher memory bandwidth and flexibility

Slide from Intel

To run effectively on Cori users will have to:

- **Manage Domain Parallelism**
  - independent program units; explicit
- **Increase Node Parallelism**
  - independent execution units within the program; generally explicit
- **Exploit Data Parallelism**
  - Same operation on multiple elements
- **Improve data locality**
  - Cache blocking; Use on-package memory

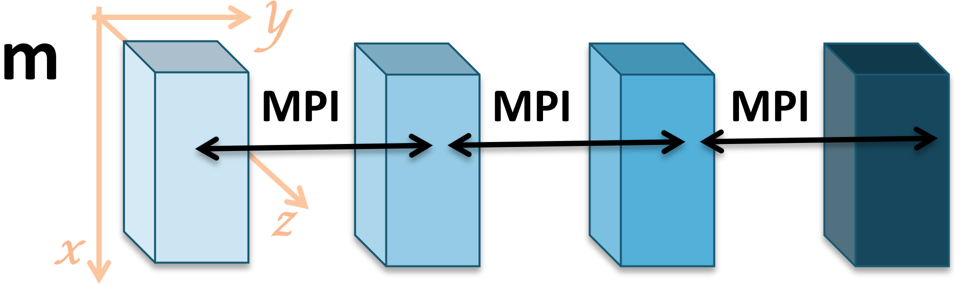


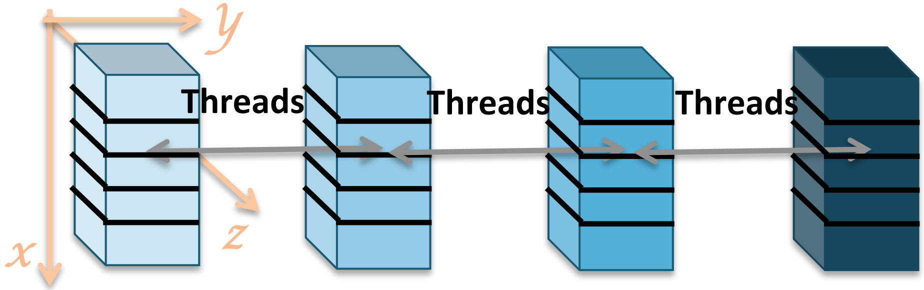


```
|--> DO I = 1, N
|        R(I) = B(I) + A(I)
|--> ENDDO
```

To run effectively on Cori users will have to:

- **Manage Domain Parallelism**
  - independent program units; explicit
- **Increase Node Parallelism**
  - independent execution units within the program; generally explicit
- **Exploit Data Parallelism**
  - Same operation on multiple elements
- **Improve data locality**
  - Cache blocking; Use on-package

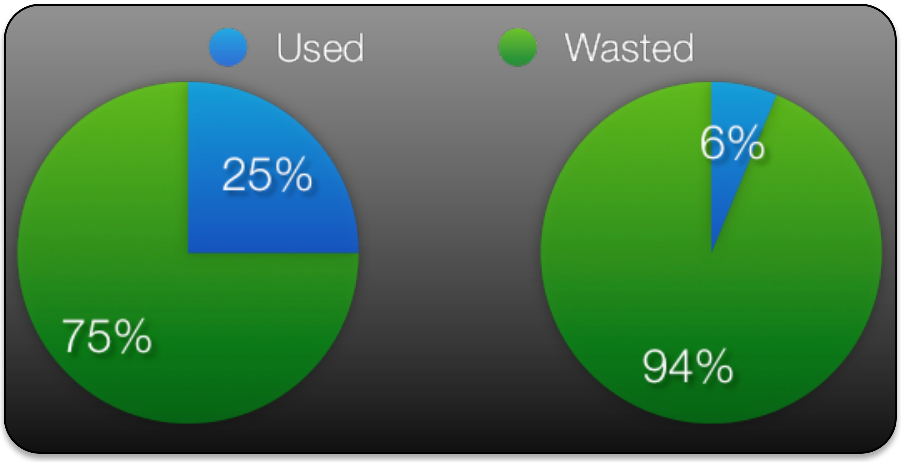




```
|--> DO I = 1, N
|       R(I) = B(I) + A(I)
```

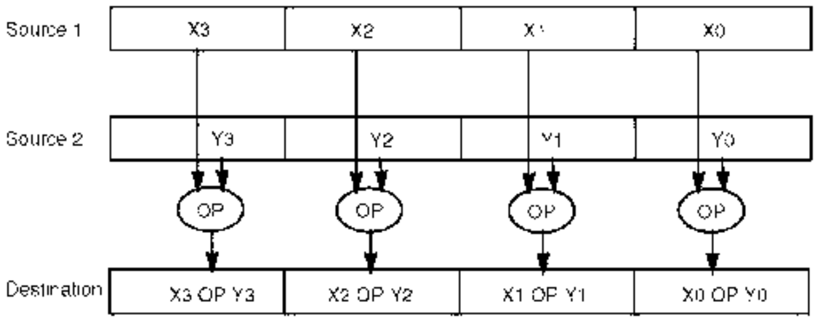
You mean vectorization. The only way you can be happy with KNL is if you can keep the pair of vector units per core busy.

Vector (SIMD) Programming



4 way SIMD (SSE)

16 way SIMD (Xeon™ PHI)

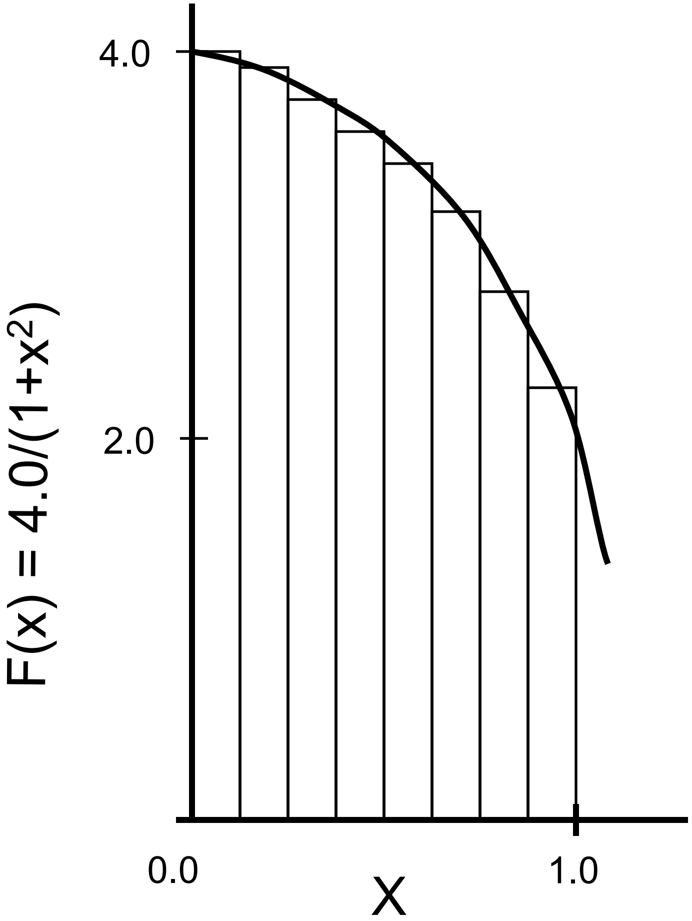


- Architects love vector units, since they permit space- and energy- efficient parallel implementations.
- However, standard SIMD instructions on CPUs are inflexible, and can be difficult to use.
- Options:
  - Let the compiler do the job
  - Assist the compiler with language level constructs for explicit vectoriztion.
  - Use intrinsics … an assembly level approach.

Slide Source: Kurt Keutzer UC Berkeley, CS194 lecture

Example Problem:

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} \, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial PI program

```
static long num_steps = 100000;
float step;
int main ()
{           int i;      float x, pi, sum = 0.0;

            step = 1.0/(float) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Normally, I'd use double types throughout to minimize roundoff errors especially on the accumulation into sum. But to maximize impact of vectorization for these exercise, we'll use float types.

Explicit Vectorization PI program

```
static long num_steps = 100000;
float step;
int main ()
{           int i;      float x, pi, sum = 0.0;

            step = 1.0/(float) num_steps;
            #pragma omp simd reduction(+:sum)
            for (i=0;i< num_steps; i++){
                  x = (i+0.5)*step;
                  sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

Note that literals (such as 4.0, 1.0 and 0.5) are not explicitly declared with the desired type. The C language treats these as "double" and that impacts compiler optimizations. We call this the default case.

Explicit Vectorization PI program

```
static long num_steps = 100000;
float step;
int main ()
{          int i;    float x, pi, sum = 0.0;

           step = 1.0f/(float) num_steps;
           #pragma omp simd reduction(+:sum)
           for (i=0;i< num_steps; i++){
                     x = (i+0.5f)*step;
                     sum = sum + 4.0f/(1.0f+x*x);
           }
           pi = step * sum;
}
```

Literals as double (no-vec), 0.012 secs
Literals as Float (no-vec), 0.0042 secs

Note that literals (such as 4.0, 1.0 and 0.5) are explicitly declared as type float (to match the types of the variables in this code. This greatly enhances vectorization and compiler optimization.

Pi Program: Vectorization with intriniscs (SSE)

```
 float pi_sse(int  num_steps)
{  float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
   step = 1.0/(float) num_steps;

   __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
   __m128 one    = _mm_load1_ps(&scalar_one);
   __m128 four   = _mm_load1_ps(&scalar_four);
   __m128 vstep  = _mm_load1_ps(&step);
   __m128 sum    = _mm_load1_ps(&scalar_zero);
   __m128 xvec;   __m128 denom;  __m128 eye;

  for (int i=0;i< num_steps; i=i+4){          // unroll loop 4 times
      ival      = (float)i;                   // and assume num_steps%4 = 0
      eye       = _mm_load1_ps(&ival);
      xvec      = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
      denom  = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
      sum      = _mm_add_ps(_mm_div_ps(four,denom),sum);
   }
   _mm_store_ps(&vsum[0],sum);
   pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
}
```

Pi Program: Vector intriniscs plus OpenMP

```
float pi_sse(int  num_steps)
{  float scalar_one =1.0, scalar_zero = 0.0,  ival, scalar_four =4.0, step, pi, vsum[4];
  float local_sum[NTHREADS];   // set NTHREADS elsewhere, often to num of cores
  step = 1.0/(float) num_steps;  pi = 0.0;
 #pragma omp parallel
 {    int i, ID=omp_get_thread_num();
      __m128 ramp   = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
      __m128 one    = _mm_load1_ps(&scalar_one);
      __m128 four   = _mm_load1_ps(&scalar_four);
      __m128 vstep  = _mm_load1_ps(&step);
      __m128 sum    = _mm_load1_ps(&scalar_zero);
      __m128 xvec;   __m128 denom;   __m128 eye;
     #pragma omp for
     for (int i=0;i< num_steps; i=i+4){
        ival      = (float)i;
        eye       = _mm_load1_ps(&ival);
        xvec      = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
        denom  = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
        sum      = _mm_add_ps(_mm_div_ps(four,denom),sum);
     }
    _mm_store_ps(&vsum[0],sum);
    local_sum[ID] = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
 }
  for(int k = 0; k<NUM_THREADS;k++) pi+=local_sum[k];
 return pi;
}
```

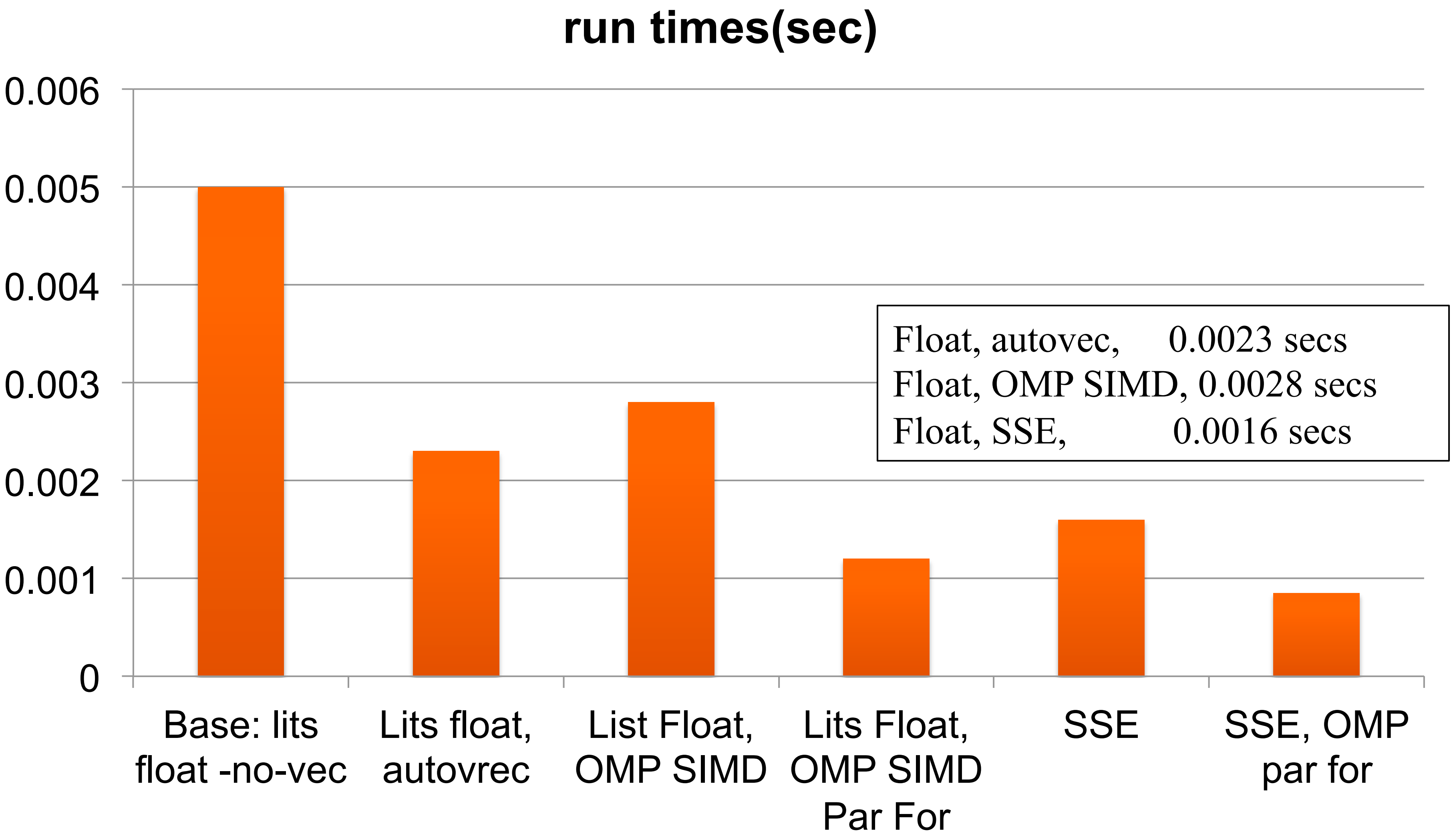
To parallelize with OpenMP:

1. Promote local_sum to an array to there is a variable private to each thread but available after the parallel region
2. Add parallel region and declare vector registers inside the parallel region so each thread has their own copy.
3. Add workshop loop (for) construct
4. Add local sums after the parallel region to create the final value for pi

PI program Results:

4194304 steps

Times in Seconds (50 runs, min time reported)

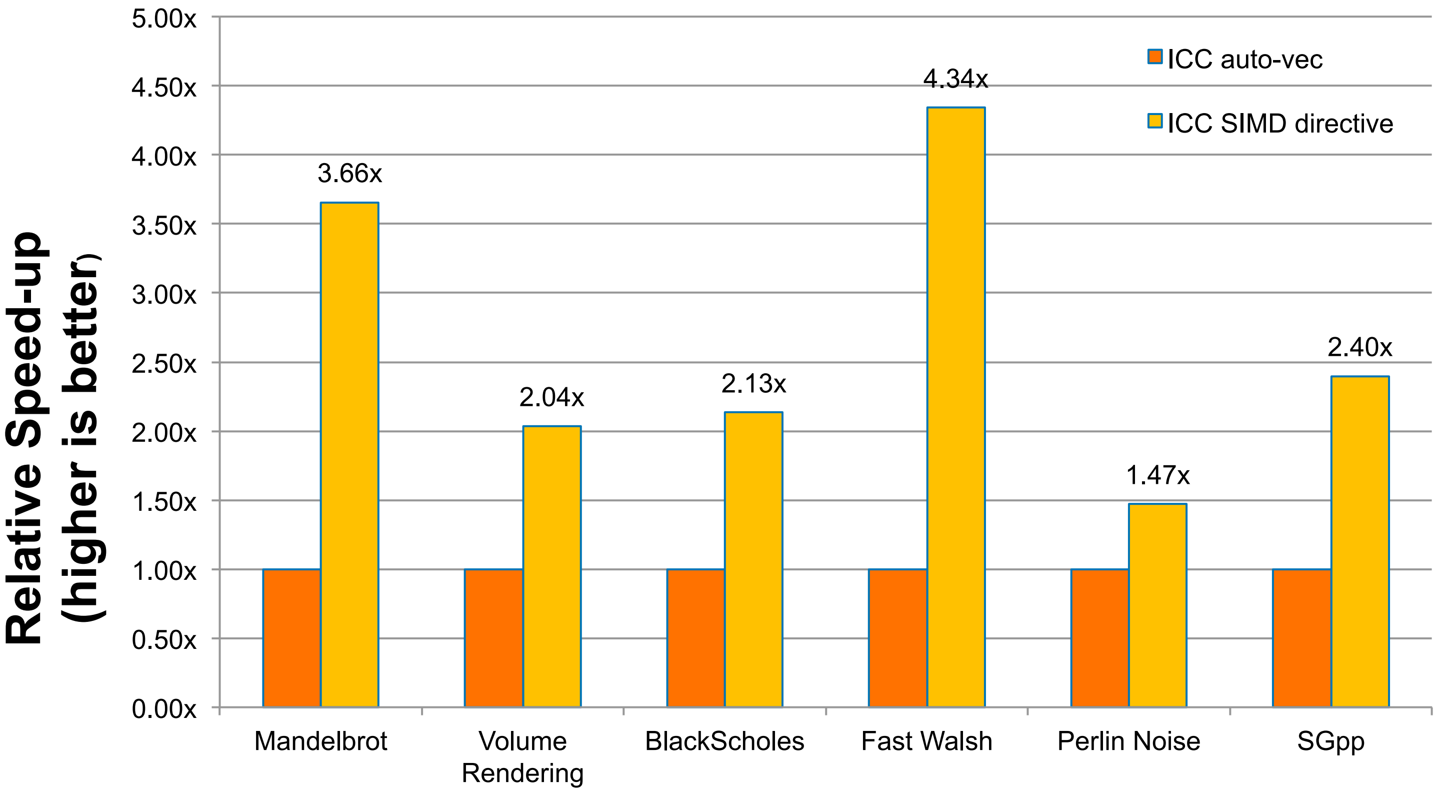


Float, autovec, 0.0023 secs
Float, OMP SIMD, 0.0028 secs
Float, SSE, 0.0016 secs

- Intel Core i7, 2.2 Ghz, 8 GM 1600 MHz DDR3, Apple MacBook Air OS X 10.10.5.
- Intel(R) C Intel(R) 64 Compiler XE for applications running on Intel(R) 64, Version 15.0.3.187 Build 20150408

Explicit Vectorization – Performance Impact

Explicit Vectorization looks better when you move to more complex problems.



Source: M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, "Extending OpenMP with Vector Constructs for Modern Multicore SIMD Architectures. In Proc. of the Intl. Workshop on OpenMP", pages 59-72, Rome, Italy, June 2012. LNCS 7312.

What about application portability?

- **Major US computer centers have and will continue to have fundamentally different architectures, for example:**
  - NERSC is based on KNL
  - OLCF and LLNL have announced an IBM+NVIDIA architecture
  - FUNDAMENTALLY DIFFERENT

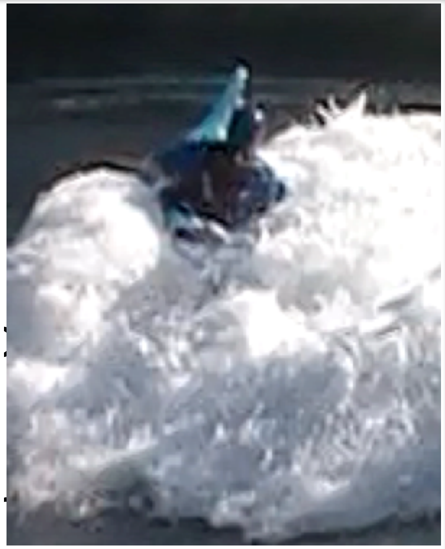
- **Will applications be able to run across both architectures?**
- **Several DOE workshops to address portability**
  - Best Practices Application portability workshop – Sept 2015

Application Programmers Dilemma

- **It actually only seemed hard before –**
  - First there were vectors, we coped
  - Then there was the MPP revolution so,
    - We ripped out all that vector code in favor of message passing
    - We finally came up with a standard that most could live with –MPI
  - For the brave of heart you could try MPI + OpenMP, but it really didn't do much
  - OpenMP worked well on smaller numbers of processors (cores) in shared memory

Application Programmers Dilemma

Scaling is typically a function of the algorithm and how you use an API, not the API itself. I haven't seen the codes my good friends from NERSC are talking about when making this statement, but in my experience, HPC codes often poorly use OpenMP. They just litter their codes with "parallel for"; not thinking about restructuring code to optimize data access patterns (NUMA issues) and reduce thread management overhead



with –MPI

- For the brave of heart you could try MPI + OpenMP, but it really didn't do much
- OpenMP worked well on smaller numbers of processors (cores) in shared memory

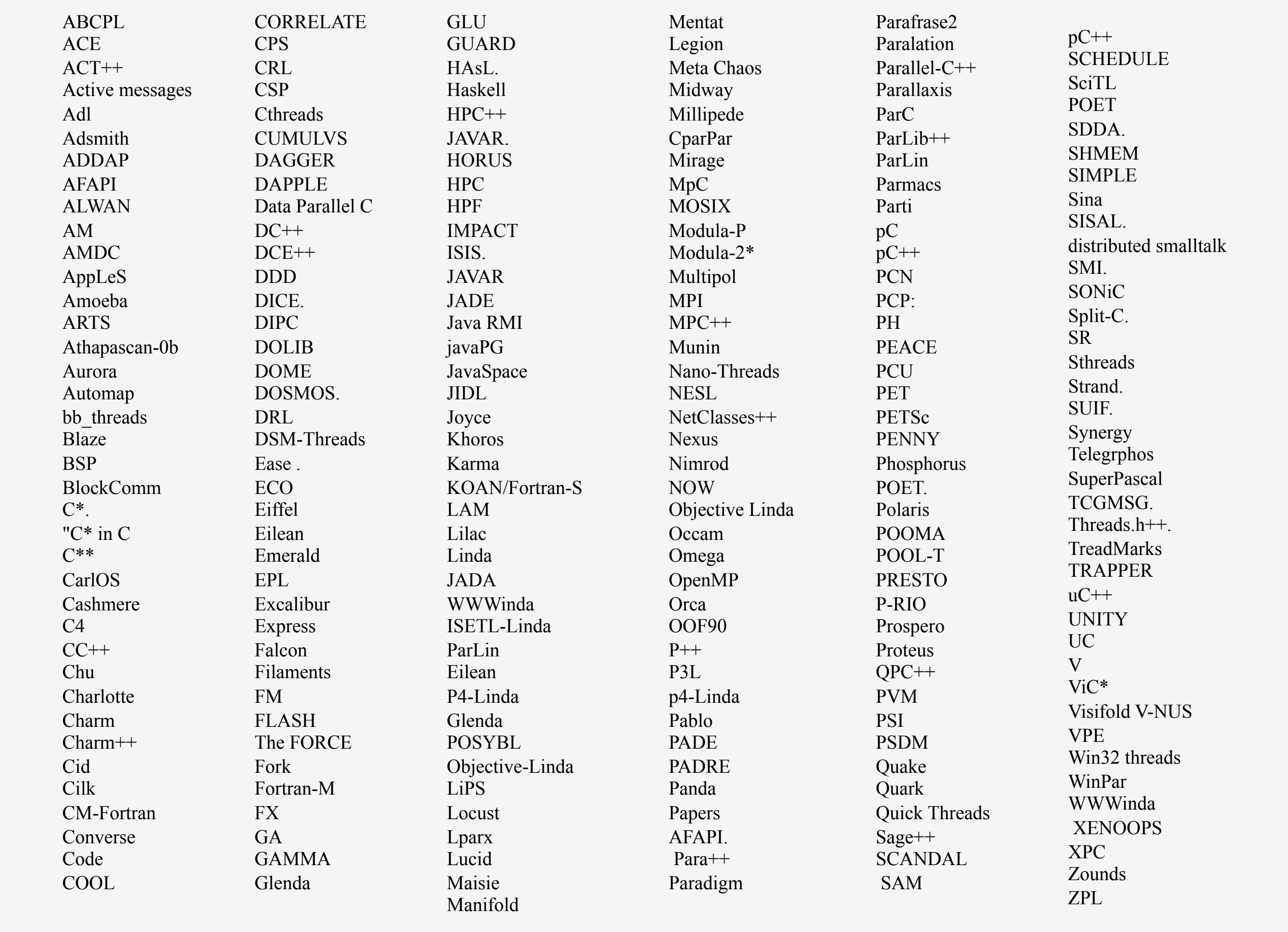
Programming Models by the Dozen, what to do now

Emperor Joseph II: My dear young man, don't take it too hard. Your work is ingenious. It's quality work. And there are simply too many notes, that's all. Just cut a few and it will be perfect.

Mozart: Which few did you have in mind, Majesty?

We tried to solve the programmability problem by searching for the right programming environment

Parallel programming environments in the 90's



ABCPL
ACE
ACT++
Active messages
Adl
Adsmith
ADDAP
AFAPI
ALWAN
AM
AMDC
AppLeS
Amoeba
ARTS
Athapascan-0b
Aurora
Automap
bb_threads
Blaze
BSP
BlockComm
C*.
"C* in C
C**
CarlOS
Cashmere
C4
CC++
Chu
Charlotte
Charm
Charm++
Cid
Cilk
CM-Fortran
Converse
Code
COOL

CORRELATE
CPS
CRL
CSP
Cthreads
CUMULVS
DAGGER
DAPPLE
Data Parallel C
DC++
DCE++
DDD
DICE.
DIPC
DOLIB
DOME
DOSMOS.
DRL
DSM-Threads
Ease .
ECO
Eiffel
Eilean
Emerald
EPL
Excalibur
Express
Falcon
Filaments
FM
FLASH
The FORCE
Fork
Fortran-M
FX
GA
GAMMA
Glenda

GLU
GUARD
HAsL.
Haskell
HPC++
JAVAR.
HORUS
HPC
HPF
IMPACT
ISIS.
JAVAR
JADE
Java RMI
javaPG
JavaSpace
JIDL
Joyce
Khoros
Karma
KOAN/Fortran-S
LAM
Lilac
Linda
JADA
WWWinda
ISETL-Linda
ParLin
Eilean
P4-Linda
Glenda
POSYBL
Objective-Linda
LiPS
Locust
Lparx
Lucid
Maisie
Manifold

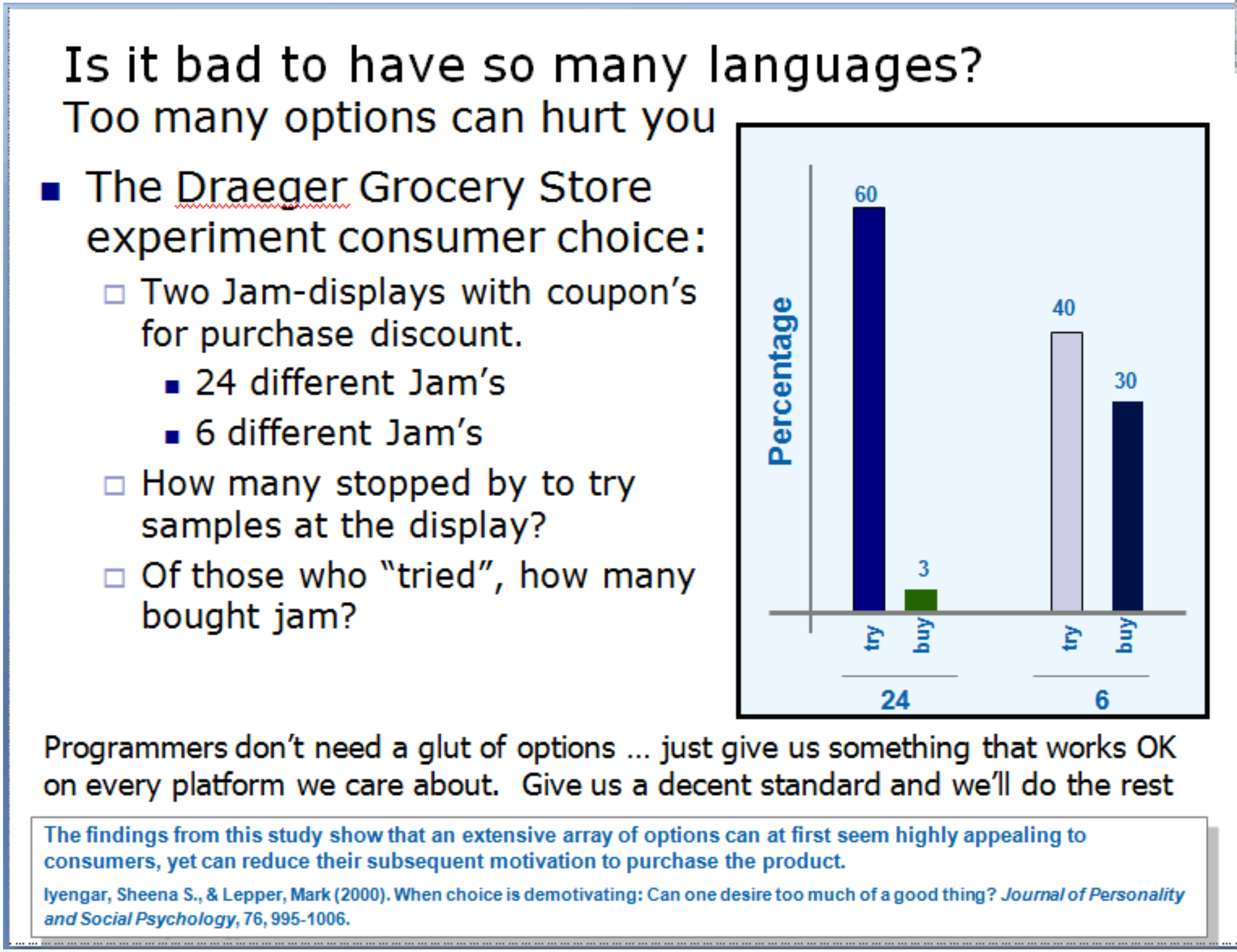
Mentat
Legion
Meta Chaos
Midway
Millipede
CparPar
Mirage
MpC
MOSIX
Modula-P
Modula-2*
Multipol
MPI
MPC++
Munin
Nano-Threads
NESL
NetClasses++
Nexus
Nimrod
NOW
Objective Linda
Occam
Omega
OpenMP
Orca
OOF90
P++
P3L
p4-Linda
Pablo
PADE
PADRE
Panda
Papers
AFAPI.
Para++
Paradigm

Parafrase2
Paralation
Parallel-C++
Parallaxis
ParC
ParLib++
ParLin
Parmacs
Parti
pC
pC++
PCN
PCP:
PH
PEACE
PCU
PET
PETSc
PENNY
Phosphorus
POET.
Polaris
POOMA
POOL-T
PRESTO
P-RIO
Prospero
Proteus
QPC++
PVM
PSI
PSDM
Quake
Quark
Quick Threads
Sage++
SCANDAL
SAM

pC++
SCHEDULE
SciTL
POET
SDDA.
SHMEM
SIMPLE
Sina
SISAL.
distributed smalltalk
SMI.
SONiC
Split-C.
SR
Sthreads
Strand.
SUIF.
Synergy
Telegrphos
SuperPascal
TCGMSG.
Threads.h++.
TreadMarks
TRAPPER
uC++
UNITY
UC
V
ViC*
Visifold V-NUS
VPE
Win32 threads
WinPar
WWWinda
XENOOPS
XPC
Zounds
ZPL

Third party names are the property of their owners.

A warning I've been making for the last 10 years



Is it bad to have so many languages?

Too many options can hurt you

- The Draeger Grocery Store experiment consumer choice:
  - □ Two Jam-displays with coupon's for purchase discount.
    - 24 different Jam's
    - 6 different Jam's
  - □ How many stopped by to try samples at the display?
  - □ Of those who "tried", how many bought jam?

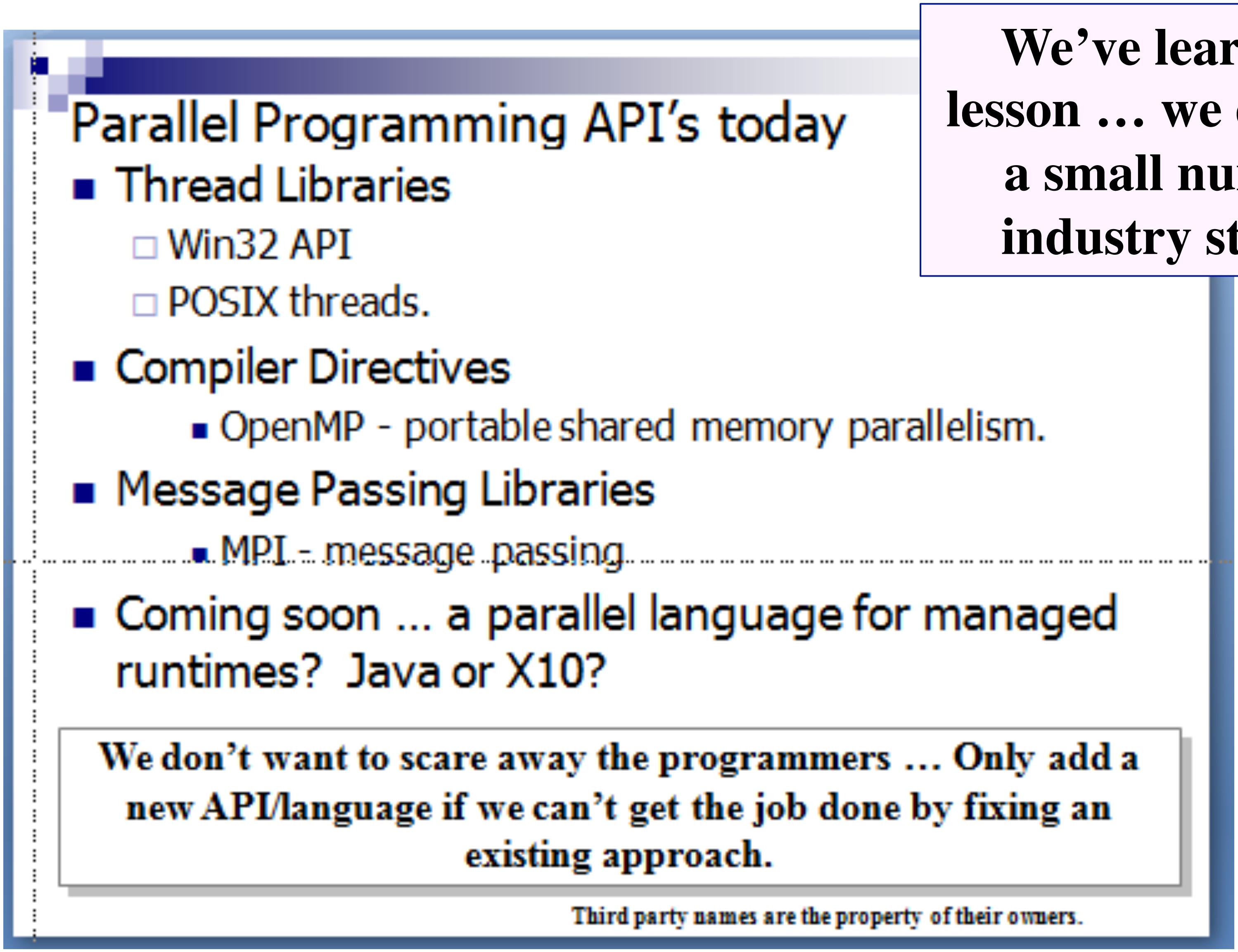
Programmers don't need a glut of options … just give us something that works OK on every platform we care about. Give us a decent standard and we'll do the rest

The findings from this study show that an extensive array of options can at first seem highly appealing to consumers, yet can reduce their subsequent motivation to purchase the product.

Iyengar, Sheena S., & Lepper, Mark (2000). When choice is demotivating: Can one desire too much of a good thing? *Journal of Personality and Social Psychology*, 76, 995-1006.

My optimistic view from 2005 …

We've learned our lesson … we emphasize a small number of industry standards



Parallel Programming API's today

- Thread Libraries
  - □ Win32 API
  - □ POSIX threads.
- Compiler Directives
  - OpenMP - portable shared memory parallelism.
- Message Passing Libraries
  - MPI - message passing
- Coming soon … a parallel language for managed runtimes? Java or X10?

We don't want to scare away the programmers … Only add a new API/language if we can't get the job done by fixing an existing approach.

Third party names are the property of their owners.

But we didn’t learn our lesson

History is repeating itself!

A small sampling of models from the NEW golden age of parallel programming (from the literature 2010-2012)

AM++
ArBB
BSP
C++11
C++AMP
Charm++
Chapel
Cilk++
CnC
coArray Fortran
Codelets
Copperhead
CUDA
DryadOpt
Erlang
Fortress
GA
GO
Gossamer
GPars
GRAMPS
Hadoop
HMMP
ISPC
Java
Liszt
MapReduce
MATE-CG
MCAPI
MPI
NESL
OoOJava
OpenMP
OpenCL
OpenSHMEM
OpenACC
PAMI
Parallel Haskell
ParalleX
PATUS
PLINQ
PPL
Pthreads
PXIF
PyPar
Plan42
RCCE
Scala
SIAL
STAPL
STM
SWARM
TBB
UPC
Win32 threads
X10
XMT
ZPL

We’ve slipped back into the “just create a new language” mentality.

Note: I’m not criticizing these technologies. I’m criticizing our collective urge to create so many of them.

What has gone wrong?

- In the old days (the 90's), the applications community were more aggressive with the vendors.
  - □ MPI was created and the applications community lined up behind it. Vendors responded so that within a year of the first MPI spec, quality implementation were everywhere
  - □ OpenMP was created and the applications community wrote it into RFPs and committed to it. Within a year of the first OpenMP spec, quality implementations were everywhere.
- Today?
  - □ Users are letting vendors lock them to a platform. What message are you giving to the vendor community when you use CUDA* or OpenACC*? If you won't commit to a vendor neutral, open standard, why should the vendors?

*Third party names are the property of their owners

An application programmers biggest fear

- **An application programmers biggest fear is that the language they toiled to learn will be the wrong choice**
  - Doesn't give performance
  - Too hard to figure out
  - No interoperability

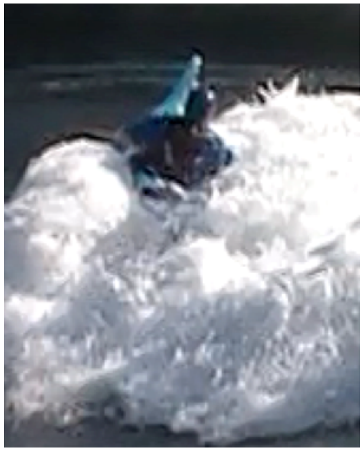
  - NOT THERE TWO YEARS LATER

Community input to open standards provides a path forward for portability

- **Portability is difficult, nothing about it makes parallel programming easier, except perhaps it encourages the programmer to hide parallelism**
- **People are generally in favor of using open standards and working towards good standards**
  - Examples: MPI Forum, OpenMP Architecture Review Board, etc.

Jeff Squyers (Cisco) at EuroMPI Sept. 2015:
..we will be “Defining what parallel computing will be for the world, this is the MPI forum. For everyone.”

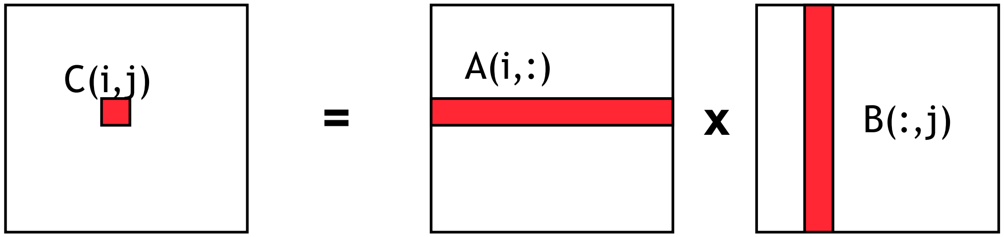
Whining about performance Portability



- Do we have performance portability today?
  - NO: Even in the “serial world” programs routinely deliver single digit efficiencies.
  - If the goal is a large fraction of peak performance, you will need to specialize code for the platform.
- But there is a pretty darn good performance portable language. It’s called OpenCL

Matrix multiplication example:
Naïve solution, one dot product per element of C

- Multiplication of two dense matrices.



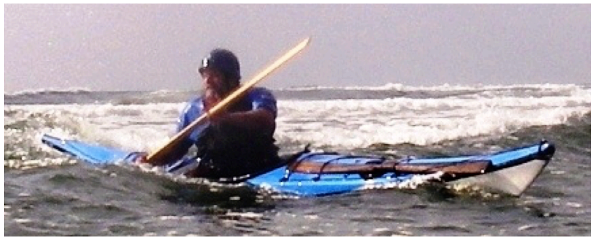
Dot product of a row of A and a column of B for each element of C

- To make this fast, you need to break the problem down into chunks that do lots of work for sub problems that fit in fast memory (OpenCL local memory).

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
      }
    }
}
```

Matrix multiplication: sequential code



```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Let's get rid of all those ugly brackets

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks “out” and leave loops over a single block together

Matrix multiplication: sequential code

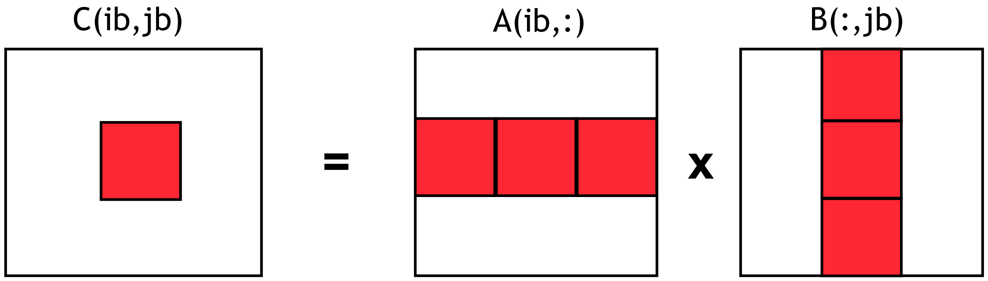
```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // $C_{ib,jb} = A_{ib,kb} * B_{kb,jb}$
}
```



Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
                const unsigned int N,
                __global float* A,
                __global float* B,
                __global float* C,
                __local  float* Awrk,
                __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;
```

```
  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
                const unsigned int N,
                __global float* A,
                __global float* B,
                __global float* C,
                __local  float* Awrk,
                __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;
```

Load A and B blocks, wait for all work-items to finish

```
  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;      int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Wait for everyone to finish before going to next iteration of Kblk loop.

Matrix multiplication … Portable Performance

- Single Precision matrix multiplication (order 1000 matrices)

Case	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

Could I do this with OpenMP today? No. But I look forward to trying once OpenMP is ready

Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB

* The comp was run twice and only the second time is reported (hides cost of memory movement.

Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, Open

Intel Core i7-4850HQ @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.

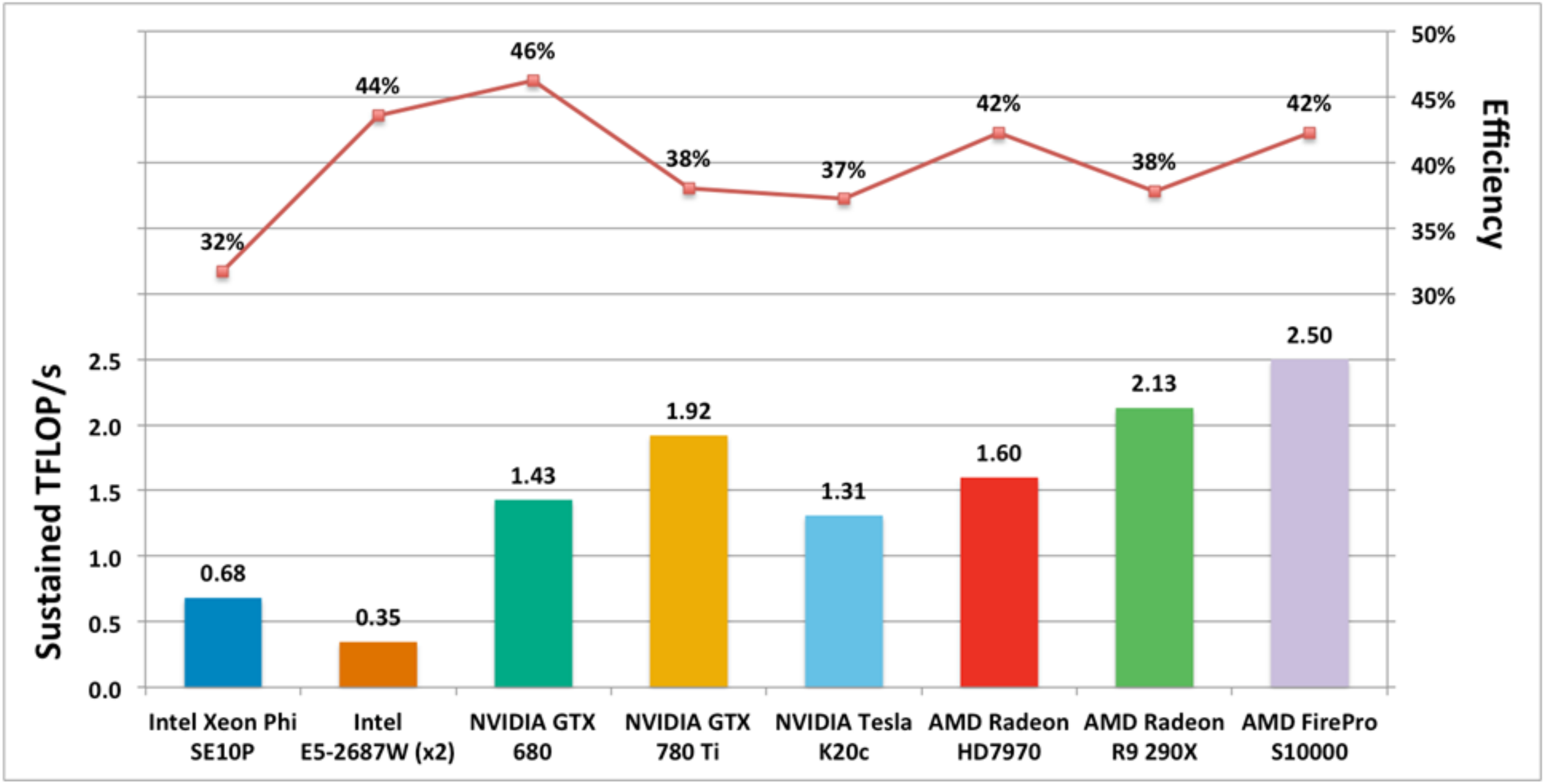
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

Third party names are the property of their owners.

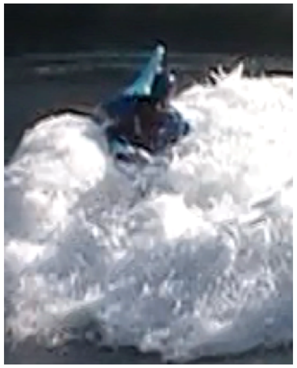
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

BUDE: Bristol University Docking Engine

One program running well on a wide range of platforms



Whining about performance Portability



- Do we have performance portability today?
  - NO: Even in the "serial world" programs routinely deliver single digit efficiencies.
  - If the goal is a large fraction of peak performance, you will need to specialize code for the platform.
- However there is a pretty darn good performance portable language. It's called OpenCL
- But this focus on mythical "Performance Portability" misses the point. The issue is "maintainability".
  - You must be able maintain a body of code that will live for many years over many different systems.
  - Having a common code base using a portable programming environment … even if you must fill the code with if-defs or have architecture specific versions of key kernels … is the only way to support maintainability.

~35 Application White Papers submitted to recent DOE Workshop on Portability

- **Take-aways:**
  - Almost Everyone is prepared to try/use OpenMP4.0 and beyond to help with portability issues
  - Even with OpenMP accelerator directives, etc., two different source codes are necessary
  - Different source codes for two or more parallel programming constructs does encourage people to contain parallel code
    - This is not as easy to see in directive based approaches as with other approaches based more on libraries
  - Most people are resigned to having different sources for different platforms, with simple #ifdef or other mechanisms

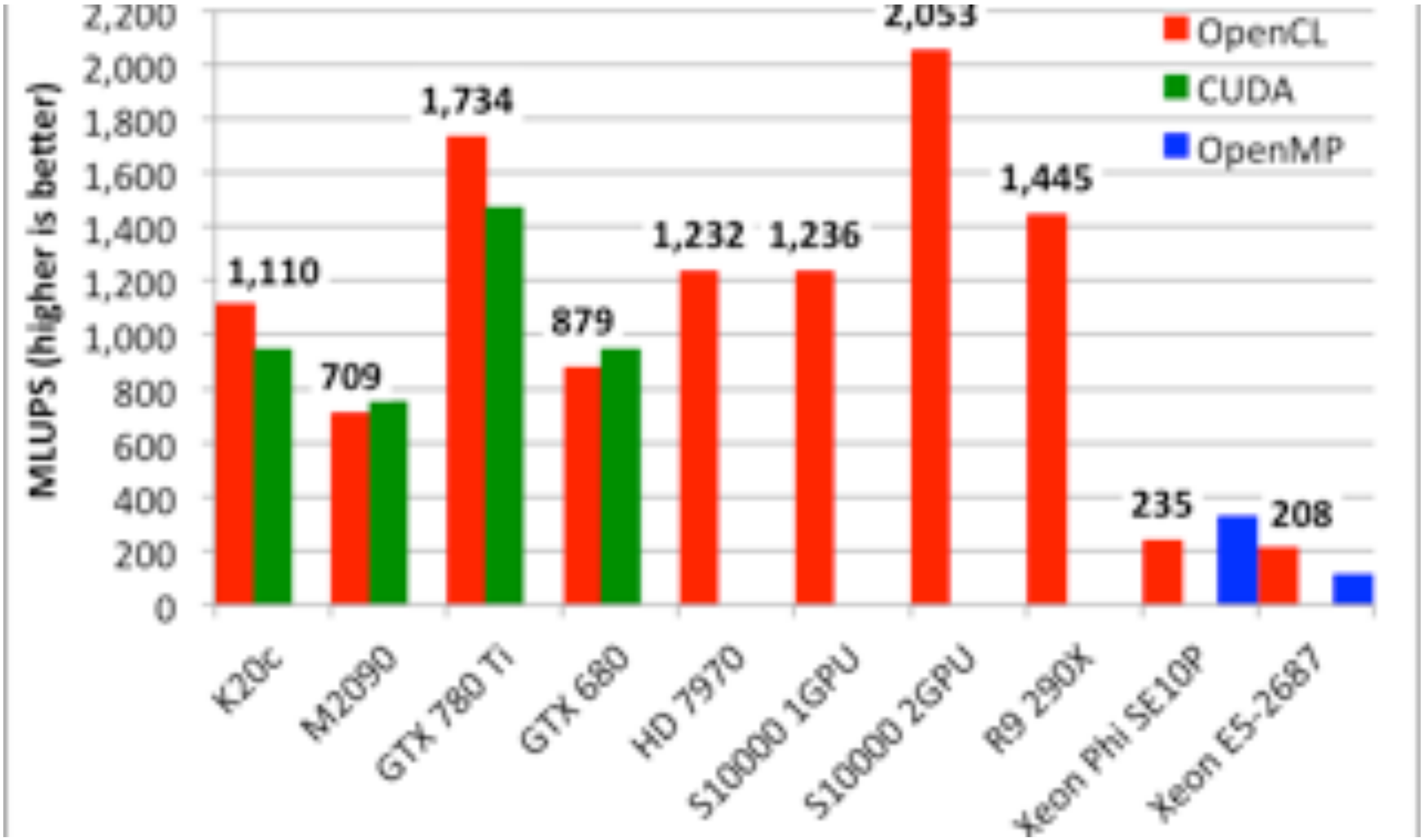
What is holding OpenMP back

- **Mature implementations are not everywhere**
- **Standard for accelerators is still being defined**
- **Performance is not there yet (see next two slides):**

On the performance portability of structured grid codes on many-core computer architectures", S.N. McIntosh-Smith, M. Boulton, D. Curran and J.R. Price. ISC, Leipzig, pp 53-75, June 2014.

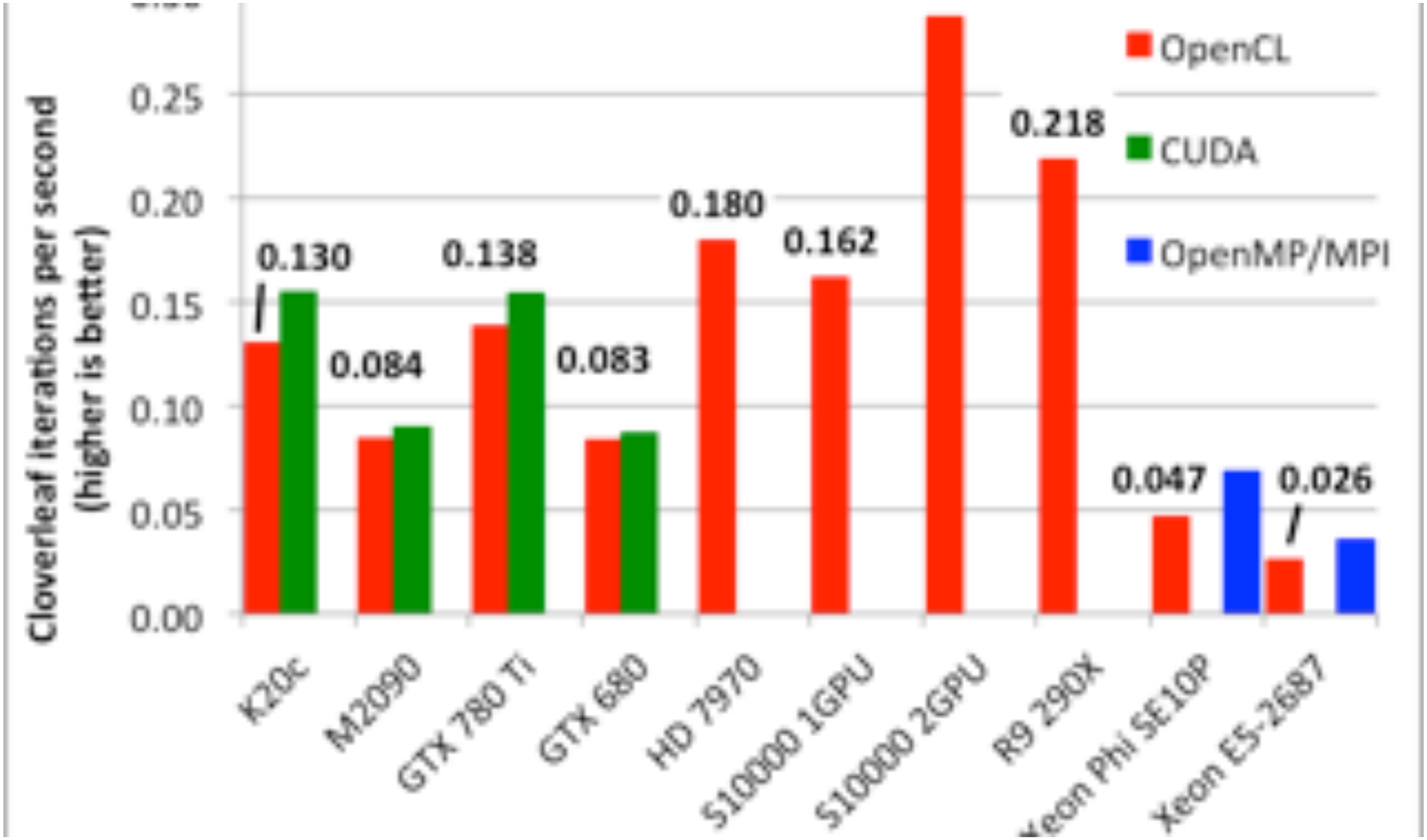
On the Performance Portability of structured Grid codes

… McIntosh-Smith et.al. ISC 2014



D3Q19-BGK Lattice Boltzman code

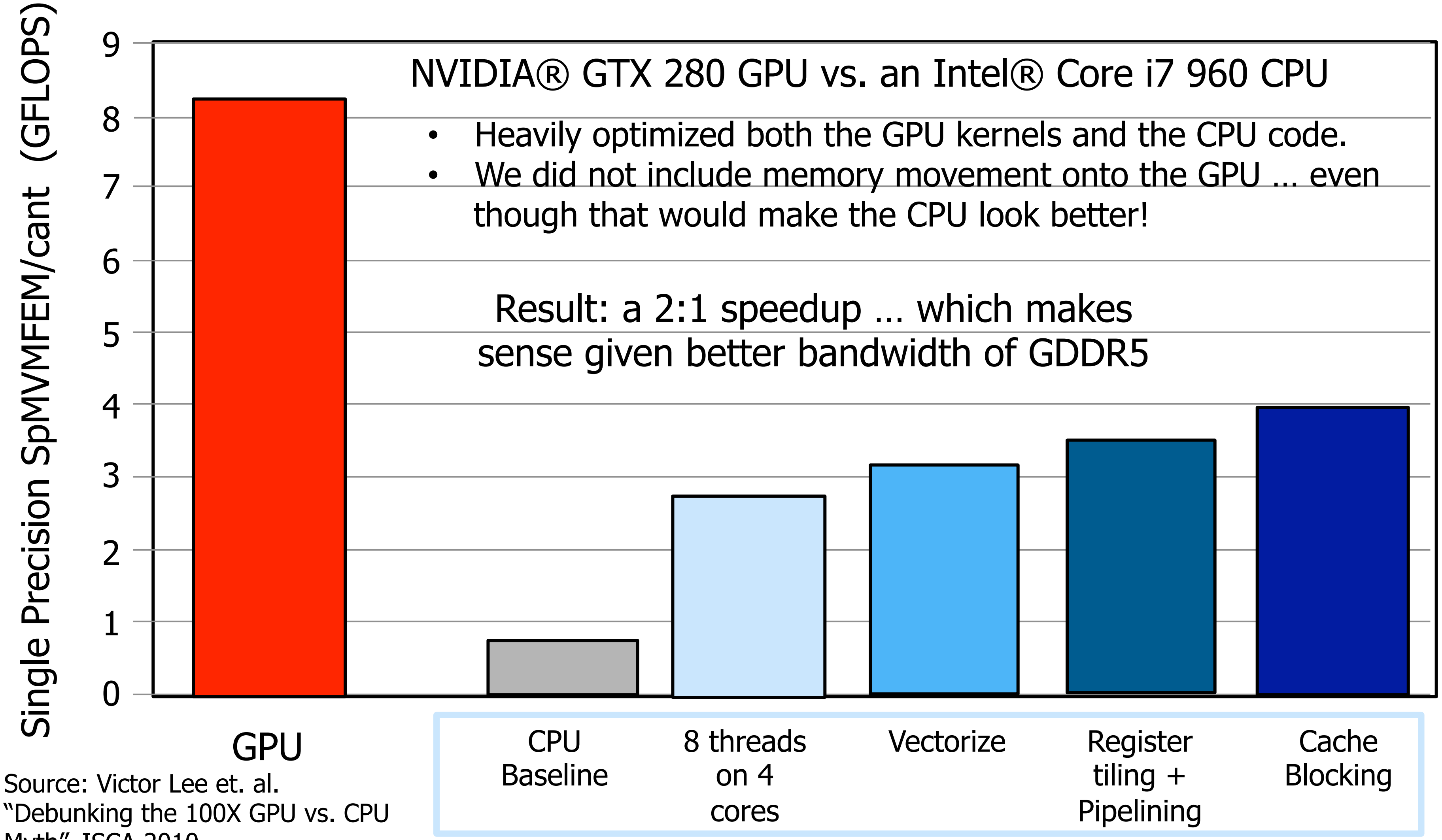
On the Performance Portability of structured Grid codes … McIntosh-Smith et.al. ISC 2014



Cloverleaf lagraingian-Eulerian hydrodynamics code

Sparse matrix vector product: GPU vs. CPU

- [Vazquez09]: reported a 51X speedup for an NVIDIA® GTX295 vs. a Core 2 Duo E8400 CPU … but they used an old CPU with unoptimized code



NVIDIA® GTX 280 GPU vs. an Intel® Core i7 960 CPU

- Heavily optimized both the GPU kernels and the CPU code.
- We did not include memory movement onto the GPU … even though that would make the CPU look better!

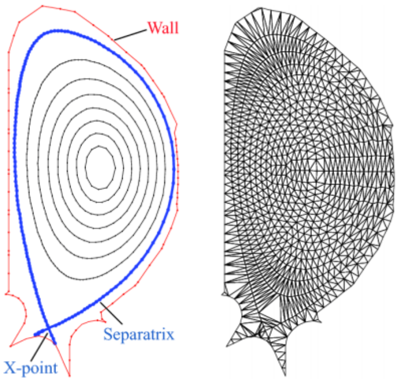
Result: a 2:1 speedup … which makes sense given better bandwidth of GDDR5

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

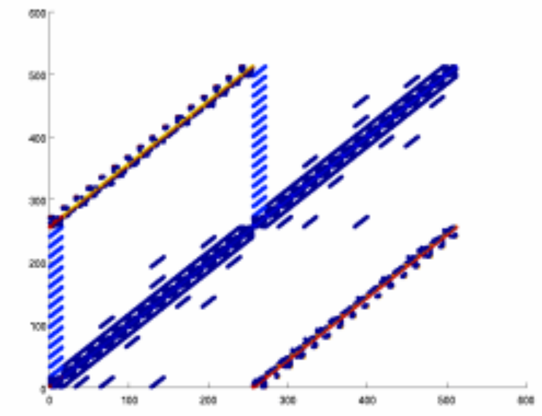
*third party names are the property of their owners

CASE STUDY: XGC1 PIC Fusion Code

- **Particle-in-cell code used to study turbulent transport in magnetic confinement fusion plasmas.**
- **Uses fixed unstructured grid. Hybrid MPI/OpenMP for both spatial grid and particle data. (plus PGI CUDA Fortran, OpenACC)**
- **Excellent overall MPI scalability**
- **Internal profiling timer borrowed from CESM**
- **Uses PETSc Poisson Solver** (separate NESAP effort)
- **60k+ lines of Fortran90 codes.**
- **For each time step:**
  - Deposit charges on grid
  - Solve elliptic equation to obtain electro-magnetic potential
  - Push particles to follow trajectories using forces computed from background potential (~50-70% of time)
  - Account for collision and boundary effects on velocity grid
- **Most time spent in Particle Push and Charge Deposition**



Unstructured triangular mesh grid due to complicated edge geometry



Sample Matrix of communication volume

Programming Portability

- Currently XGC1 runs on many platforms
- Part of NESAP and ORNL CAAR programs
- Applied for ANL Theta program
- Previously used PGI CUDA Fortran for accelerators
- Exploring OpenMP 4.0 target directives and OpenACC.
- Have #ifdef _OpenACC and #ifdef _OpenMP in code.
- Hope to have as fewer compiler dependent directives as possible.
- Nested OpenMP is used
- Needs thread safe PSPLIB and PETSc libraries.

CUDA Fortran code conversion (Jianying Lang, PPPL)

GPU kernel subroutine

Call host program in FORTRAN

```
#ifdef USE_GPU
     call pushe_gpu (istep,…,…)
#else
     call pushe (istep,…,…)
#endif
```

Launch GPU kernel in host program

```
 call
pushe_kernel_gpu<<<blocks,t
hreads>>>(istep,epc,phase0_g
pu,diag_on,dt_now)
```

```
attributes(global) &
subroutine pushe_kernel_gpu(istep,ipc,phase0, &
                diag_on,dt_now)
               .
               .
               .
ith = 1+ ((threadIdx%x-1) + (threadIdx%y-1)*blockDim%x) +  &
     ((blockIdx%x-1) + (blockIdx%y-1)*gridDim%x )* &
     (blockDim%x * blockDim%y)
 do i=ith-1, sp_num_gpu, nthreads_dim
       if(ptl_gid_gpu(i)>0) then
         x=ptl_ph_gpu(i,1:2)
         phi=ptl_ph_gpu(i,3)
         phi_mid=(floor(phi/grid_delta_phi) + 0.5_work_p) *   &
         grid_delta_phi
         call field_following_pos2_gpu(x,phi,phi_mid,xff)
         call search_tr2_gpu(xff,itr,p)
               .
               .
```

Current Implementation XGC1 code(example)

```
#ifdef _OPENACC
!$acc kernels present(Ms,EDs) ASYNC(istream)
!$acc loop independent  collapse(2) gang
#else
!$OMP  PARALLEL DO default(none) &
!$OMP& shared(mesh_Nzm1,mesh_Nrm1,f_half,dfdr,dfdz,Ms) &
!$OMP& shared(cs1,cs2,EDs,mass1,mass2) &
!$OMP& PRIVATE( index_I,index_J, index_2D, index_ip, index_jp, index_2dp, &
!$OMP& shared(cs1_mesh_r_half,cs1_mesh_z_half) &
!$OMP& shared(cs2_mesh_r_half,cs2_mesh_z_half) &
!$OMP& num_threads(col_f_nthreads)
#endif
       do index_I=1, mesh_Nzm1
       do index_J=1, mesh_Nrm1
           z = cs1_mesh_z_half(index_I)
           . . . . . . . . . .
!$acc      loop independent collapse(2) vector
       do index_ip = 1, mesh_Nzm1
       do index_jp = 1, mesh_Nrm1
           c = cs2_mesh_z_half(index_ip)
           . . . . . . . . . .
#ifdef _OPENACC
!$acc end kernels
#endif
```

- Use **preprocessor statement** to switch between OpenMP and OpenACC

- Vectorization is critical for both Cori and Summit

Some Recommendations from Portability Workshop Especially w.r.t. Library Portability

- **Common base software environment across HPC Centers**
  - Base HPC software stack (standard base set of libs, tools)
  - Share software build, installation, management, testing procedures/mechanisms for HPC centers (e.g. spack)
  - SW development utilities for users
  - Common build recipes, methods at HPC centers
- **Performance portability: encourage investment, adoption, & guidance**
  - Back-end code generation
  - Compiler-based approaches: LLVM/JIT, Rose
  - Open Standards for Parallel Computing
  - C++11/14/17
- **DOE investment in standards committees**
- **Library developers can define strict interface, then ask vendors to confirm to them**
- **Extensions to MPI to exploit fine-grained parallelism (intra-node)**
- **Ability to transform individual research projects or libraries into production capabilities**

No One-Size Fits all solutions

- **With MPI we made it work, eventually**
- **Didn't matter which of the characteristics your application had –**
  - Particles – divide among processors
  - Grid – hand-off sections
  - Matrix –divide off rows and columns
- **We may come to the conclusions that no one heterogeneous architecture nor one single parallel programming model will work for all applications**

Portable parallel programming is in bad shape. Who to blame?

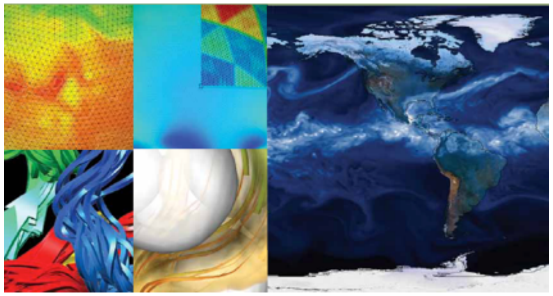
- Application programmers …
  This mess is your fault!
- We live in a market economy. Your interests (consistent and stable environments across platforms from multiple vendors) are not the same as the vendor's interests.
- When you reward vendors for bad behavior (e.g. pushing their own standards), you get what you deserve.

- History has shown you the solution!
  - <u>**Unite and fight back. Revolt and force the change you need!!!!**</u>
  - Isolated, you lack power. Together you can shape the industry.
  - Just look at the creation of MPI and OpenMP and OpenCL.
  - Be firm in your resolve:
    - ONLY USE vendor neutral, open standards (e.g. OpenMP, OpenCL, MPI)
    - Standards take commitment and hard work. Join us in that work.

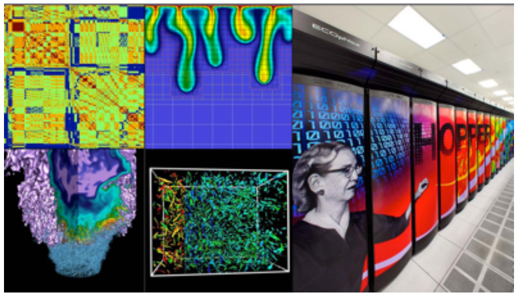
NERSC is the Mission HPC Facility for DOE Office of Science Research

U.S. DEPARTMENT OF ENERGY | Office of Science

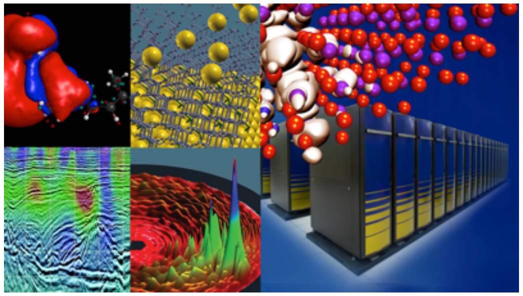
Largest funder of physical science research in U.S.



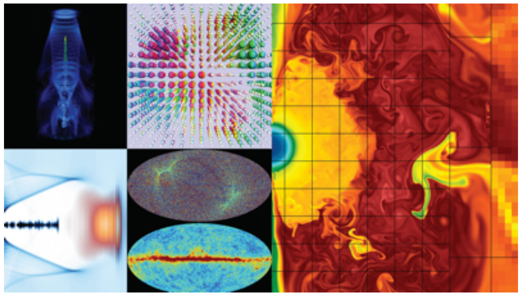
Bio Energy, Environment



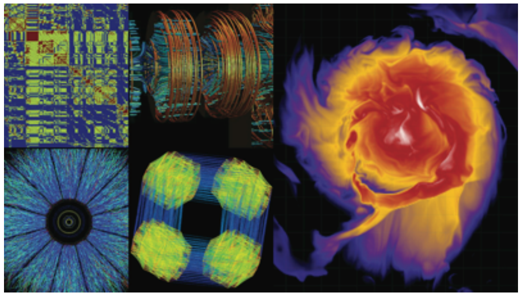
Computing



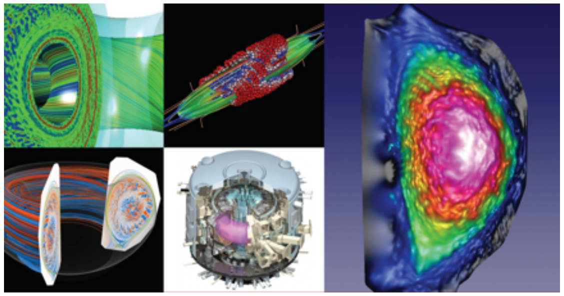
Materials, Chemistry, Geophysics



Particle Physics, Astrophysics



Nuclear Physics



Fusion Energy, Plasma Physics