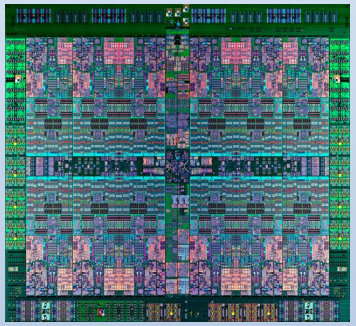
IBM XL Compiler: OpenMP offloading support for GPU

Ettore Tiotto, Kelvin Li
IBM Canada Laboratory

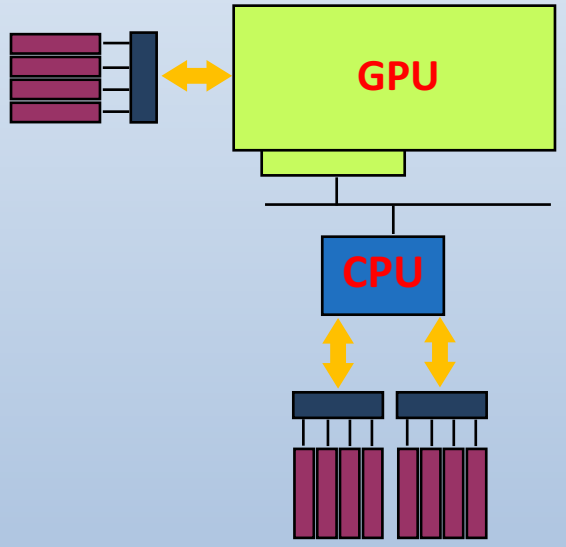
Some slides courtesy of Alex Eichenberg (IBM)

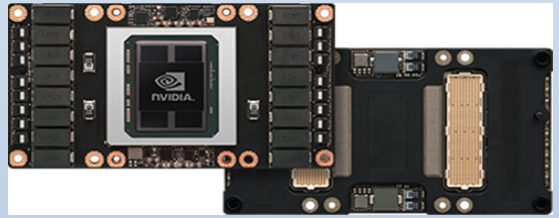
Programming Heterogeneous Systems

- Modern high-performance scientific applications must exploit **heterogeneous resources** in a performance portable manner



IBM POWER8 processors





NVIDIA Tesla P100 GPU

- **What Programming Models should we use? What's the right level of abstraction?**

OpenPOWER™

Programming Heterogeneous Systems

- Extracting maximum performance:
  - to program a GPU: you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenACC
  - to program a host SIMD unit: you have to use Intrinsics, OpenCL, or auto-vectorization (possibly aided by compiler hints)
  - to program the CPU threads, you might use C11, C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, …
- **With OpenMP 4.0/4.5:**
  - **you can use the same standard to program the GPU, the SIMD units, and the CPU threads**
  - **Better yet: you can do so in a portable way**

OpenMP 4.5

- OpenMP is an industry standard for directive based parallel programming
  - OpenMP has been (and is) widely used to program CPUs
  - In OpenMP 4.0/4.5, new features have been added to provide support for offloading computation to accelerators
  - Industry-wide acceptance: IBM, Intel, PathScale, Cray, PGI, Oracle, MS ➔ application portability

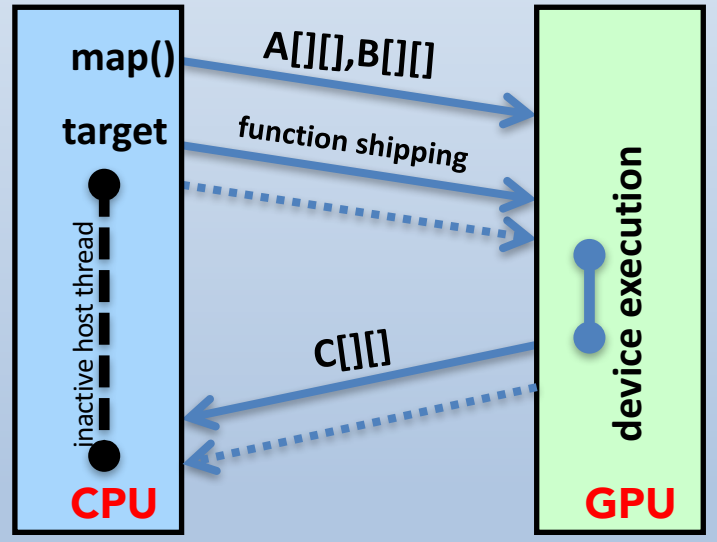
A Quick Introduction to OpenMP 4.5

- How do we exploit an accelerator in OpenMP?
- Simply add a **target** construct around the computation to be offloaded to the accelerator
- **map** clauses are used to copy data

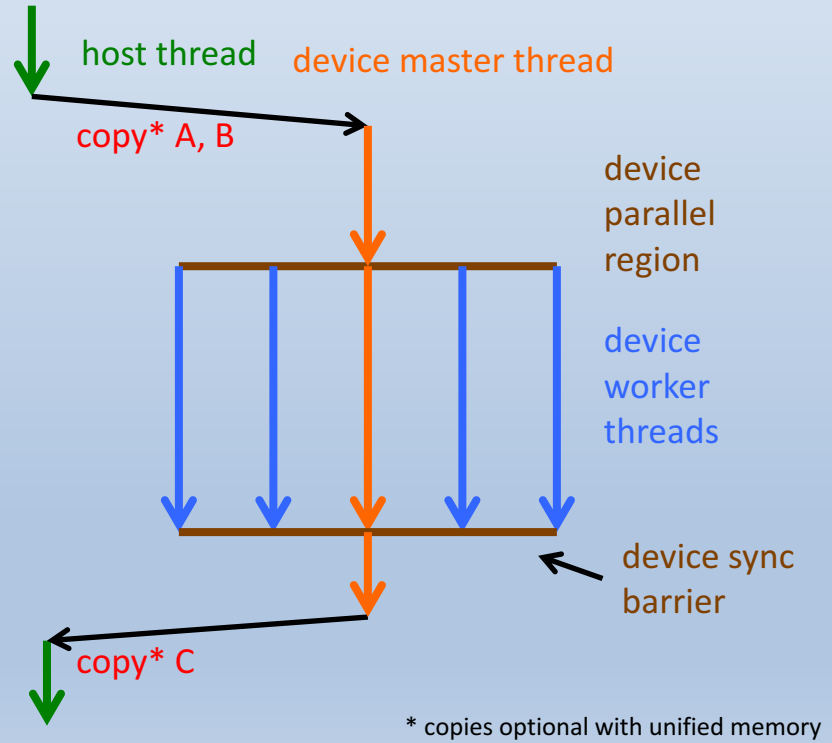
```
#pragma omp target map(to: A, B) map(from: C)
  #pragma omp parallel for
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        C[i][j] = A[i][k] * B[k][j];
  }
```

A Quick Introduction to OpenMP 4.5

- **target** transfer control of execution to a SINGLE device thread
- the compiler packages the target region into a function
- the OpenMP runtime transfer execution of the function to the device




```
#pragma omp target map(to: A, B) map(from: C)
{ … }
```

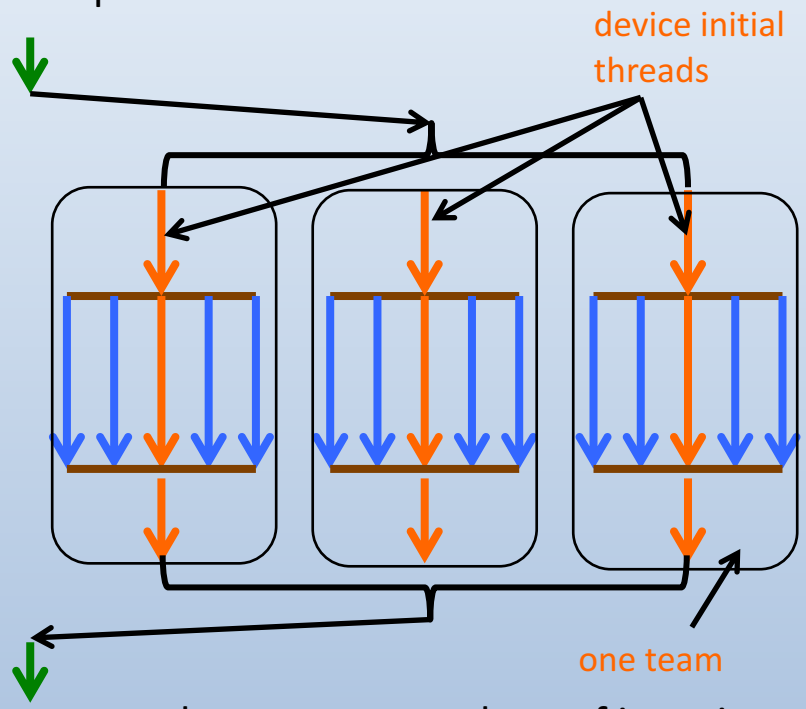


OpenPOWER™

OpenMP 4.5: Device Execution

- The **"distribute"** directive can be used to assign loop iterations to teams

```
#pragma omp target teams
    map(to: a, b) map(from: c)
{
  #pragma omp distribute
  for (int i=0; i<n; i++) {
    #pragma omp parallel for
    for (int j=0; j<n; j++)
      for (int k=0; k<n; k++)
        c[i][j] = a[i][k] * b[k][j];
  }
}
```



- the target region is executed by several teams, each team gets a subset of iteration space for the i-loop
- the j-loop iterations are distributed amongst the threads in a team
- distribute schedule controls size of iterations per team, **there is no synchronization between teams**

Optimization: omp distribute parallel for

- Programming model: OpenMP vs CUDA
  - OpenMP uses a fork-join abstraction
  - team regions start with one thread, and parallel threads are created as needed when a parallel region is found
  - CUDA kernels are launched using a grid of blocks/threads (SPMD model)

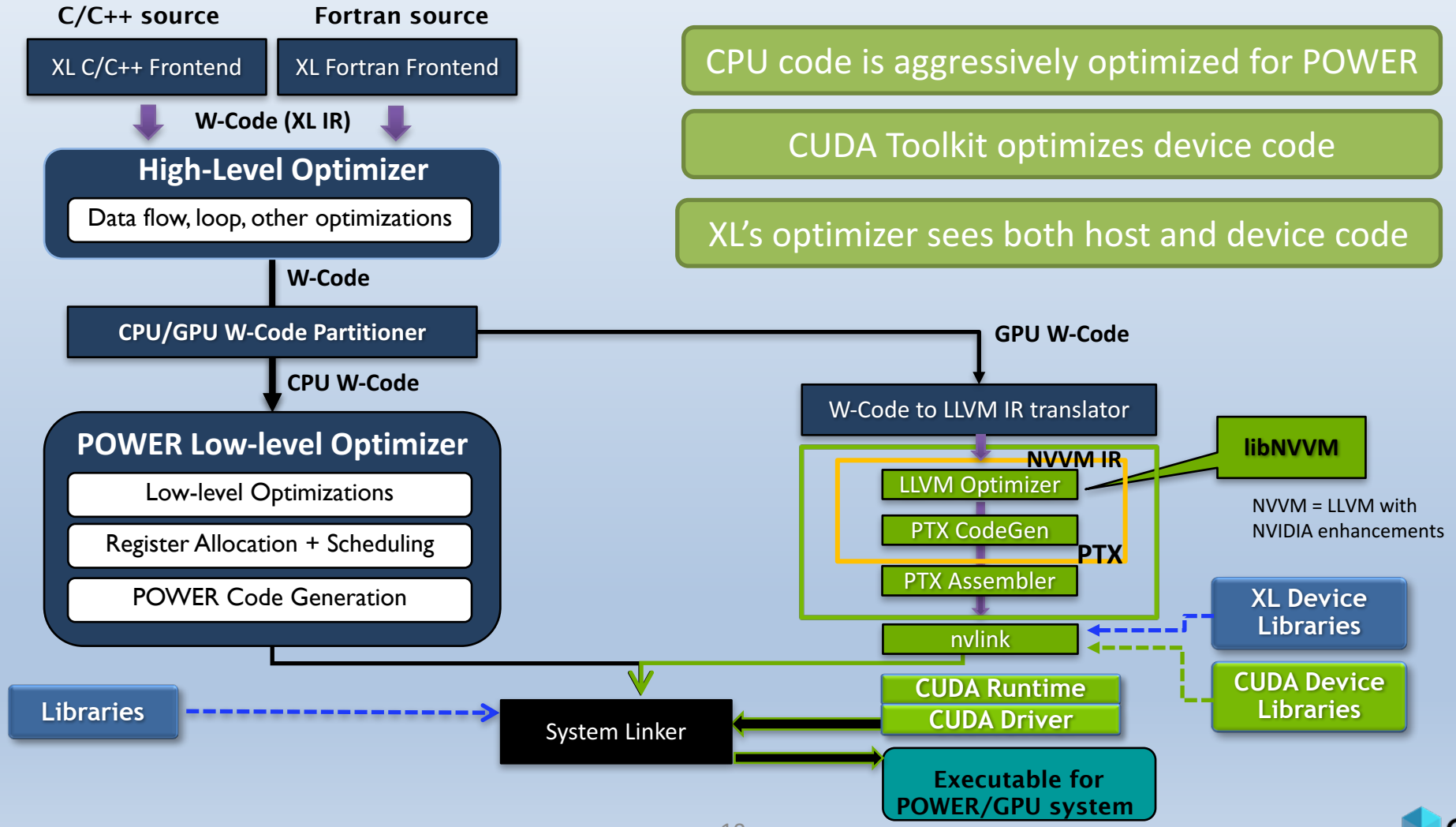
```
#pragma omp target map(from: z) map(to:x,y)
#pragma omp teams
#pragma omp distribute parallel for
  for (i=0; i<N; i++)
    z[i] = a*x[i] + y[i];
```

- Orchestrating CUDA threads to fit the OpenMP programming model can have significant overhead (runtime manages state transitions)
- However OpenMP provides “SPMD-like” directives
  - **distribute parallel for** directive can be used to distribute loop iterations amongst teams and then execute those iteration in parallel using the threads in each team
  - Compiler can generate efficient GPU code for this construct (state transitions not required ➔ bypass OpenMP runtime system)
  - Default schedule recommended to maximize performance portability
    - HW coalescing on GPU, good cache locality on CPU

XL C/C++ and XL Fortran Compilers

- XL C/C++ and XL Fortran are full-featured compilers that has been targeting the POWER platform since 1990
  - Aggressively tuned to take maximum advantage of IBM processor technology as it becomes available
  - Industry leading customer support & service
- The XL compiler products use common optimizer and backend technology
  - Leverage mature compiler optimization infrastructure for both CPU and GPU exploitation, across source languages

OpenMP 4.5 support in XL C/C++ and XL Fortran



OpenMP 4.0 & 4.5 offloading features

Features	OpenMP 3.1 (in target region)	OpenMP 4.0	OpenMP 4.5
OpenMP Directive	***Parallel Construct*** • omp parallel • omp sections • *parallel workshare* ***Worksharing*** • parallel do/for • omp ordered • omp single ***Synchronization*** • omp master • omp critical • omp barrier • omp atomic • omp flush	***Device Constructs*** • omp target data • omp target • omp target update • omp declare target • omp teams • omp distribute • omp distribute parallel for • omp declare target • combined constructs ***SIMD Constructs*** • omp loop simd • Omp distribute parallel do simd • *omp simd* • *omp declare simd* • *omp distribute simd*	***Offloading Enhancements*** • firstprivate, private, default map • map changes (4.5 semantics) • if clause for combined directives • implicit firstprivate (4.5) • omp target enter data • omp target exit data • omp target parallel • *target nowait & depend* • *omp target simd* [***italic means in progress***]

New!

XL Compilers

XL C/C++ for Linux V13.1.5
XL Fortran for Linux V15.1.5

Power Systems

GA:

December

Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

- Initial support for OpenMP V4.5 features for GPU offloading
- Support S822LC systems (POWER8 + P100 via NVLink)
- Support for NVIDIA K40, K80, and P100 GPUs
- Support for CUDA Toolkit 8.0
- Supported Operating Systems: Ubuntu 16.04, RHEL 7.3 ...

OpenPOWER™

compiling the OpenMP programs

- -qsmp=omp option: enables the OpenMP compile in the compiler
- -qoffload option: enables the target constructs being offloaded to GPU (if available)
  - without the –qoffload option, the target regions are executed on the CPU host
- for example

```
$ xlf90 -qsmp=omp -qoffload test1.f

$ xlc -qsmp=omp -qoffload -qhot -O3 test2.c
```

use profiling tool

- nvprof provides information about execution
- output:

```
kli@yc01sros:~/wrk$ nvprof ./test1
==123002== NVPROF is profiling process 123002, command: ./test1
 3
==123002== Profiling application: ./test1
==123002== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 89.18%  21.632us         1  21.632us  21.632us  21.632us  main$_$OL$_$1
  5.94%  1.4400us         2     720ns     512ns     928ns  [CUDA memcpy DtoH]
  4.88%  1.1840us         1  1.1840us  1.1840us  1.1840us  [CUDA memcpy HtoD]

==123002== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 83.50%  89.855ms         1  89.855ms  89.855ms  89.855ms  cuCtxCreate
 12.24%  13.170ms         1  13.170ms  13.170ms  13.170ms  cuModuleLoadDataEx
  2.05%  2.2069ms       364  6.0620us     208ns  229.30us  cuDeviceGetAttribute
  0.75%  803.09us         4  200.77us  195.09us  203.61us  cuDeviceTotalMem
  0.59%  638.39us         1  638.39us  638.39us  638.39us  cuMemAlloc
```

Questions?