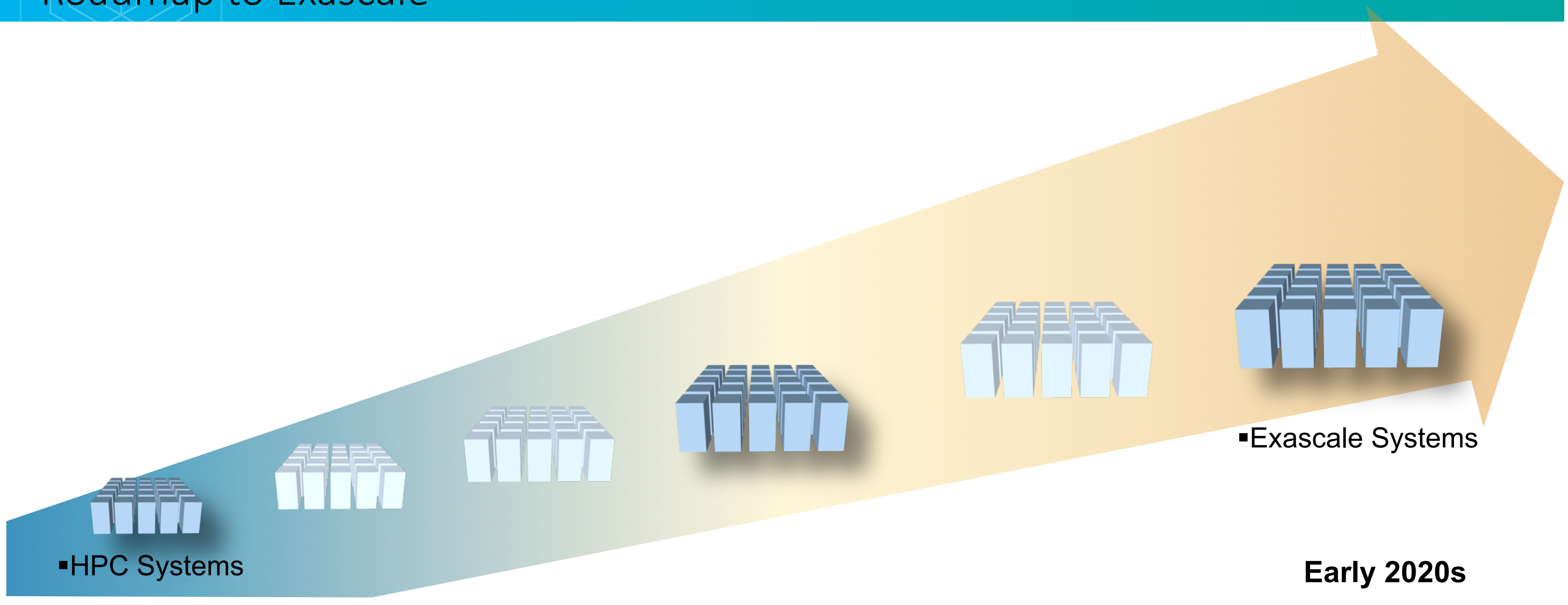
OpenMP in the Exascale Era

Kathryn O’Brien IBM Research

Roadmap to Exascale



Early 2010s

▪7X Performance

Early 2020s

50X Performance

Past:

Focus on HPC performance

Today:

Focus on Data, Analytics, Cognitive, and HPC

Heterogeneous compute

Future:

Add improved cognitive capabilities,
Heterogeneous compute and memory
Integration of new technologies.

The Programming Model Challenge

Architectual complexity impacts many aspects of the system software stack and execution environment:

OS and runtime implications
Control Systems, Resource managers, schedulers
Tools – performance monitors, visualization
Debuggers,
. . .

Architectural complexity impact is expected to increase significantly by 2022

Heterogeneity in compute and memory
Memory attributes
Data management
Resiliency
. . .

The programming model is the interface between all that complexity and the application developer . . .

Programming Model Constraints

- Address a broad range of programmer expertise
  - Low level programming expertise – **performance** is paramount
  - Application expertise – rapid deployment, **productivity and portability** are key

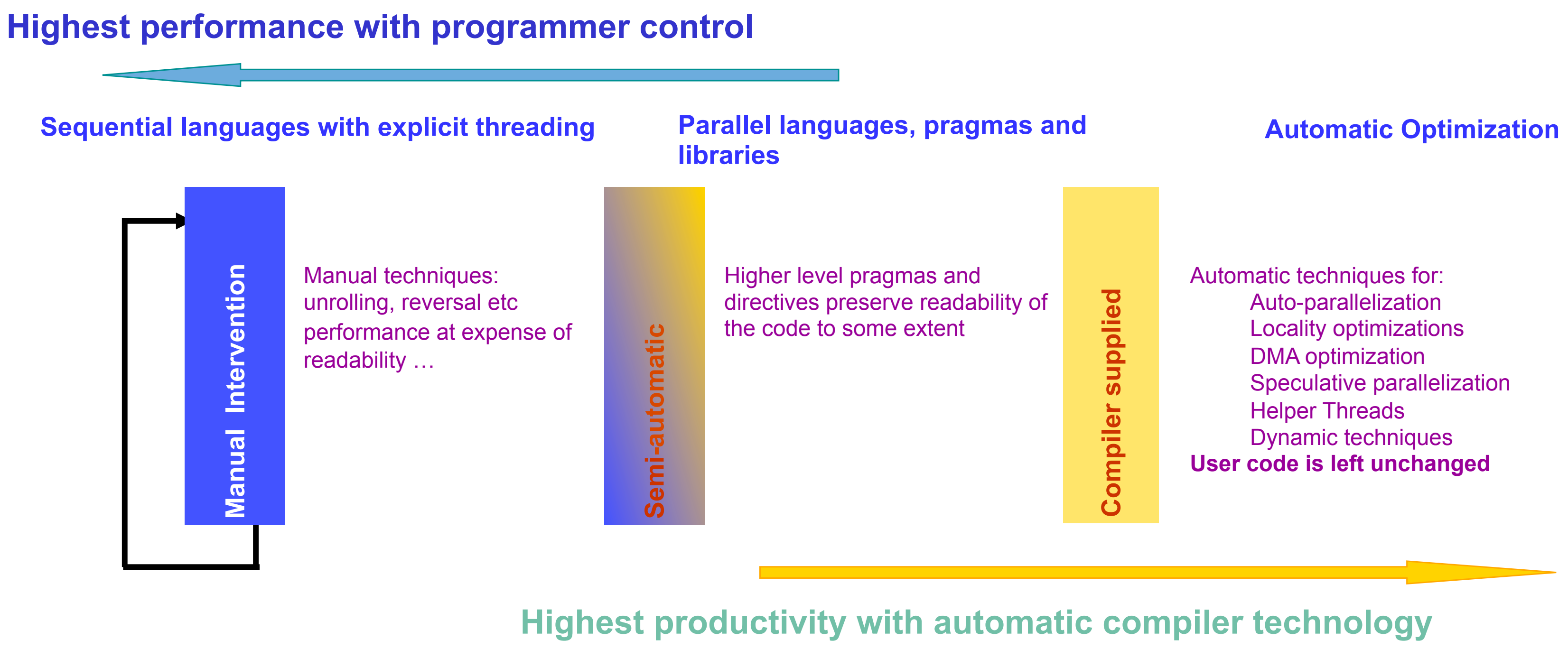
- Provide high level abstractions with ‘breakout’ mechanisms for critical performance paths

- Provide a migration path for Legacy codes and cross-platform performance portability

- Support a range of implementation paths
  - Libraries
  - Language/compiler extensions – pragmas, directives
  - New languages

- Barriers to adoption

The Application Programmers Dilemma



Highest performance with programmer control

Sequential languages with explicit threading

Manual techniques: unrolling, reversal etc performance at expense of readability …

Parallel languages, pragmas and libraries

Higher level pragmas and directives preserve readability of the code to some extent

Automatic Optimization

Automatic techniques for:
- Auto-parallelization
- Locality optimizations
- DMA optimization
- Speculative parallelization
- Helper Threads
- Dynamic techniques

User code is left unchanged

Highest productivity with automatic compiler technology

Exascale Programming Models: … what we were thinking ~ 2009/10

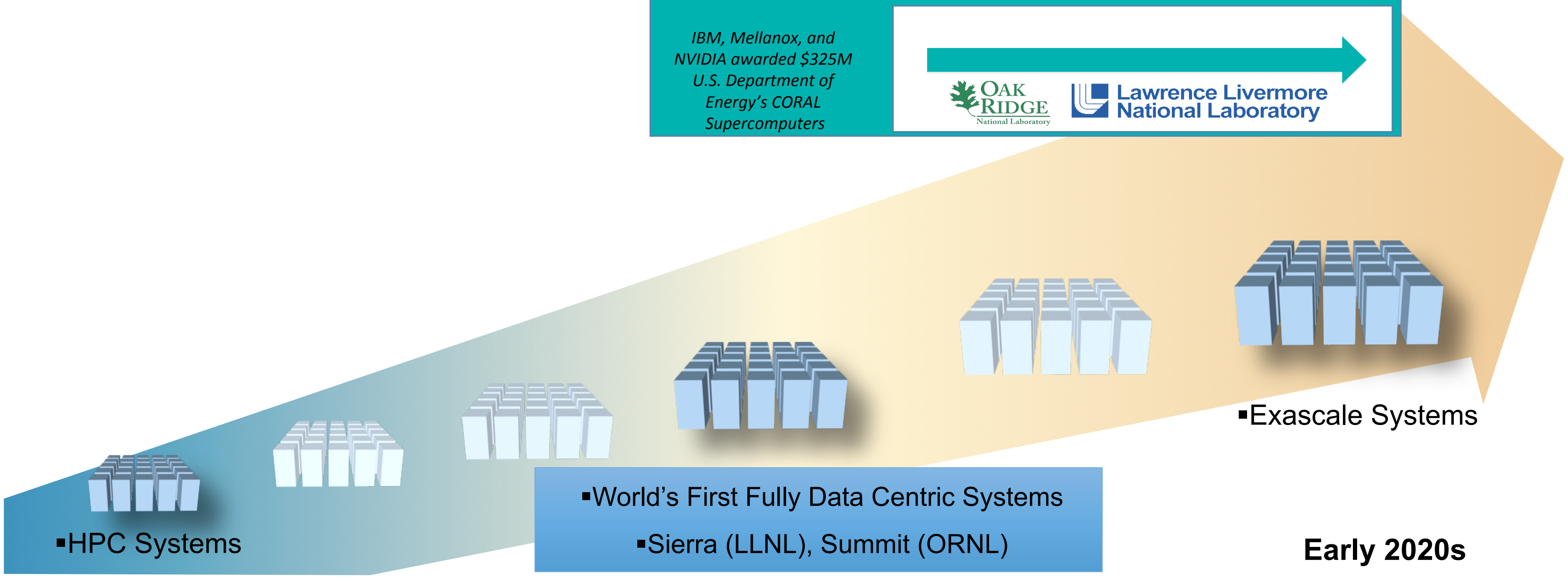
- For 2015/18 – new models not feasible in the timeframe
  - Abstract machine model is changing - Major focus needs to be on intra-node
    - HPC programming models have tended to follow rather than lead in the area of GPU technology
  - Inter-node – MPI is likely to be good enough
    - Unified models another option – but need hardware support for global address space
  - Interaction of programming model and RAS will be very important
  - More focus on asynchronous design
    - will enable applications to be more resilient, latency tolerant and more resistant to impact of jitter in large systems

- Invest in a range of programming models
  - Monitor evolving models beyond exascale community: CUDA, OpenCL, TBB …
  - Evolve established hybrid : MPI + OpenMP, Pthreads,
  - Develop new hybrid: MPI + PGAS ??
  - Holistic models: CAF, UPC, HPCS,
  - Revolutionary approaches - new languages not a good idea - unlikely that revolution will happen …

- Consensus to pursue three technologies:
  - Well defined abstract machine model and open runtime layer
  - Multiple diverse node level models with MPI internode
  - Tools

Roadmap to Exascale: co-design through collaborations

IBM, Mellanox, and NVIDIA awarded $325M U.S. Department of Energy's CORAL Supercomputers



Oak Ridge National Laboratory
Lawrence Livermore National Laboratory
▪HPC Systems
▪World's First Fully Data Centric Systems
▪Sierra (LLNL), Summit (ORNL)
▪Exascale Systems

Early 2010s

▪7X Performance

Early 2020s

50X Performance

Past:

Focus on HPC performance

Today:

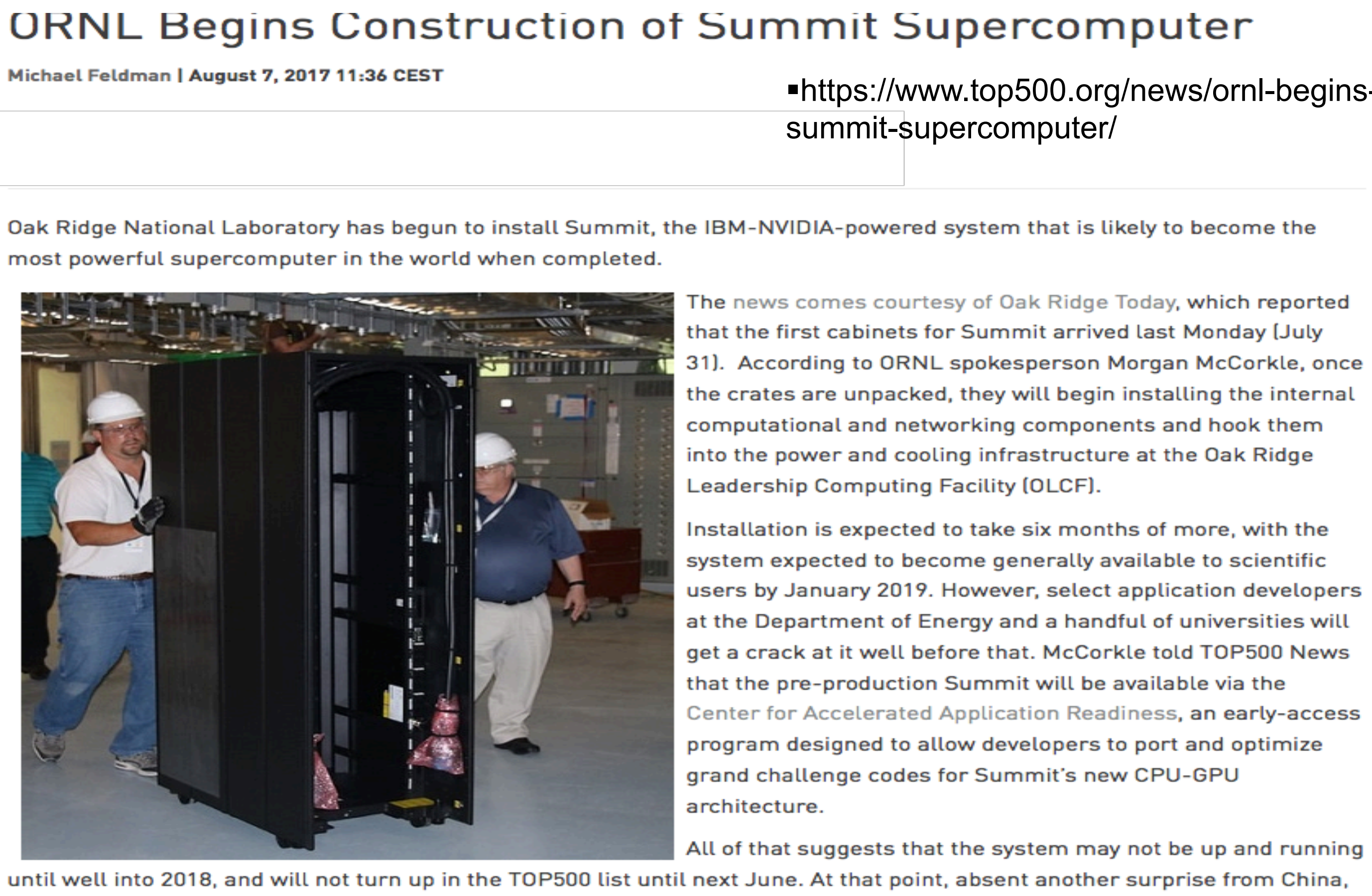
Focus on Data, Analytics, Cognitive, and HPC

Heterogeneous compute

Future:

Add improved cognitive capabilities,
Heterogeneous compute and memory
Integration of new technologies.

Summit and Sierra: The first pre-exascale systems . . .



ORNL Begins Construction of Summit Supercomputer

Michael Feldman | August 7, 2017 11:36 CEST

▪https://www.top500.org/news/ornl-begins-construction-of-summit-supercomputer/

Oak Ridge National Laboratory has begun to install Summit, the IBM-NVIDIA-powered system that is likely to become the most powerful supercomputer in the world when completed.

The news comes courtesy of Oak Ridge Today, which reported that the first cabinets for Summit arrived last Monday (July 31). According to ORNL spokesperson Morgan McCorkle, once the crates are unpacked, they will begin installing the internal computational and networking components and hook them into the power and cooling infrastructure at the Oak Ridge Leadership Computing Facility (OLCF).

Installation is expected to take six months of more, with the system expected to become generally available to scientific users by January 2019. However, select application developers at the Department of Energy and a handful of universities will get a crack at it well before that. McCorkle told TOP500 News that the pre-production Summit will be available via the Center for Accelerated Application Readiness, an early-access program designed to allow developers to port and optimize grand challenge codes for Summit's new CPU-GPU architecture.

All of that suggests that the system may not be up and running until well into 2018, and will not turn up in the TOP500 list until next June. At that point, absent another surprise from China,

High Level Programming for OpenPOWER and CORAL: A case study

Compiler View of OpenPower Architecture

Fat cores

SMT 8 (8 threads/core)

160 threads/processor

Complex branch prediction

Out-of-order pipeline

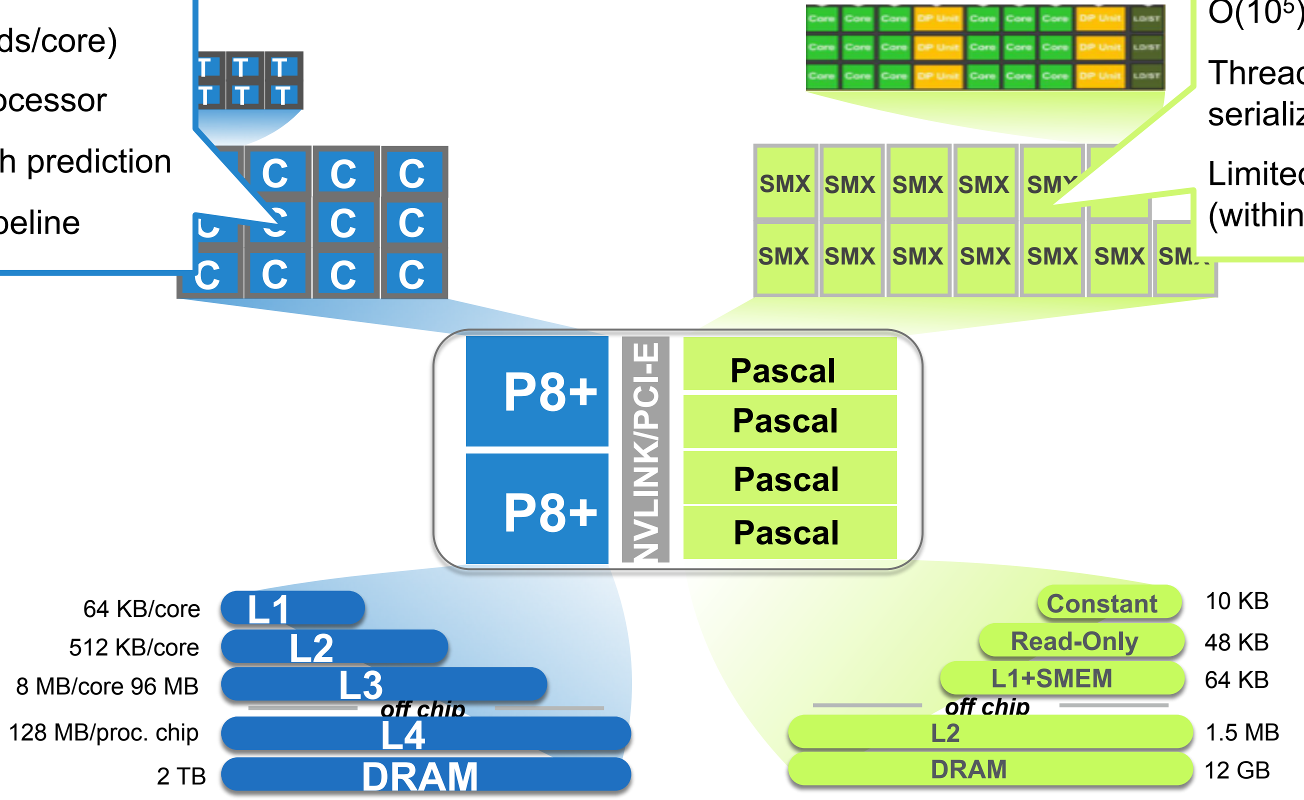
Highly parallel “thin” cores

2048 threads/SMX

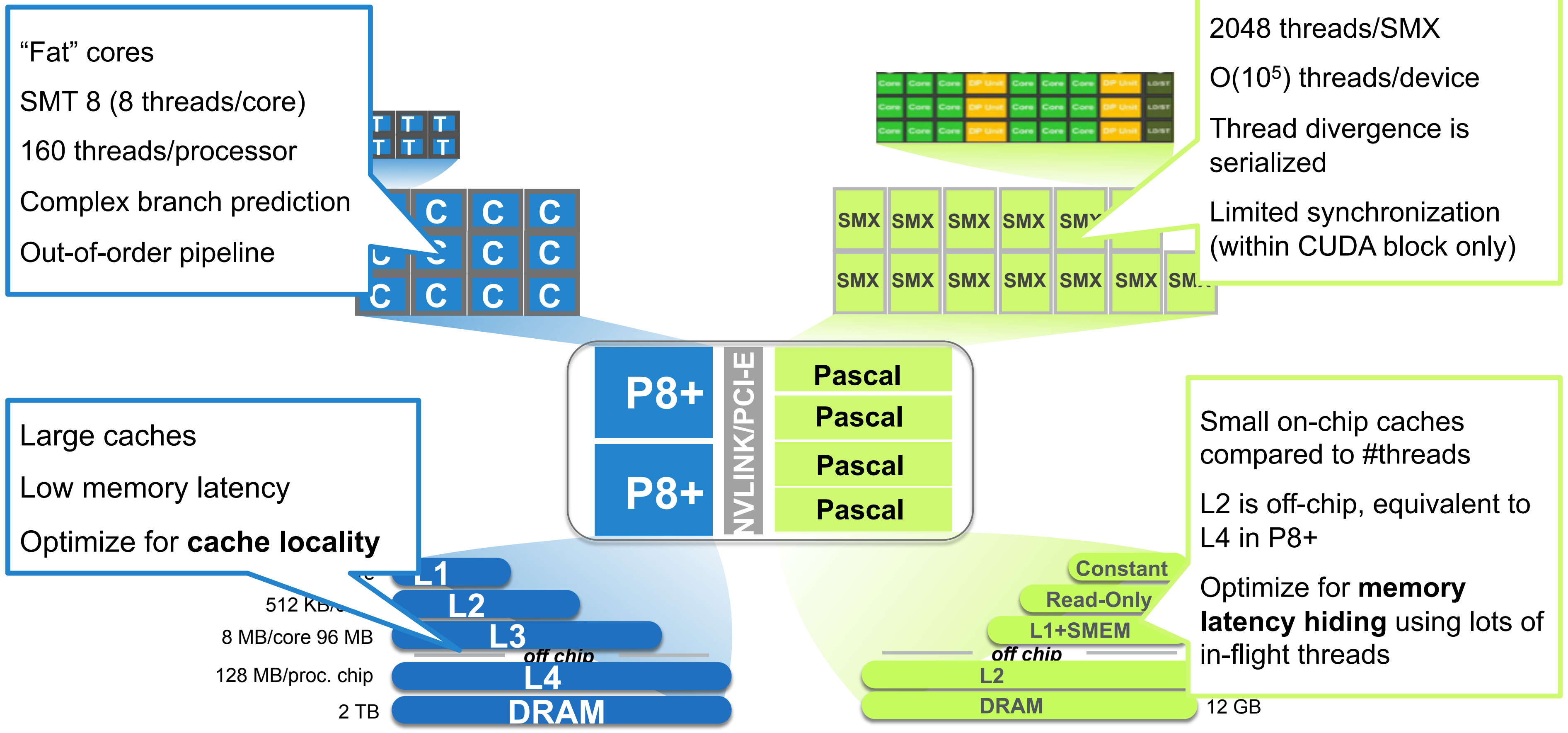
O(10^5) threads/device

Thread divergence is serialized

Limited synchronization (within CUDA block only)



Compiler View of OpenPower Architecture



"Fat" cores

SMT 8 (8 threads/core)

160 threads/processor

Complex branch prediction

Out-of-order pipeline

Highly parallel "thin" cores

2048 threads/SMX

O(10^5) threads/device

Thread divergence is serialized

Limited synchronization (within CUDA block only)

Large caches

Low memory latency

Optimize for **cache locality**

Small on-chip caches compared to #threads

L2 is off-chip, equivalent to L4 in P8+

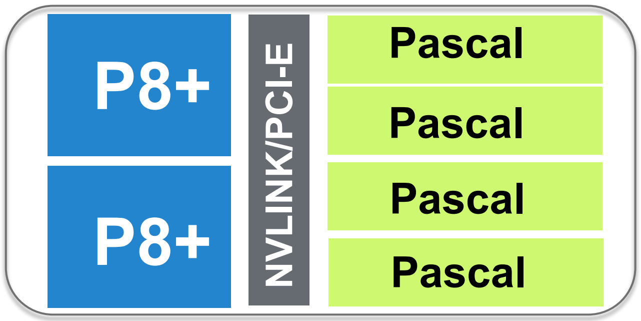
Optimize for **memory latency hiding** using lots of in-flight threads

OpenPower: Flexible Acceleration

The OpenPower Architecture delivers best performance and flexible acceleration beyond HPC field

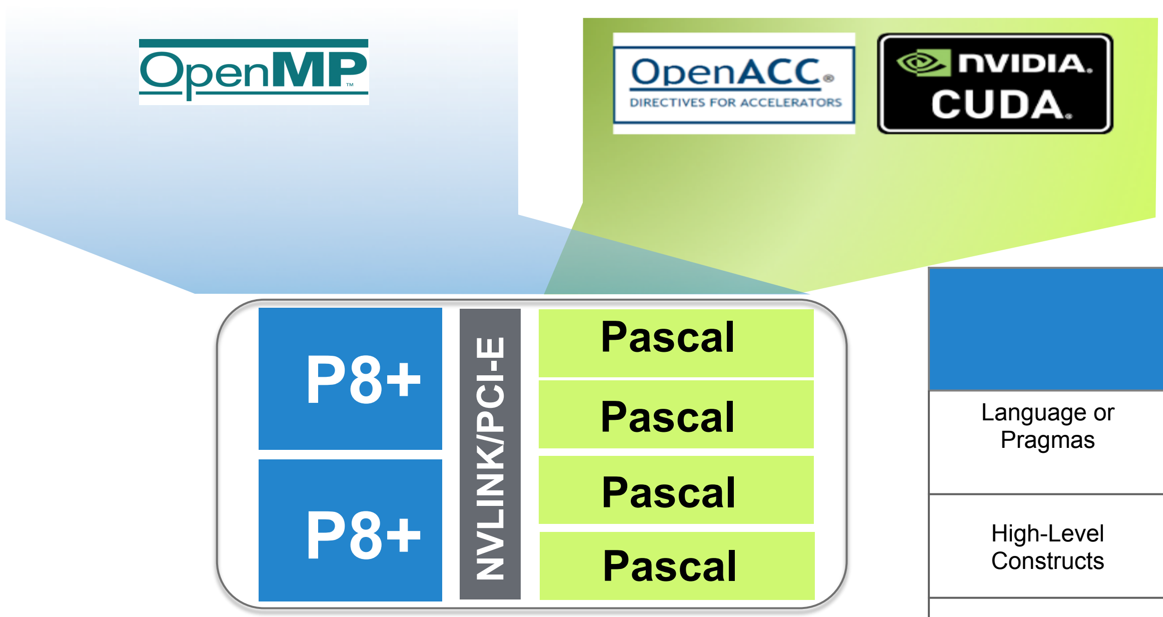
- Balanced combination of Power processors and NVIDIA GPU accelerators
  - High memory bandwidth, low memory latency
  - High memory bandwidth, large number of FMA units

How to effectively and efficiently program OpenPower becomes critical



PROGRAMMING OPTIONS FOR CORAL/OPENPOWER

Programming Options



	OpenMP	OpenACC	CUDA
Language or Pragmas	**Pragmas**	Pragmas	Based on C/C++ or Fortran
High-Level Constructs	**Parallel, loop-like, simd, tasking, etc.**	Parallel, loop-like, simd	None
Companies/ Organizations Actively Involved	**IBM, AMD, Intel, Cray, Pathscale, Texas Instruments, etc.**	Cray, PGI, gcc	NVIDIA, PGI, IBM
Features	**Architecture-independent Flexible Parallelism**	Targeted for GPU-like accelerators	Only available on GPUs

Programming Strategy for OpenPower

OpenPower

- Ideal combination of Power “fat” cores, and NVIDIA ”thin” GPU cores
- Different components of HPC workloads can be executed optimally
- Power cores low memory latency, GPU high memory bandwidth

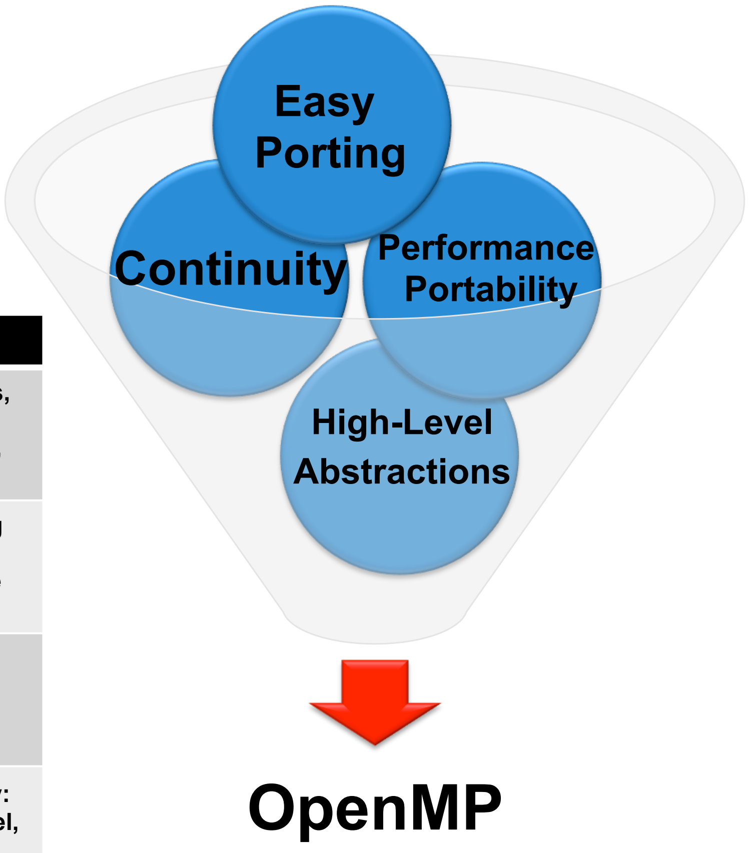
Programming Model Wish List
High Level Parallel Abstractions • Architecture/accelerator independent • High application pattern coverage (dense, sparse, graphs)
Performance Portability • Single version of kernels runs on Power and GPU • With the best performance possible
Easy incremental development • No need to rewrite entire application in new language • Interoperability with assembly kernels and libraries (CUDA, CUDNN) • **Early Availability**
Continuity • Industry standard • Supported everywhere • Survives project/architecture/fashion/etc.

Programming Strategy for OpenPower

OpenPower

- Ideal combination of Power “fat” cores, and NVIDIA ”thin” GPU cores
- Different components of HPC workloads can be executed optimally
- Power cores low memory latency, GPU high memory bandwidth

Programming Model Wish List	
High Level Parallel Abstractions • Architecture/accelerator independent • High application pattern coverage (dense, sparse, graphs)	**Parallel loops, simd, tasks, tasks+loops, etc.**
Performance Portability • Single version of kernels runs on Power and GPU • With the best performance possible	**Early porting shows performance potential**
Easy incremental development • No need to rewrite entire application in new language • Interoperability with assembly kernels and libraries (CUDA, CUDNN) • **Early Availability**	**Code Annotations (#parallel)**
Continuity • Industry standard • Supported everywhere • Survives project/architecture/fashion/etc.	**Supported by: IBM, AMD, Intel, Cray, PGI, Pathscale, etc.**



OpenMP 4.5 Design Goals

Write Once, Run Everywhere with Best Performance

How to program a GPU: CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenACC, etc.

How to program a CPU SIMD unit: intrinsics, OpenCL, or auto-vectorization (possibly aided by compiler hints), etc.

How to program CPU threads: C/C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, etc.

OpenMP 4.5 Design Goals

Write Once, Run Everywhere with Best Performance

How to program a GPU: CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenACC, etc.

How to program a CPU SIMD unit: intrinsics, OpenCL, or auto-vectorization (possibly aided by compiler hints), etc.

How to program CPU threads: C/C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, etc.

With OpenMP 4.5 and up:

- **Same standard to program GPUs, SIMD units, and CPU threads**

OpenPower/CORAL Programming Models Roadmap

Code migration and new code development is a key focus

- Near term: OpenMP4.5, OpenACC,
  - Compiler support will be available for both
  - Interoperability with CUDA important

- Longer term: OpenMP5 and beyond targeted to be the dominant approach
  - Driving towards convergence of both directives standards
  - Evolving unified standards to address portability first and ultimately performance portability
  - Ultimately prefer a directives optional threading model that can support execution across a range of homogeneous and heterogeneous core types (CPU/GPU …)

OpenPower - Programming Languages and Compilers

CUDA

Key Features:

- Gives direct access to the GPU instruction set
- Supports C, C++ and Fortran
- Generally achieves best leverage of GPUs for best application performance
- Compilers: nvcc, pgi CUDA fortran
- Host compilers: gcc, XL

Key Features:

- Designed to simplify Programming of heterogeneous CPU/GPU systems
- Directive based parallelization for accelerator device
- Compilers: PGI, Cray, gcc

Key Features:

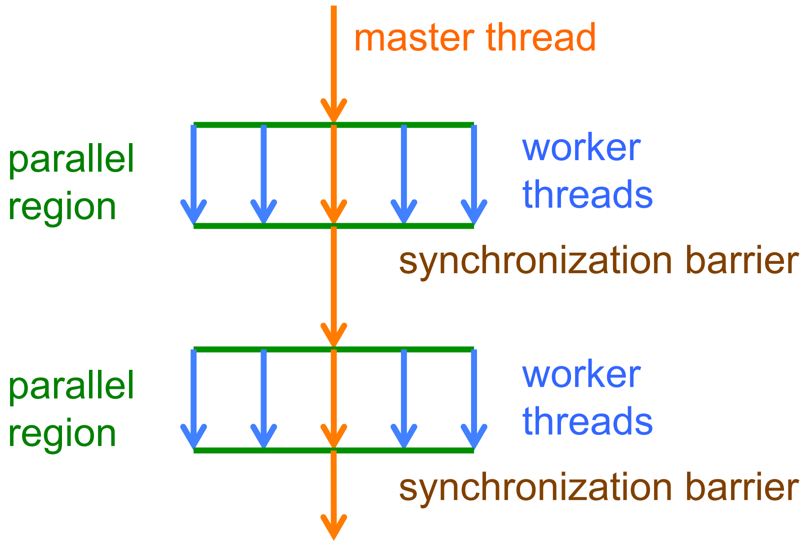
- OpenMP 4.5 offloading and support for heterogeneous CPU/GPU
- Leverage existing OpenMP high level directives support
- Compilers: Open Source LLVM OpenMP Compiler, IBM XL

OPENMP PROGRAMMING MODEL

OpenMP Execution Model for Parallel Regions

Fork and join model

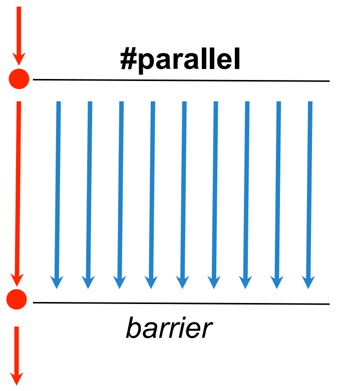
- sequential code executed by the master thread
- parallel code executed by the master and workers
- parallel region terminated by a synchronization barrier
- memory touched in parallel region is "released/flushed" at barrier



Flexible Parallelism

Parallel Loops

```
#pragma omp parallel for
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

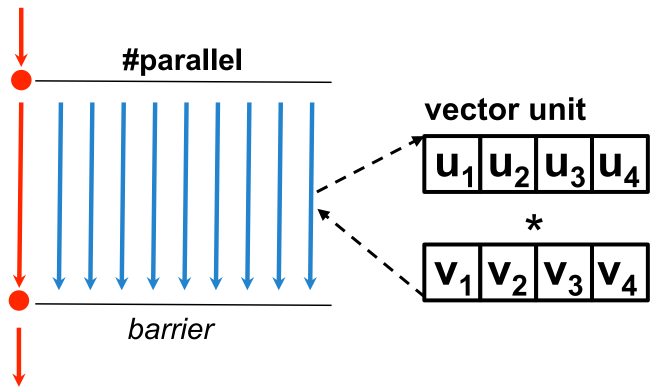


#parallel for

- **#parallel** recruits threads
- **#for schedules M iterations to parallel threads**
- At the end of **#parallel** there is a barrier
- Significant performance optimizations for successive small parallel loops

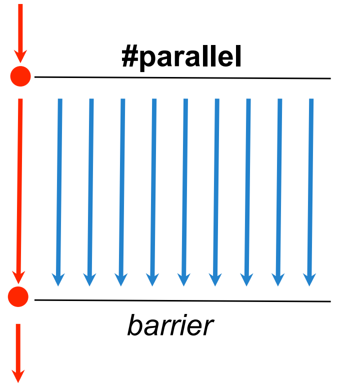
Parallel Loops with SIMD

```
#pragma omp parallel for
for (i = 0; i < M; i++)
  #pragma omp simd
  for (j = 0; j < N; j++)
    A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```

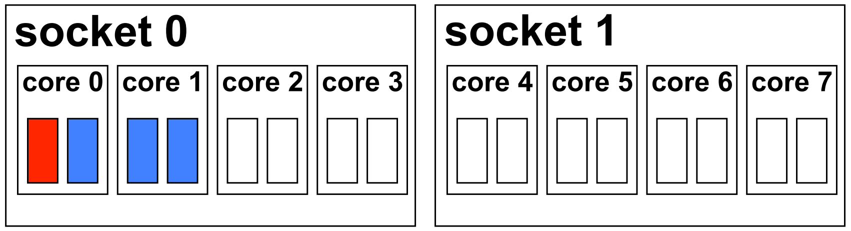


Thread Affinity

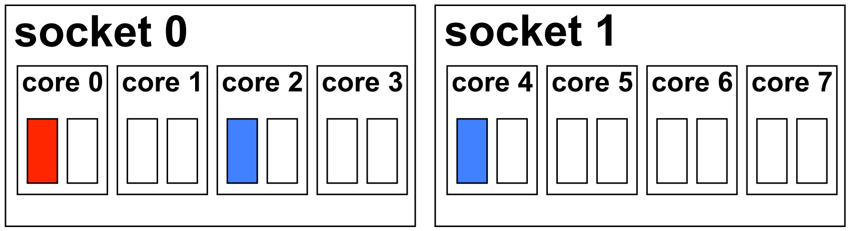
```
#pragma omp parallel for proc_bind(spread)
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    A[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
```



Close affinity: pack threads for cache locality



Spread affinity: spread threads to maximize bandwidth



Why use OpenMP 4 ?

The ultimate goal for developers using OpenMP4.0 and beyond is to achieve:

a) portability

b) performance portability

while using the same source code and compiling it on different platforms.

OpenMP4.5 allows incremental transition of applications:
non-threaded codes can be first parallelized using
OpenMP directives (if algorithm allows parallelization)
tested on the host (CPU) and then
offloaded to the device (GPU)

```
for (i=0; i<N;i++)
  y[i] = a*x[i]+y[i]
```

```
#pragma omp parallel for
for (i=0; i<N;i++)
  y[i] = a*x[i]+y[i]
```

```
#pragma omp target teams distribute parallel for if(0)
for (i=0; i<N;i++)
  y[i] = a*x[i]+y[i]
```

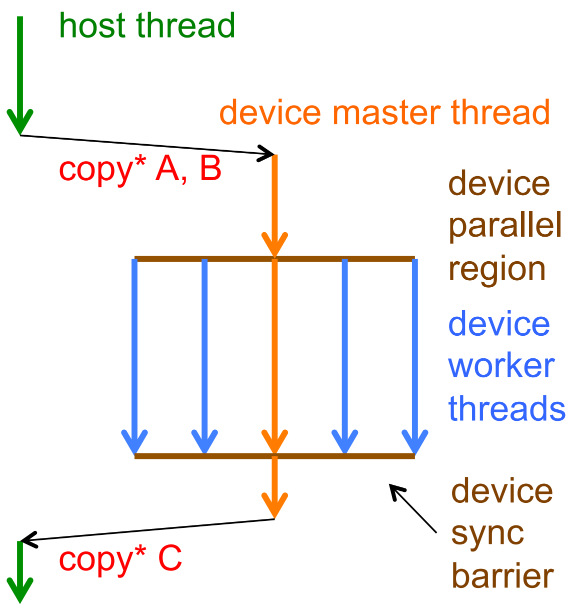
```
#pragma omp target teams distribute parallel for map(to:x[0:N]) map(tofrom:y[0:N]) if(1)
for (i=0; i<N;i++)
  y[i] = a*x[i]+y[i]
```

OpenMP Accelerator Overview

```
integer(4) :: n = 64
real(8), dimension(n,n) :: A, B, C
!$omp target
   map(to: A, B) map(from: C)

    !$omp parallel do
    do i = 0, n, 1
      do j = 0, n, 1
        do k = 0, n, 1
          C(i, j) = A(i, k) * B(k, j)
        end do
      end do
     end do

$!omp end target
```



- **target** transfers control of execution to a SINGLE device thread
- **map** clause is used to fine tune copying of data; default is “**map(tofrom:)**”

▪* at most one copy of each data structure exists on a device; outermost target map copies data to/from device, copies optional with unified memory

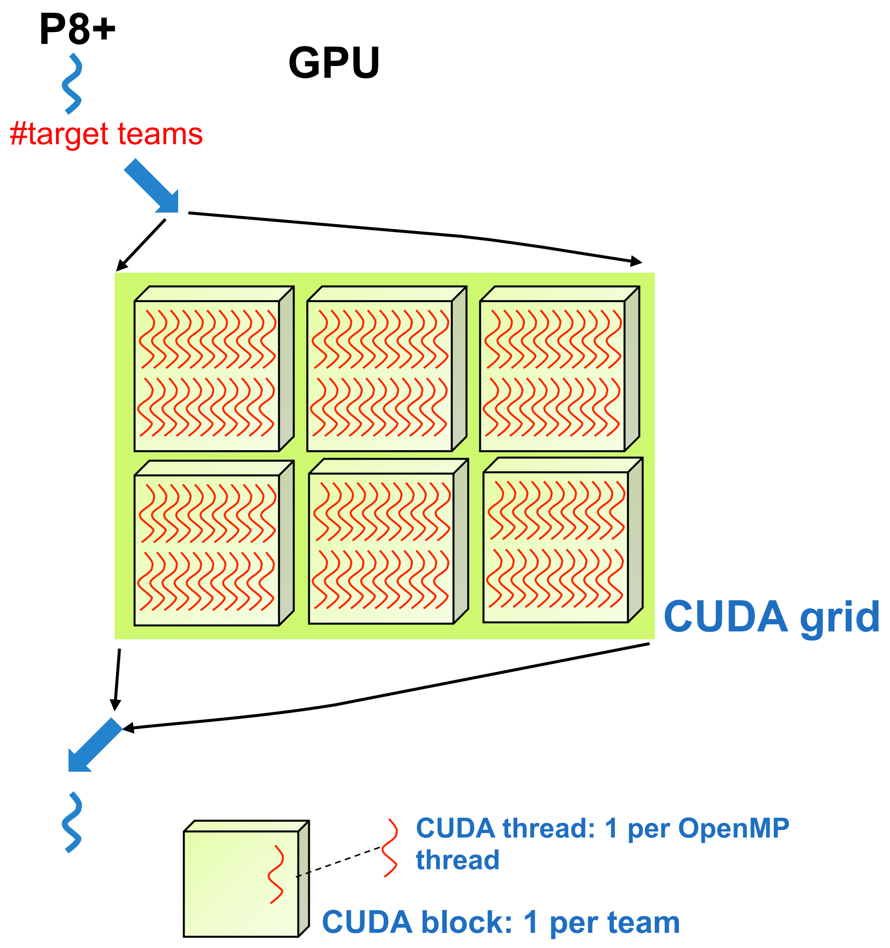
Flexible Parallelism on Devices

Parallel Loops on GPU

```
!$omp target teams distribute parallel for
do j = 0, M, 1
  do i = 0, N, 1
    A(i,j) = A(i,j) + u1(i) * v1(j) + u2(i) * v2(j)
```

On the GPU

- Target offloads region to GPU
- **Each team corresponds to a CUDA block**
- **OpenMP threads are CUDA threads**
- **distribute schedules blocks of iterations to teams**

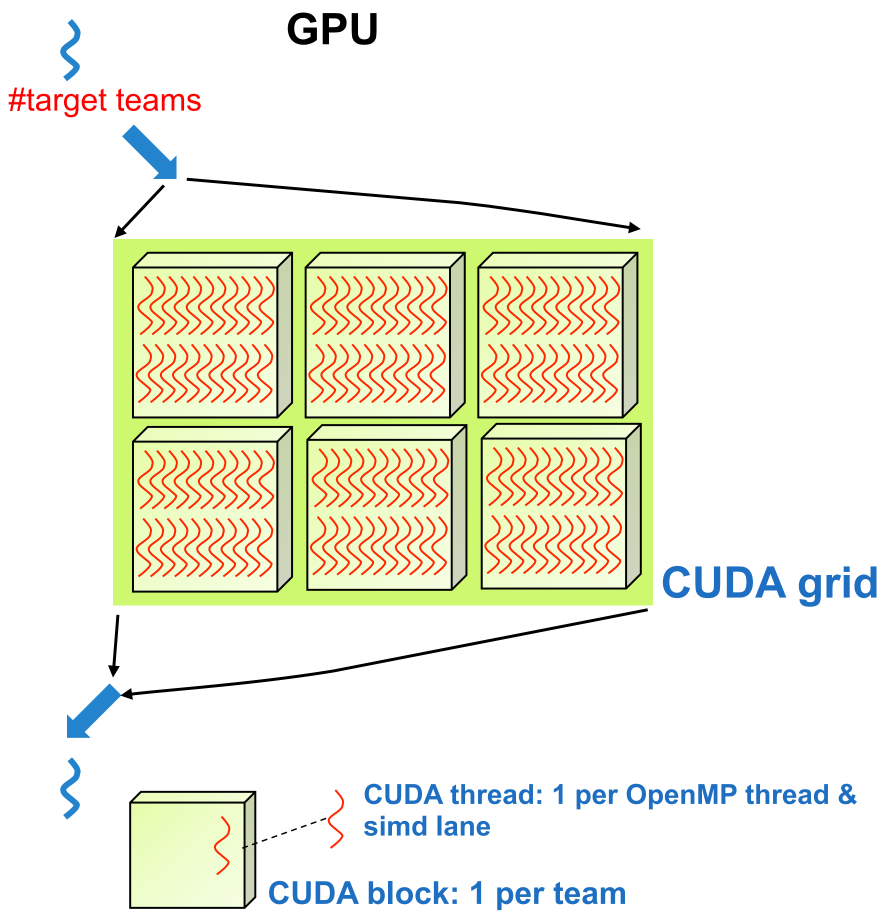


Flexible Parallelism on Devices

Parallel Loops with SIMD on GPU

```
#pragma omp target teams distribute parallel for
for (i = 0; i < M; i++)
  #pragma omp simd
  for (j = 0; j < N; j++)
    A[i][j] += u1[i*M+j] * v1[j] + u2[i*M+j] * v2[j];
```

- **simd** inside **parallel** is widely used on host
  - Leverage vector units per thread
- The GPU has no vector units
  - **Map simd lanes into CUDA threads**



CPU & GPU Parallelism using Tasks

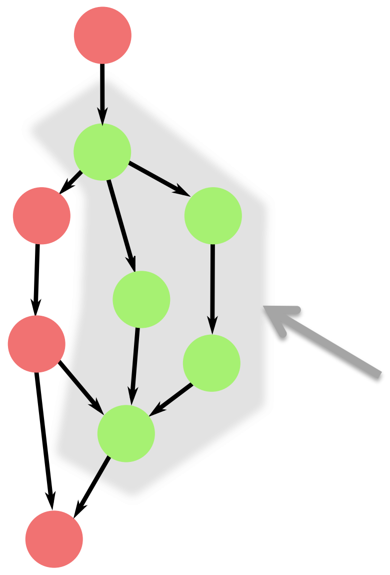
Target constructs are *implicit tasks*

A host thread may initiate several target tasks asynchronously

Target tasks may have dependencies

🔴 **Host task**

🟢 **Target task**



Dependencies between target tasks are resolved completely on the GPU without host intervention

CPU & GPU Parallelism

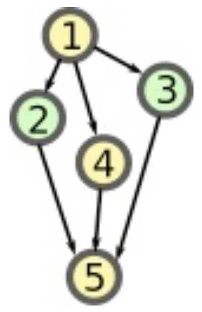
Concurrency in a node

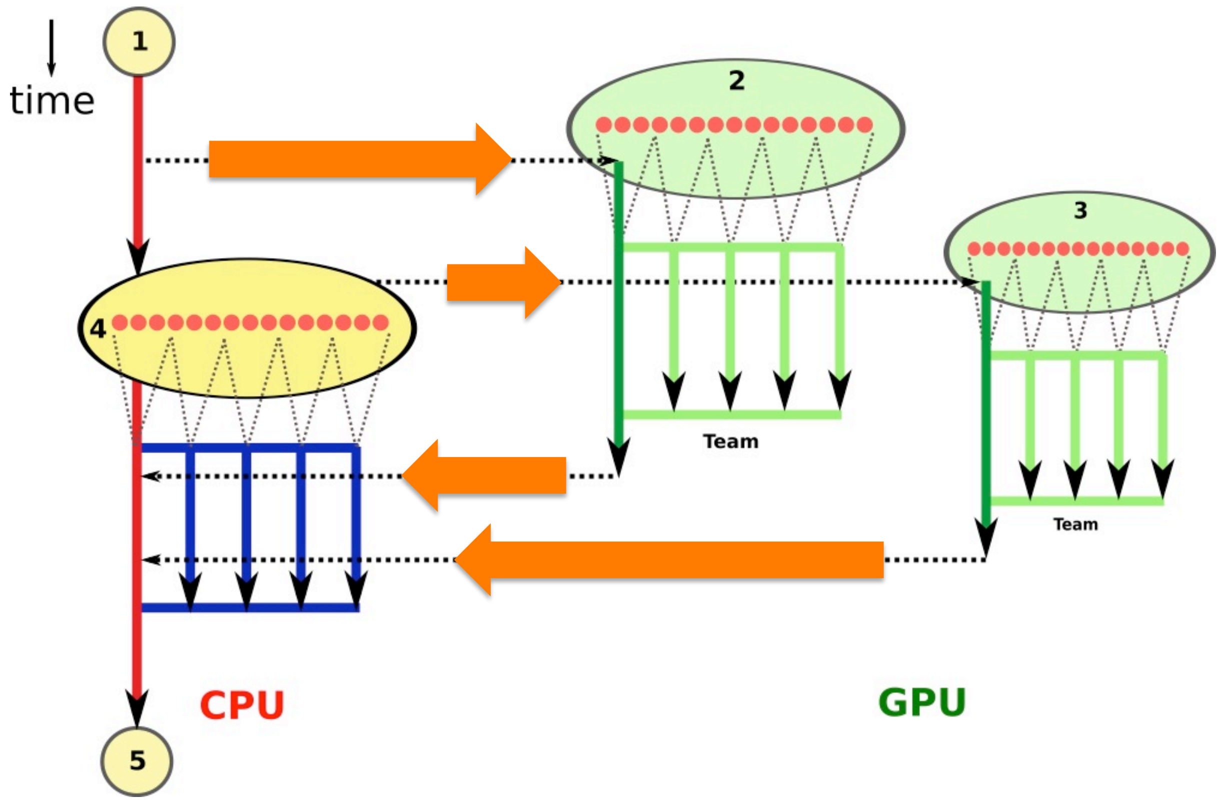
Host threads and device threads

Multiple GPUs in a node

Overlap device computation and communication

Concurrent target tasks on a GPU with task dependencies





Target Data

Data **scope** and data **movement**

- Minimize transfers by design

Data types that can be mapped

- **Scalars**, static and dynamic **arrays**, **structured** data types (struct, class, type)

Memory Model

- Distributed memory in the current implementation

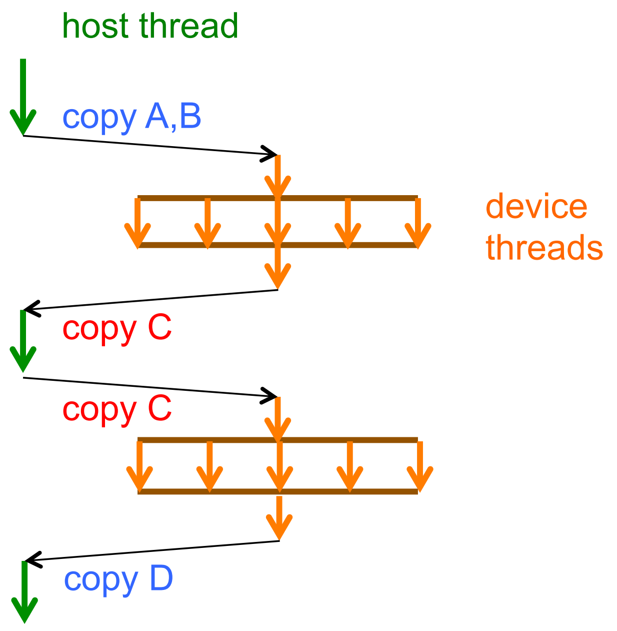
Overcoming Data Movement

Scope of data is important

```
double A[n,n], B[n,n], C[n,n];

#pragma omp target \
   map(to: A, B) map(from: C)
{
 // define C in terms of A, B
}

#pragma omp target \
   map(to: C) map(from: D)
{
 // define D in terms of C
}
```



- Data scope is limited by the target constructs
- No data scope for variable C between the two constructs on the device
- Results in needless copies of C

Overcoming Data Movement (cont.)

Scope of data is important

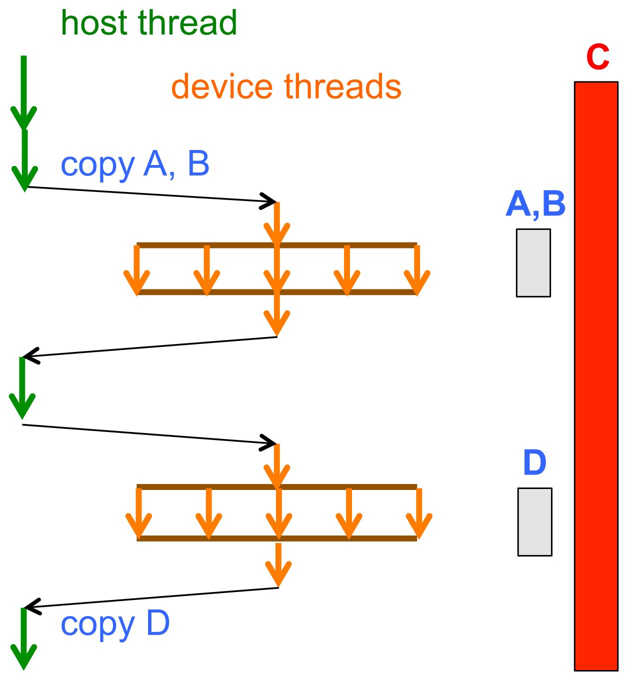
```
real(8) :: A(n,n), B(n,n), C(n,n);

$!omp target data map(alloc: C)

  !$omp target map(to: A, B)
    // define C in terms of A, B
  !$omp end target

  !$omp target map(from: D)
    // define D in terms of C
  !$omp end target

$!omp end target data
```



- C is now a temporary variable that remains on the device
- C is not initialized on the device (**alloc**)

Forcing Data Movement

Device has at most a single copy of each mapped variable

- map clauses are *ignored* when data is already in device scope

```
double A[n,n], B[n,n], C[n,n];
#pragma omp target data map(alloc: C)
{
  #pragma omp target map(to: A, B) map(from: C)
  {
    // define C in terms of A, B
   }

  #pragma omp target map(from: D) map(to: C)
  {
    // define D in terms of C
  }
}
```

c is already in device scope

thus inner map clauses of c are ignored

- Add "**#pragma omp target update from(C)**" force a copy back to the host
- Or use "**always**" qualifier in the map clause, e.g. "**map(always from: C)**"

Unstructured Data Movement

Target enter/exit data do not have a lexical scope

Scope of duration of device data dictated by runtime

```
real(8), dimension(:), allocatable :: A, B, C
allocate(A(N), B(N), C(N))
call init (A, B, C)

$!omp target enter data map (alloc: C)
$!omp target enter data map (to: A, B)

call foo (A, B, C)

$!omp target exit data map (delete: C, B)
$!omp target exit data map (from: A)
```

```
subroutine foo (A, B, C)
  real(8), dimension(:) :: A, B, C

 !$omp target teams distribute parallel for
  do i = 1, N, 1
    C(i) = i
  end do

  !$omp target teams distribute parallel for
  do i = 1, N, 1
    A(i) = A(i) + B(i) + C(i)
  end do
end subroutine
```

Some Data Always Resides on Accelerator

Static data

- Use “target declare” to create a resident copy
- If need to move back and forth, can use “target update”

```
#pragma omp declare target
double A[100];
int *p;
#pragma omp end declare target

#pragma omp target
{
     A[20] = 100;
     p = malloc(10*sizeof(int));
}
#pragma omp target update from(A)
```

- Dynamic data
  - Use “target declare” for pointer to data structure
  - Use malloc within target regions to populate the pointer
  - Cannot bring pack the dynamic data (not mapped)

Summary of Data Scope

Scope linked with device execution: target

- **#pragma omp target map(x) {…}**
- defines a data scope for the duration of execution on device

Pure Scope, without associated device execution: target data

- **#pragma omp target data map(x) {…}**
- only defines a data scope, without launching execution on device

User can also declare data on the device

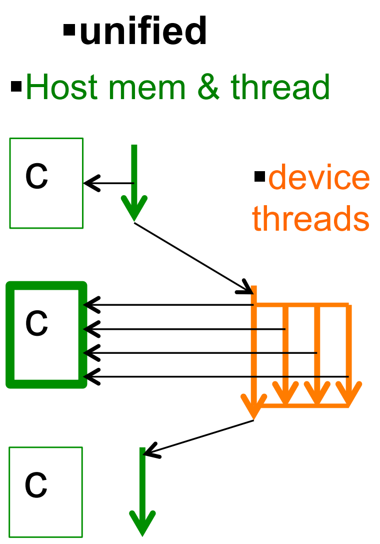
- **#pragma omp declare target to(x)**
- **#pragma omp declare target … #pragma omp end declare target**
- user is responsible to move data back and forth (except for static initialization)

Unstructured pure scopes: target entry/exit

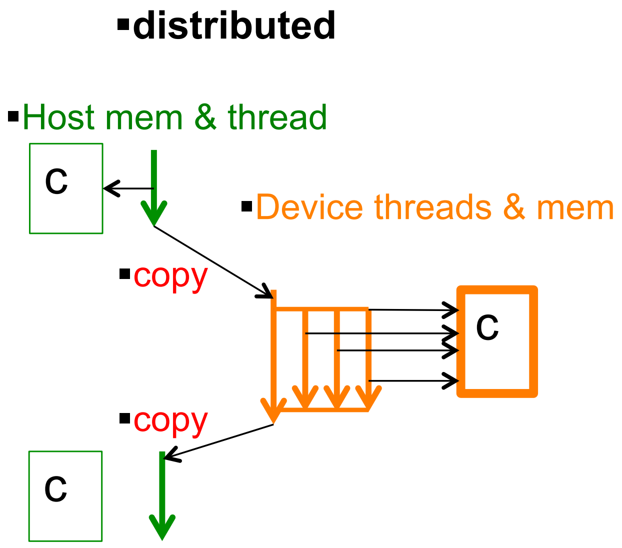
- **pragma omp target enter/exit data map(x)**
- unstructured scope, can be inserted anywhere while executing on the host

Accelerator Memory Model

Programmers may not assume which model is used



```
▪#pragma omp target
▪   map(from: a,b)
▪   map(to: c)
▪{
▪   // define c in
▪   // terms of a, b
}
▪
```



- so the values of c may (unified) or may not (distributed) change during target execution
- user should not assume one or the other in a valid OpenMP program

Accelerator Memory Model: Valid Program

Different results depending on memory model: not a valid program

How to write a legal OpenMP program:

- must schedule a ‘target update’ or ’target map(always:)’
  - each time that a value def/used on one device
  - and then def/used on another device

- [use/use pattern is fine without intervening target update/map always]

Accelerators with Unified Memory: implementation perspective

Map clause does not need to copy data to device private memory

- since it can access shared memory
- user must still have them…

But we (compiler) may decide to selectively copy data

- e.g. read only data accessed by both host and accelerators
  - without copy: may generate misses if not cacheable in both
  - with explicit copy: no misses

- e.g. dense arrays may be copied over
  - single DMA moves all of the data

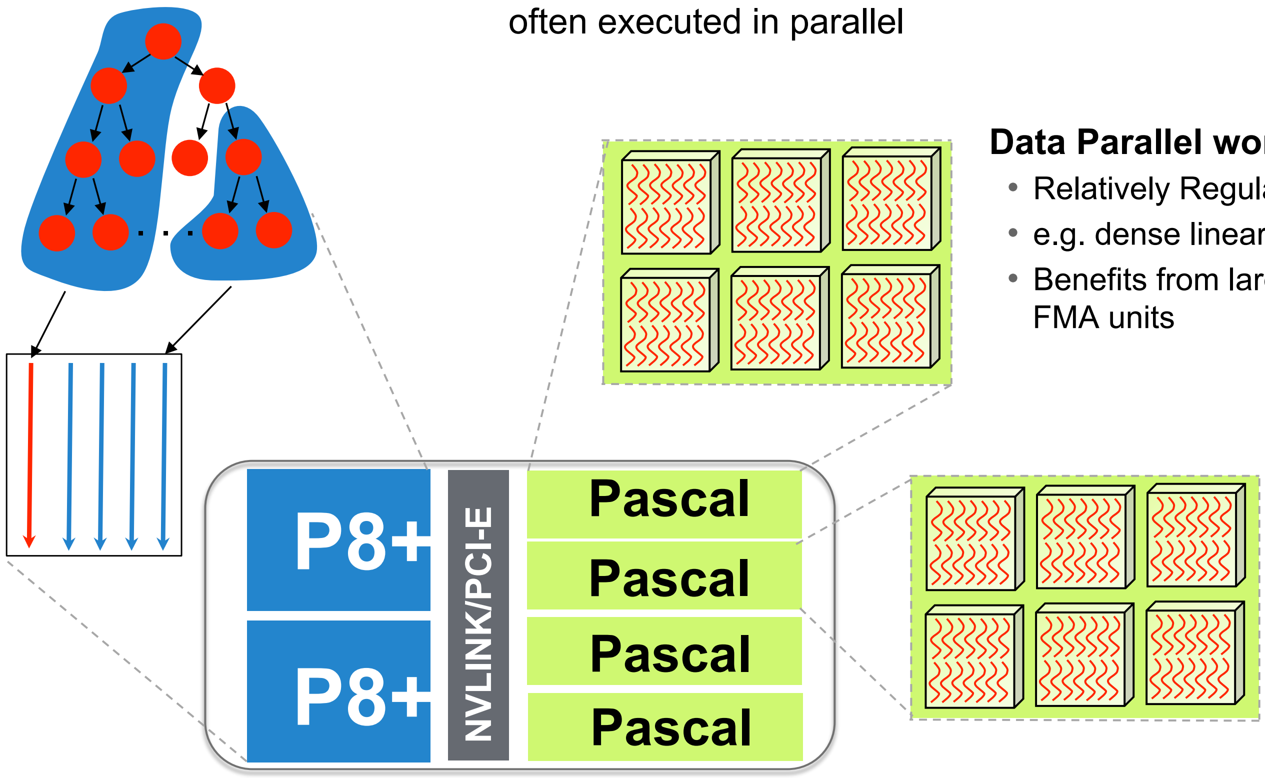
- e.g. data structures with pointers may not be copied over
  - to “deep copy” (feature not avail as of now) a linked list, one needs to DMA each element of the list to the device, update all of the pointers, … and they may not be used anyway

Architecture-based Flexible Parallelism

Real-world scientific applications feature **multi-physics**, often executed in parallel

Graph-like workload

- Highly irregular
- e.g. particle simulation
- Benefits from large caches
- Master thread creates several tasks
  - Dynamically scheduled to free worker threads by runtime



Data Parallel workload

- Relatively Regular
- e.g. dense linear algebra
- Benefits from large numbers of FMA units

Difference with Typical GPU Programming

Traditional models: “Execute this one, exclusively-parallel loop”

- such as found in CUDA, OpenCL,…
- transfer control to a single “parallel loop”
- no sequential code (e.g. to initialize data serially on GPU)

OpenMP model: “Just another normal OpenMP program, on device”

- leverages every+ OpenMP construct
- includes parallel regions, parallel loops, tasks, …
- includes fine grain and coarse grain synchronizations
  - e.g. locks, critical regions, barriers…
- can have sequential and parallel code

OpenMP supports traditional model too:

- it is a “target teams distribute parallel for simd” combined construct

+ exception: target constructs cannot be nested

OpenMP 4+ Features

Directives

- parallel regions
  - thread affinity
- worksharing
  - loop, sections,…
  - *ordered(do across)*
- SIMD
- tasking
  - loops, groups, dep, prio
- accelerator (target)
  - unstructured, *nowait*
- synchronization
- cancellation
- data attributes
  - shared, private [first/last]
  - [user] reductions
  - target: map data to/from
  - target: [first] private, subset

- **Environment Vars**
  - number of threads
  - scheduling type
  - dynamic thread adjustment
  - nested parallelism
  - thread limit
  - description of hardware thread affinity
  - thread affinity policy
  - default accelerator devices

- **Runtime Variables**
  - number of threads
  - thread id
  - dynamic thread adjustment
  - nested parallelism
  - schedule
  - active levels
  - thread limit
  - nesting level
  - team size
  - locks [*hint*]
  - mapping API

- [*italic means in progress*]

APPLICATIONS & PERFORMANCE

KRIPKE – Optimal Performance on CPU and GPU

Original CPU version

```
#pragma omp parallel {
 // Loop over the hyperplanes (slices)
 for (int slice = ...) {
  #pragma omp for
  for (int element = ...) {
   for (int directions = ...) {
    // calculate data depending on d
    for (int group = ...) {
     // calculate new zonal flux
     // apply diamond-difference relationships
    }
   }
  }
 }
}
```

Basic **performance portable** version

- No loop-interchange necessary
- Multiple GPU with multiple host threads on different slices

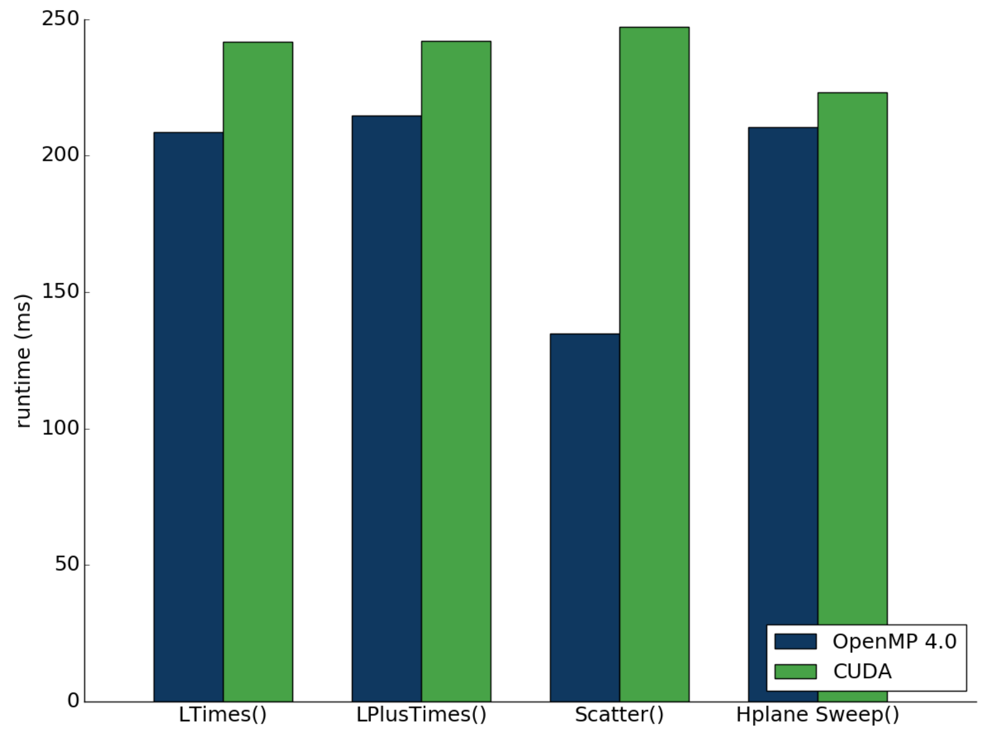
```
// Loop over the hyperplanes (slices).
for (int slice = ...) {
 #pragma omp target teams distribute parallel for collapse(3)
 for (int element = ...) {
  for (int directions = ...) {
   for (int group = ...) {
    // calculate data depending on d and new zonal flux
    // Apply diamond-difference relationships
   }
  }
 } //end element (distribute)
} //end of "for (slice"
```

Performance example: Kripke Runtimes OpenMP vs CUDA

Porting to OpenMP and CUDA started at the same time

- OpenMP version with collapse
  - Complex code synthesis
  - Hard to reproduce in CUDA
- CUDA version uses multiple block dimensions
- Eventually CUDA catches up, after some debugging

OpenPower P8 and K40m NVIDIA GPU



Performance example: LULESH*

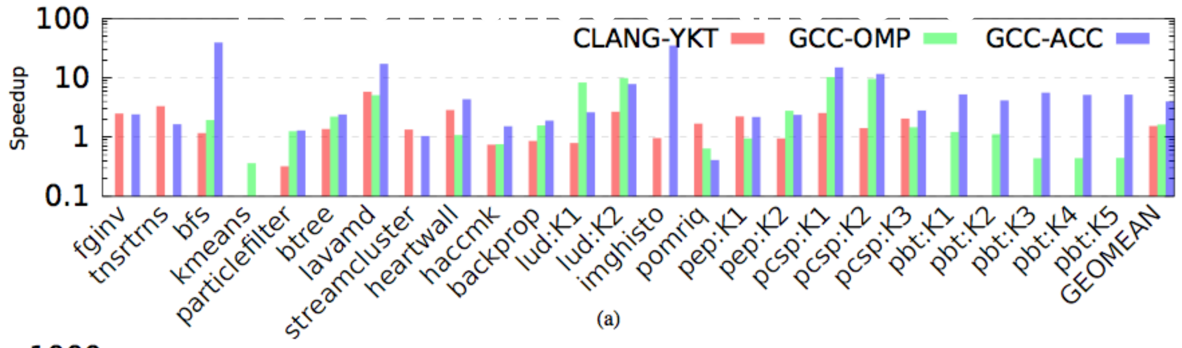
Kernel	OpenMP (us)	CUDA (us)	Speedup (x)
.*CalcTimeConstraints.*	24	42	1.75
.*CalcMonotonicQRegionForElem.*	90	115	1.28
.*CalcLagrange.*	28	30	1.07
.*CalcPositionAndVelocityForNodes.*	67	62	0.93
.*CalcAccelerationForNodes.*	34	31	0.91
.*CalcKinematicsForElems.*	190	130	0.68
.*ApplyAccelerationBoundaryConditions.*	4.6	2.4	0.52
.*CalcMonotonicQGradient.*	197	100	0.51

Better or **comparable** performance

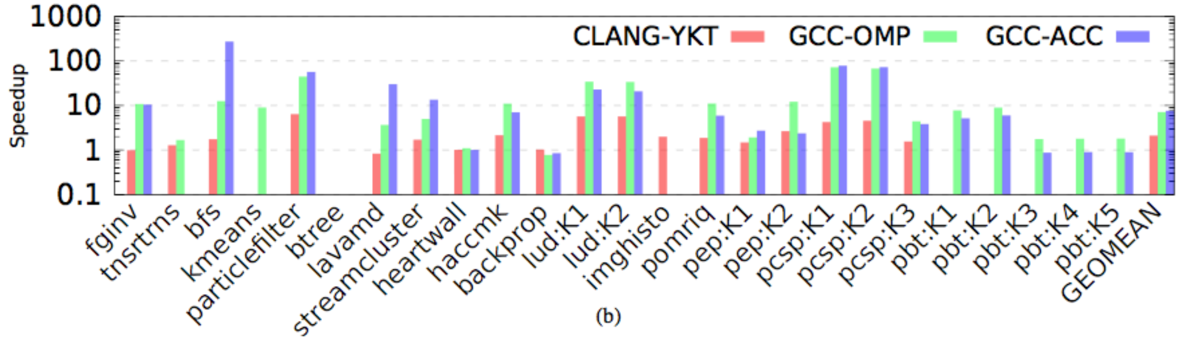
Worse performance

*Courtesy of IBM/Research and XL compiler teams

Performance Example: Some kernels from SpecACCEL, Rodinia and others



(a)



(b)

▪Speedup of our compiler over our prior release and GCC (OpenMP & OpenACC) when exploiting (a) only outer loop and (b) outer and inner loop Parallelism.

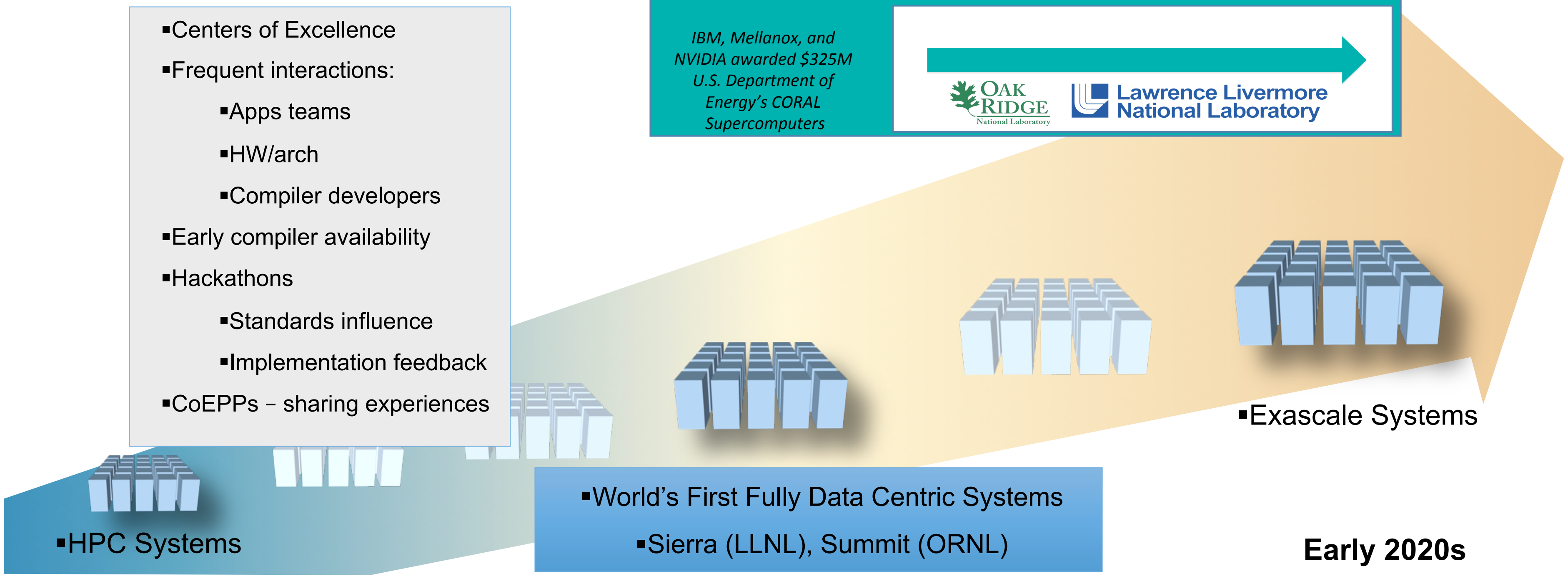
More details at:"Efficient Fork-Join on GPUs through Warp Specialization", Jacob et al, To be published at the IEEE International Conference on High Performance Computing, Data,and Analytics (HiPC 2017)

Summary thoughts

- OpenMP4.5 is a relatively new standard and evolving to OpenMP5.0
  - Implementation of OpenMP4.5 and optimization are on-going efforts: firming the standard, developing compilers, and porting applications are happening concurrently
  - Lessons learned so far from porting complex codes, and specifically from managing multiple memories and data, may lead to new features in the standard and also in its implementation.
- The IBM LLVM implementation is fully 4.5 compliant (also includes some prototyping of new standard features)
- Experience so far, porting applications with OpenMP 4.5 is positive. Often, code portability to various processors is achieved with relatively low efforts. Most performance issues are well understood.
  - Device specific programming models (like CUDA) focus on achieving high performance on specific devices, while compilers implementing OpenMP4.5 should support a variety of processors
  - Large number of kernels written in CUDA and OpenMP4.5 have almost 1:1 mapping and do deliver comparable performance
  - Some kernels rely on intrinsic functions and those may not have the same performance
  - Some kernels coded with OpenMP4.5 use omp collapse(n) clause and may perform better than corresponding CUDA kernels where collapsing loops is not available with compiler directives.

Roadmap to Exascale: co-design through collaborations

IBM



▪Centers of Excellence
▪Frequent interactions:
- ▪Apps teams
- ▪HW/arch
- ▪Compiler developers

▪Early compiler availability
▪Hackathons
- ▪Standards influence
- ▪Implementation feedback

▪CoEPPs – sharing experiences

IBM, Mellanox, and NVIDIA awarded $325M U.S. Department of Energy's CORAL Supercomputers

Oak Ridge National Laboratory
Lawrence Livermore National Laboratory
▪HPC Systems
▪Exascale Systems

▪World's First Fully Data Centric Systems

▪Sierra (LLNL), Summit (ORNL)

Early 2010s

▪7X Performance

Early 2020s

50X Performance

Past:

Focus on HPC performance

Today:

Focus on Data, Analytics, Cognitive, and HPC

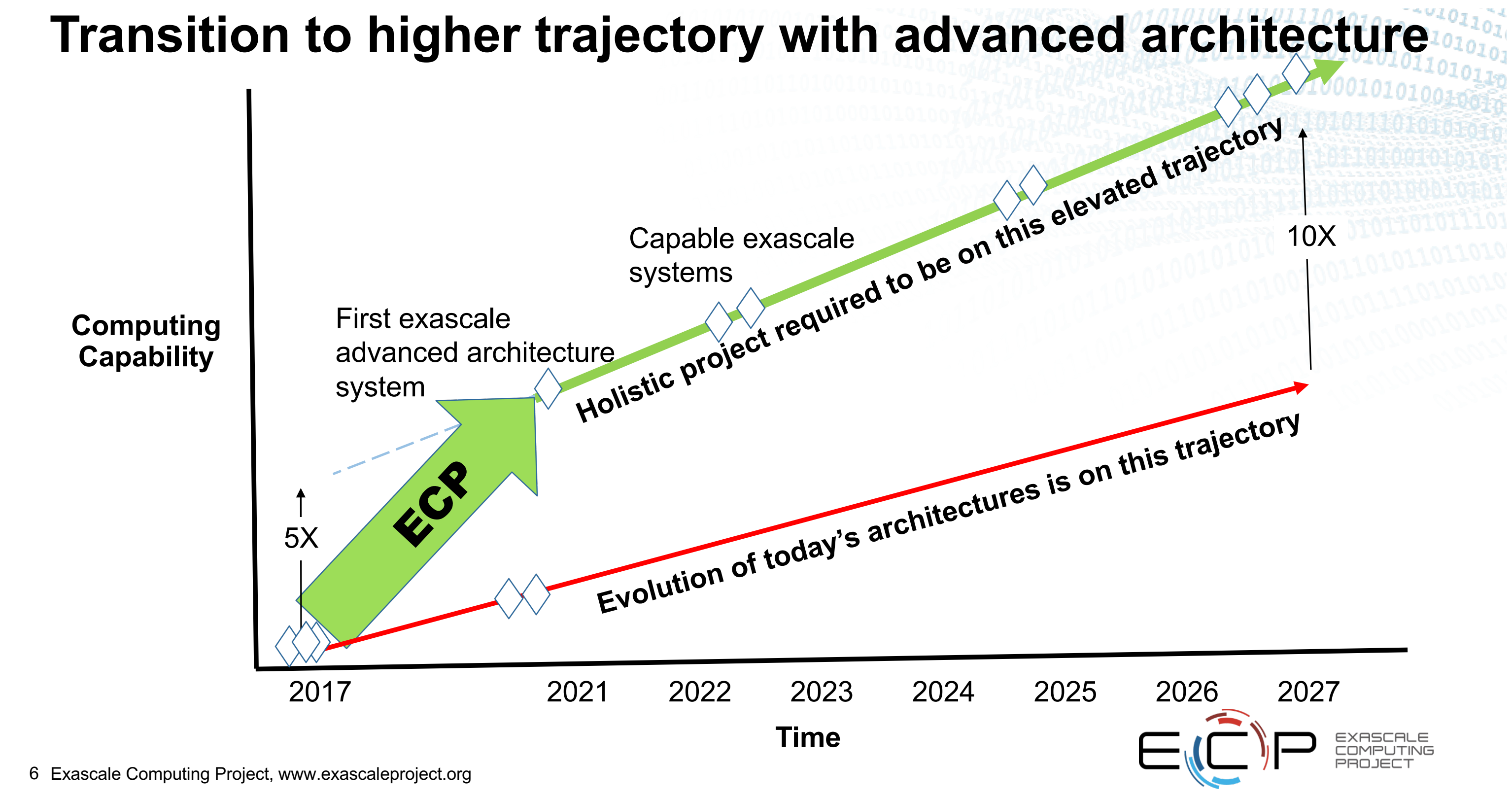
Heterogeneous compute

Future:

Add improved cognitive capabilities,
Heterogeneous compute and memory
Integration of new technologies.

OpenMP looking forward to Exascale

IBM



Transition to higher trajectory with advanced architecture

6 Exascale Computing Project, www.exascaleproject.org

▪https://exascaleproject.org/wp-content/uploads/2017/04/Messina-ECP-Presentation-HPC-User-Forum-2017-04-18.pdf

Reaching the elevated Trajectory …

will require
Advanced and Innovative Architectures

In order to reach the elevated trajectory, advanced architectures must be developed that make a big leap in:

- Parallelism
- Memory and Storage
- Reliability
- Energy Consumption

The exascale advanced architecture developments benefit all future U.S. systems on the higher trajectory

In addition, the exascale advanced architecture will need to solve emerging data science and machine learning problems in addition to the traditional modeling and simulations applications.

Some Applications Risks and Challenges

- Exploiting on-node memory and compute hierarchies
- Programming models: what to use where and how (e.g., task-based RTS)
- Integrating S/W components that use disparate approaches (e.g., on-node parallelism)
- Developing and integrating co-designed motif-based community components
- Achieving portable performance (without “if-def’ing” 2 different code bases)
- Multi-physics coupling: both algorithms and software
- Integrating sensitivity analysis, data assimilation, and uncertainty quantification technologies
- Understanding requirements of Data Analytic Computing methods and applications
  - Critical infrastructure, superfacility, supply chain, image/signal processing, in situ analytics
  - Machine/statistical learning, classification, streaming/graph analytics, discrete event, combinatorial optimization

Exascale Programming Models: statements from ECP in 2016

- The next layer in the software stack consists of programming models and runtimes.
- In the context of exascale systems, the programming model primarily provides a way for the applications to express how they intend to run in parallel. Such capability is important because the languages that are commonly used in HPC applications—primarily C++ and Fortran—don't have built-in language features to efficiently convey the abundance of parallelism that must be exploited.
- The most common programming model in use today generally is referred to as MPI+X. MPI is the Message Passing Interface used for internode distributed memory communication, and "X" refers to a number of shared-memory threading models such as OpenMP, OpenACC, OpenCL, and CUDA for using on-node parallelism and heterogeneous computing devices such as graphics processing units and fine-grained shared-memory threading.
- OpenMP represents a community standard with the ultimate objective of working effectively across the wide variety of nodes. Other ECP efforts provide language-based libraries that allow the application to select from a palette of programming models most suitable for a particular platform. Both approaches focus on achieving performance-portability, or the ability for an application to run effectively on multiple exascale platforms without the need to maintain multiple versions of the source code.
- In addition to building on MPI+X, the ECP is exploring newer programming models primarily embodied in the concept of asynchronous many-task (AMT) models.
- AMT programming models show early potential in addressing some of the bottlenecks of traditional MPI+X programs such as programmer productivity and are included in the ECP software stack for ambitious application efforts looking to exploit the potential of this new programming model approach.

OpenMP in the Exascale Era: *Strengths* and Challenges

Advances in OpenMP in last 3 years have positioned it well for a dominant role at Exascale

- Evolving the standard to address increasing architectural complexity is being successfully demonstrated
  - Affinity
  - Offload
  - Multi-level Memory
- Reference implementations and research prototyping of proposed features
- ECP SOLLVE project (*SOLLVE: Scaling OpenMP with LLVM for Exascale performance and portability,* Barbara Chapman, Brookhaven National Laboratory (BNL) with ANL, LLNL, ORNL, Rice Univ., UIUC)

OpenMP in the Exascale Era: Strengths and *Challenges*

Significant challenges will need to be confronted going forward:

- Continued support for a broad range of heterogeneity(in progress)
- Performance Portability (In progress)
- Increased Complexity (needs to be contained)
- Backward compatibility – getting things right first time
- Broaden adoption beyond current user community: more users – more feedback
- Expand into additional application domains: Machine Learning, AI …

Performance Portability: what might help

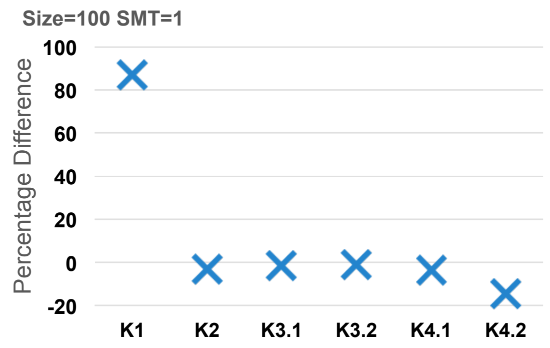
- Centers of Excellence – working well
  - COEPP workshops
  - Hackathons

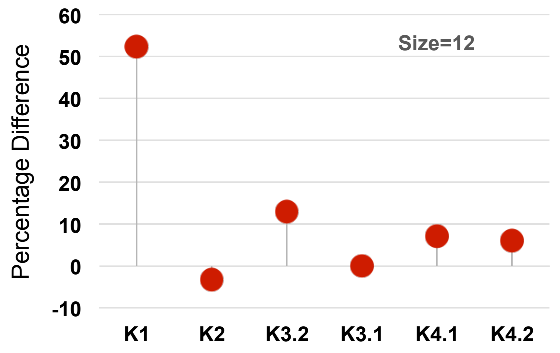
- Interoperation with DSLs, e.g. RAJA

- Increasing consideration of including a ‘descriptive’ capability
  - Support for architectural features such as Unified Memory

Performance Study of OpenMP+RAJA

- Lulesh 2.0 on OpenPower S822LC “Minsky” – Power 8 and Pascal GPU
- Comparison of loop execution times when using vanilla OpenMP vs RAJA+OpenMP





Power 8 Tests

- No impact in memory bandwidth-bound loops
- Missing vectorization limits compute-bound loops
- How to fix: change RAJA std::iteration space use to plain old loop / improve LLVM vectorizer

Pascal GPU Tests

- No performance impact of using lambda in most cases
  - Register allocation figures almost identical
- One loop shows bad performance
  - Only difference with vanilla OpenMP is loading of captured arguments in loop body
  - Can be fixed in compiler

Concluding Thoughts

- The Programming Model Landscape is huge
  - Vendor driven: Exploit proprietary HW features; Can create ‘lock-in’
  - Standards Driven: Create a unified approach that benefits end users
  - Research oriented: Govt funded; University driven
    - Plethora of high level abstractions and home grown DSLs

- Few new models have captured the field in the last 10 years
  - Many have fallen by the wayside
  - Established hybrid models continue to have the most traction(MPI + X)

OpenMP has made significant progress in last 20 years

- Need to seize the moment, gain increased adoption, and capture the broader application spaces at Exascale and beyond!

Thank you!

ibm.com/systems/hpc