

GPU programming models



NERSC Perlmutter training

Brandon Cook

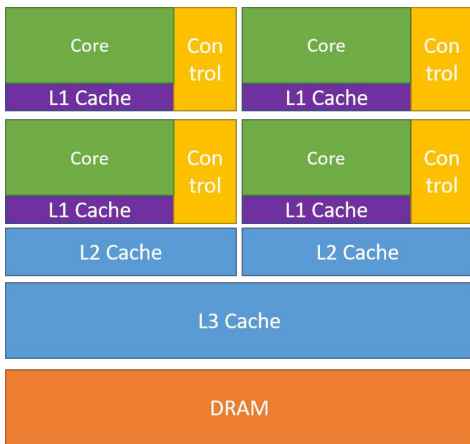
CPU

Optimized to reduce latency

Serial work

Few threads with high frequency

large amount of memory, but slow



CPU

GPU

Optimized for throughput

Parallel work

Many threads

smaller memory capacity, but faster



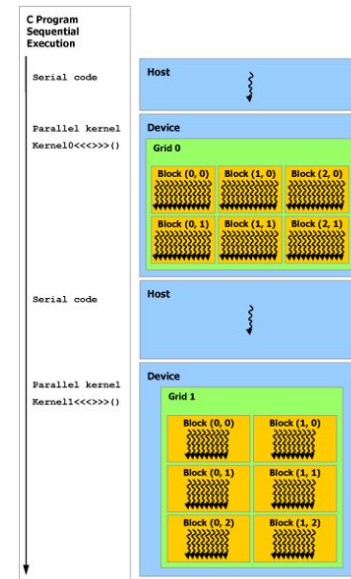
GPU

General principles for CPU + GPU

Offload parallel work to GPU (device)

Keep latency sensitive serial work on the CPU (host)

Keep data where it is used (on device or host)



GPU programming models

Multiple options for compiled languages (i.e. C, C++, Fortran)

Python, ML/AI covered in after the break today

GPU Programming models landscape



CUDA

Native model for NVIDIA GPUs

Reference point for other models

- Full control
- Maximum performance possible
- Not portable (NVIDIA only)
- Verbose

CUDA - kernels

```
// Kernel definition
__global__ void VecAdd(float* A, float* B,
float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

CUDA C++ extends C++ by allowing the programmer to define C++ functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions.

A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a `new<<<...>>>` execution configuration syntax (see [C++ Language Extensions](#)). Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables.

- CUDA C Programming Guide

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htm>

CUDA - thread hierarchy

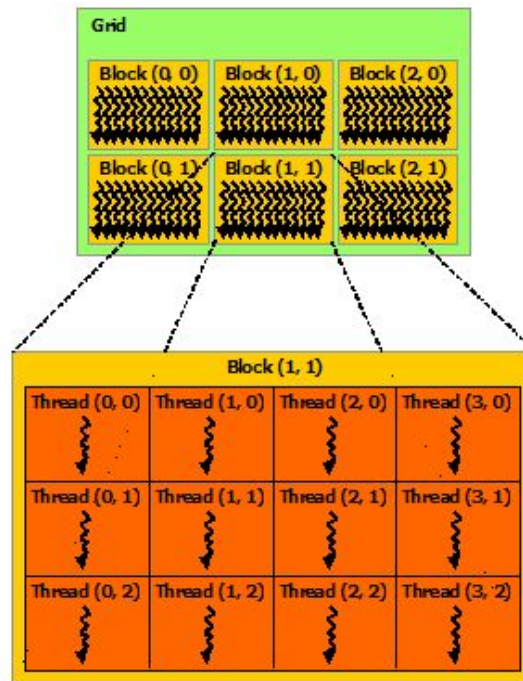
Each kernel consists of a grid

A grid consists of blocks and can be 1,2 or 3 dimensional

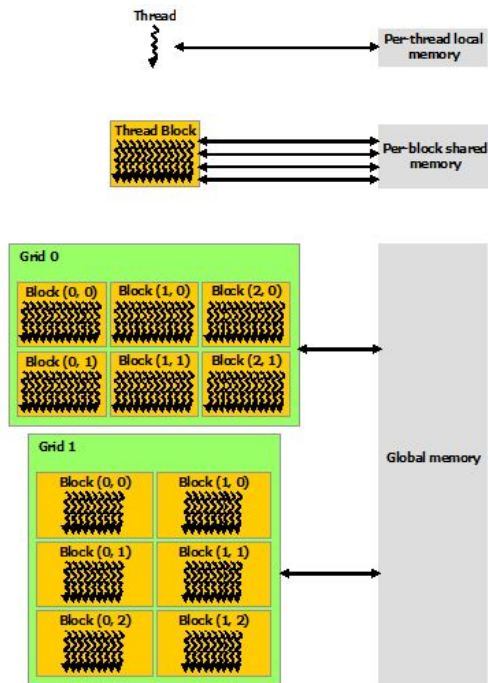
A block consists of threads and can be 1,2 or 3 dimensional

`<<<blocks, threads_per_block>>>`

blocks, threads_per_block is either int or dim3



CUDA - memory hierarchy



CUDA resources

CUDA C++ programming guide

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

Recommended reading no matter the programming language/ model you intend to use

NVIDIA blog and GTC talks/slides

<https://developer.nvidia.com/blog>

<https://www.nvidia.com/en-us/on-demand/>

Tip: There is a lot of CUDA content available, check the dates since CUDA has evolved over the years with added features and relaxed restrictions!

C++ based “frameworks”

Cross platform abstraction layers

Modern C++

Target accelerators and CPUs from multiple vendors

Pro	Con
Powerful abstractions	Requires “buy in”
Integrated tools/ libraries	learning curve
Portability	Vendor support and ecosystem maturity



an “Ecosystem” with
programming model,
memory abstractions, math
kernels, tools, etc



(pronounced ‘sickle’)

a cross-platform abstraction layer
that enables code for
heterogeneous processors to be
written using C++ with the host
and kernel code for an application
contained in the same source file



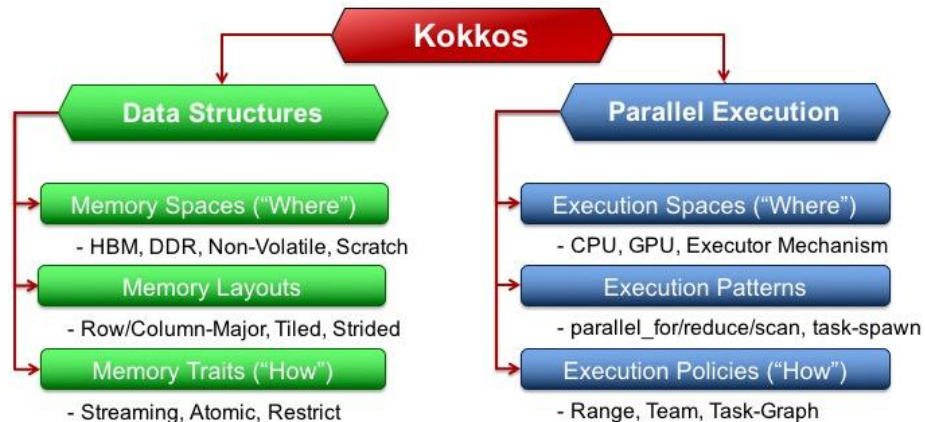
The Kokkos C++
Performance Portability
EcoSystem
[GTC 19 S9662]

- billed as an “Ecosystem” with programming model, memory abstractions, math kernels, tools, etc
- ECP funded project, multiple DOE labs contributing
 - NERSC has staff members contributing
- <https://github.com/kokkos>
 - extensive tutorials, examples, etc available

Kokkos 3: Programming Model Extensions for the Exascale Era
10.1109/TPDS.2021.3097283

Kokkos abstractions

- Views
 - like a shared_ptr to multidimensional data in a “MemorySpace”
 - with a “Layout” ie which index is fast
- Memory spaces
 - Where data is stored
- Execution spaces
 - Where code is run



```

1 #include <cmath>
2 #include <iostream>
3 #include <vector>
4
5 #include <Kokkos_Core.hpp>
6
7 int main(int argc, char *argv[]) {
8     Kokkos::initialize(argc, argv);
9     {
10         int n = 100000;
11         Kokkos::View<double *> a("a", n), b("b", n), c("c", n);
12
13         std::cout << "Kokkos execution space: "
14             << Kokkos::DefaultExecutionSpace::name() << std::endl;
15
16         Kokkos::parallel_for(
17             "initialize", n, KOKKOS_LAMBDA(size_t const i) {
18                 auto x = static_cast<double>(i);
19                 a(i) = sin(x) * sin(x);
20                 b(i) = cos(x) * cos(x);
21             });
22
23         Kokkos::parallel_for(
24             "xpy", n, KOKKOS_LAMBDA(size_t const i) { c(i) = a(i) + b(i); });
25
26         double sum = 0.0;
27         Kokkos::parallel_reduce(
28             "sum", n, KOKKOS_LAMBDA(size_t const i, double &lsum) { lsum += c(i); },
29             sum);
30         std::cout << "sum = " << sum / n << std::endl;
31     }
32     Kokkos::finalize();
33     return 0;
34 }

```

Vector Addition Kokkos

- Accessing everything through views
- “Basic” usage of Kokkos
 - default memory and execution space selected at compile time
- While it doesn’t matter with a 1d case the view abstraction hides the layout of data in memory which allows
 - good cache utilization on CPU
 - coalescing on GPU



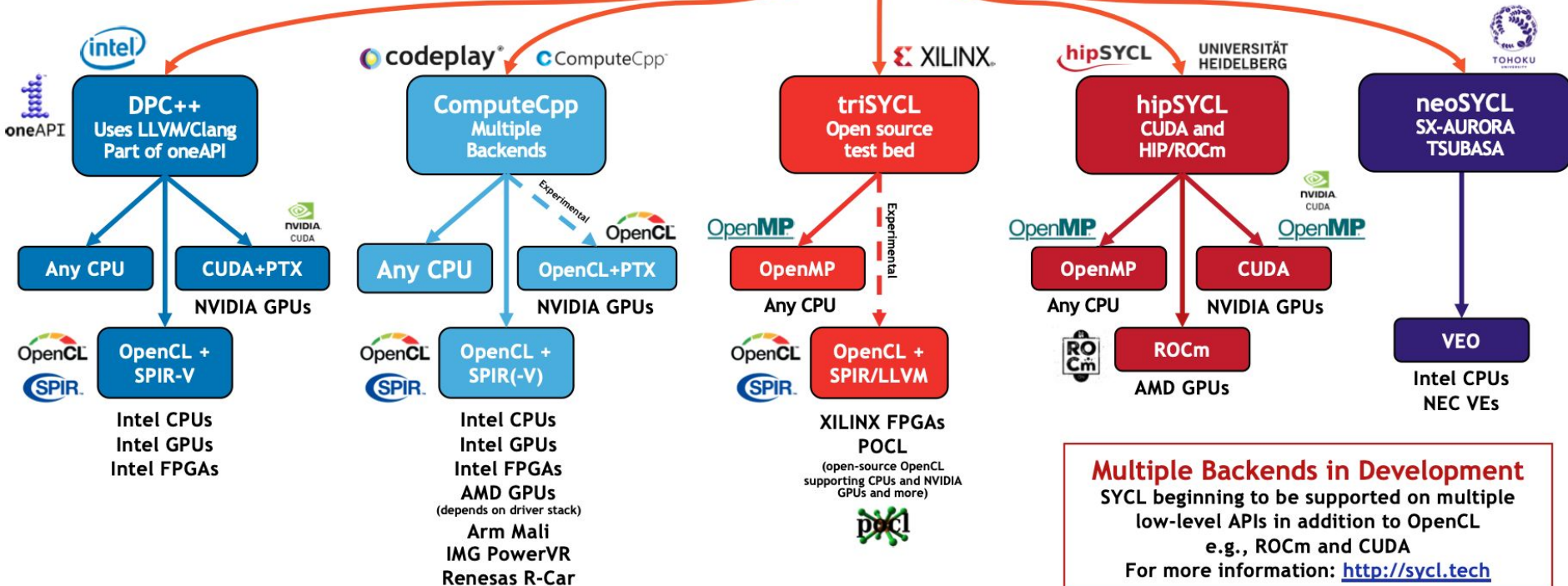
(pronounced 'sickle')

a cross-platform abstraction layer
that enables code for
heterogeneous processors to be
written using C++ with the host
and kernel code for an application
contained in the same source file

- A100 support under active development
- DPC++ is native model for Aurora @ ALCF
 - supported by Intel
- Support from NERSC and Codeplay is available
- Modern C++
- Familiar to OpenCL developers

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute



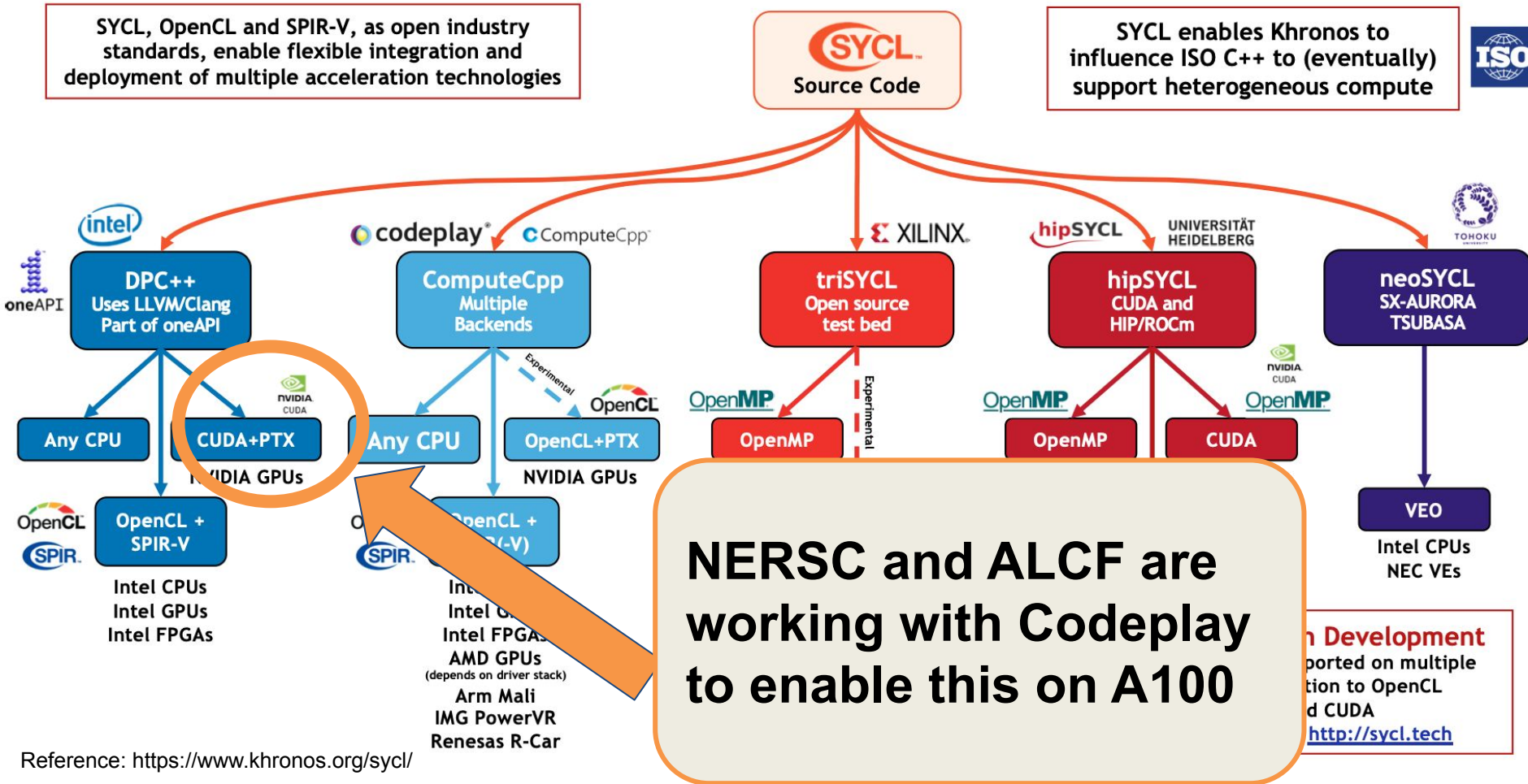
Multiple Backends in Development

SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL e.g., ROCm and CUDA
For more information: <http://sycl.tech>

Reference: <https://www.khronos.org/sycl/>

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute



In Development
ported on multiple
tion to OpenCL
d CUDA
<http://sycl.tech>

Reference: <https://www.khronos.org/sycl/>





- project targeting open-source LLVM based support for A100
 - Perlmutter and ThetaGPU
 - other platforms with A100 ie DGX
- LLVM CUDA backend support for SYCL2020
 - Unified Shared Memory, unnamed lambdas, reductions, subgroups, and more
- Extensions for A100 performance in development
 - Tensor Core APIs/ Types
 - Asynchronous copy and barriers



```

1 #include <CL/sycl.hpp>
2 #include <cmath>
3 #include <iostream>
4
5 namespace sycl = cl::sycl;
6
7 int main() {
8     const int n = 100000;
9     const sycl::range<1> m{n};
10    sycl::buffer<double, 1> b_a{n}, b_b{n}, b_c{n};
11    {
12        auto a = b_a.get_access<sycl::access::mode::discard_write>();
13        auto b = b_b.get_access<sycl::access::mode::discard_write>();
14        for (size_t i = 0; i < n; i++) {
15            a[i] = sin(i)*sin(i);
16            b[i] = cos(i)*cos(i);
17        }
18    }
19    sycl::queue q{sycl::gpu_selector{}};
20    q.submit([&](sycl::handler& h) {
21        auto a = b_a.get_access<sycl::access::mode::read>(h);
22        auto b = b_b.get_access<sycl::access::mode::read>(h);
23        auto c = b_c.get_access<sycl::access::mode::write>(h);
24
25        h.parallel_for<class xpy>(m, [=](sycl::id<1> i) { c[i] = a[i] + b[i]; });
26    });
27    {
28        double sum = 0.0;
29        auto c = b_c.get_access<sycl::access::mode::read>();
30        for (size_t i=0; i<n; i++) sum += c[i];
31        std::cout << "sum = " << sum/n << std::endl;
32    }
33    return 0;
34 }

```

Vector Addition

SYCL with Buffers

- Familiar to OpenCL devs
- Buffers + Accessors allow compiler to infer dependencies and data movement
- Nice idea, but similar amount of tedium as “traditional” CUDA for complex data structures



```

1 #include <CL/sycl.hpp>
2 #include <cmath>
3 #include <iostream>
4
5 namespace sycl = cl::sycl;
6
7 int main() {
8     const int n = 100000;
9     const sycl::range<1> m{n};
10    sycl::queue q{sycl::gpu_selector{}};
11    double *a = sycl::malloc_shared<double>(n, q);
12    double *b = sycl::malloc_shared<double>(n, q);
13    double *c = sycl::malloc_shared<double>(n, q);
14    for (size_t i = 0; i < n; i++) {
15        a[i] = sin(i)*sin(i);
16        b[i] = cos(i)*cos(i);
17    }
18
19    q.submit([&](sycl::handler& h) {
20        h.parallel_for<class xpy>(m, [=](sycl::id<1> i) { c[i] = a[i] + b[i]; });
21    });
22    q.wait();
23
24    double sum = 0.0;
25    for (size_t i=0; i<n; i++) sum += c[i];
26    std::cout << "sum = " << sum/n << std::endl;
27    return 0;
28 }

```

Vector Addition SYCL with USM

- New in SYCL 2020
- Analogous to CUDA with managed memory

SYCL @ NERSC



A100 via LLVM available now!
(Perlmutter modulefile coming soon)

We want to hear from you!

- **#sycl in NERSC User slack**
- help.nersc.gov

Parallel Fortran

```
do concurrent (i=1:n)  
    y(i) = a*x(i) + y(i)  
end do  
  
A = matmul(B,C)
```

Parallel STL

With C++17 comes with several parallel algorithms:

- transform, reduce, for_each_n, ...

See:

- `<numeric>`
- `<algorithm>`
- `<execution>`

Compiler support for this is emerging, nvc++ in particular can generate GPU accelerated code

<https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar>

Adding two vectors with stdpar

```
#include <cmath>
#include <numeric>
#include <vector>
#include <iostream>
#include <execution>
#include <algorithm>

int main() {
    int n = 100000;
    std::vector<double> a(n), b(n), c(n);

    for (size_t i = 0; i < n; i++) {
        a[i] = sin(i)*sin(i);
        b[i] = cos(i)*cos(i);
    }

    std::transform(std::execution::par_unseq, a.begin(), a.end(), b.begin(), c.begin(),
        [](double x, double y){return x + y;});

    auto sum = std::reduce(std::execution::par_unseq, c.begin(), c.end());
    std::cout << "sum = " << sum << std::endl;

    return 0;
}
```

Currently a simple case like adding two vectors in parallel does not require anything more than C++ and a compiler that supports parallel STL:

```
nvc++ -stdpar vecadd.cpp
```

<https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar>

Useful things for Science (and to watch for in the future)

`<atomic>, std::atomic_ref`

`std::barrier`

`<ranges>`

`zip_iterator` - today in thrust, boost

`counting_iterator` - for now write your own, soon `iota`

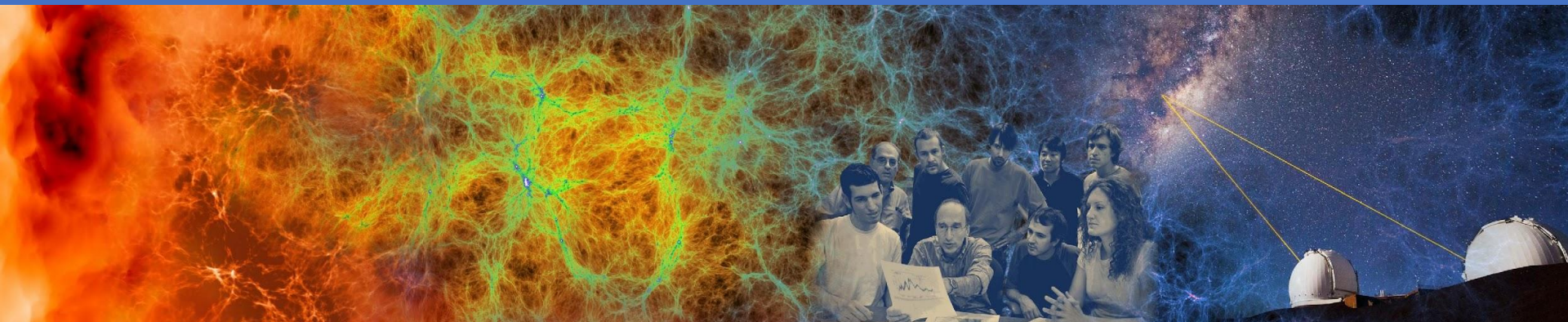
`mdspan` - C++23 (maybe) - <https://github.com/kokkos/mdspan>

<https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar>



OpenMP programming model

Presented by Chris Daley



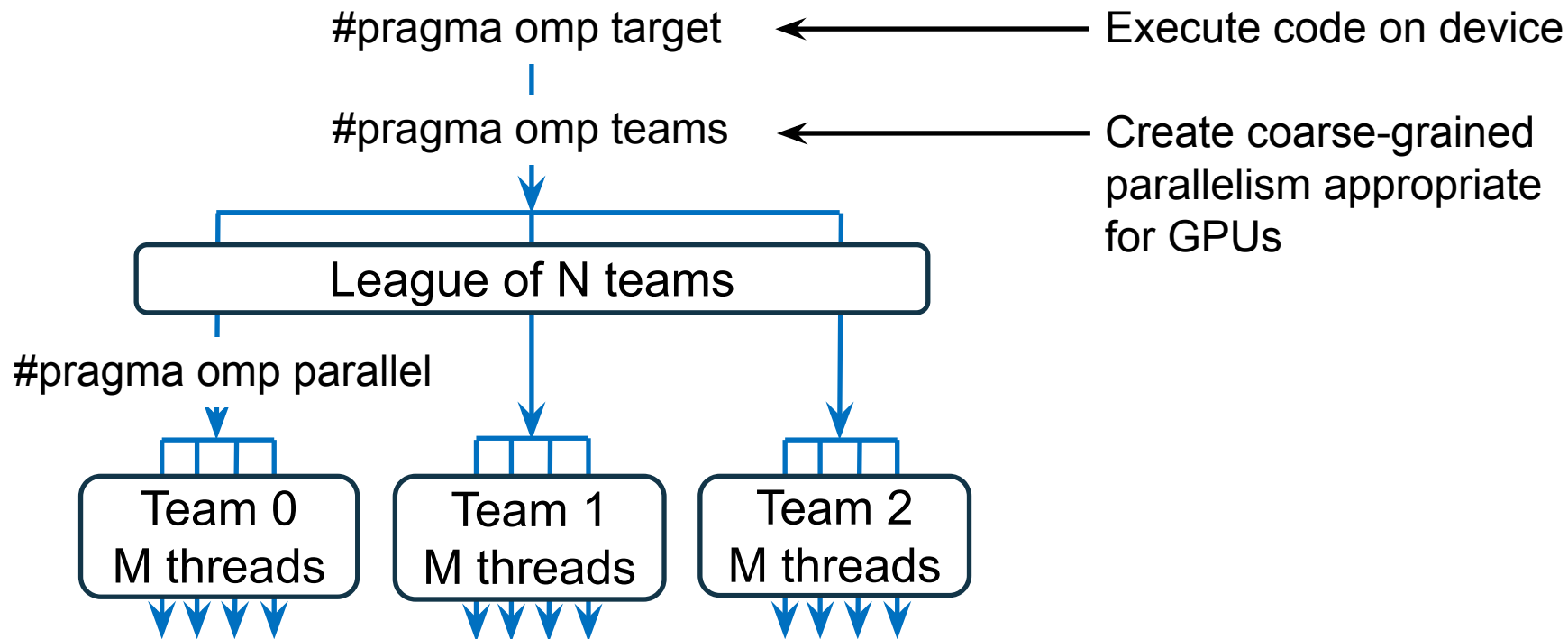
OpenMP for CPUs and GPUs

- OpenMP is a set of directives and APIs to parallelize C, C++ and Fortran applications
- Many NERSC codes use OpenMP on the CPU:

```
#pragma omp parallel for  
for (int i=0; i<N; ++i) x[i] += 1.0;
```

- This presentation will show you how to use OpenMP on the GPU

OpenMP thread hierarchy for GPUs



Comparison to CUDA thread hierarchy

CUDA grid of thread blocks

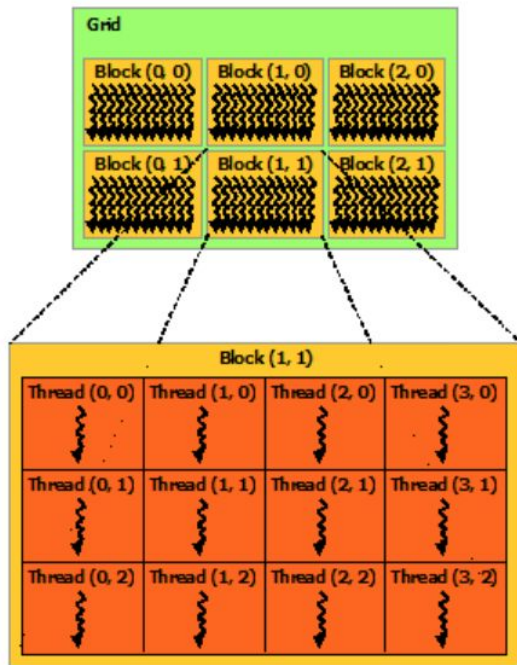


Image from CUDA C++ Programming Guide

1 CUDA thread block = 1 OpenMP team
Workshare with “distribute”

```
#pragma omp teams distribute  
for (int j=0; j<N; ++j)
```

1 CUDA thread = 1 OpenMP thread
Workshare with “for/do”

```
#pragma omp parallel for  
for (int i=0; i<N; ++i)
```

Getting data to/from the GPU

- The CPU and GPU have distinct memory spaces
- OpenMP manages the device data environment using a combination of implicit and explicit data management

Explicitly map the data buffer “x” to/from the GPU using the map clause:

```
printf("%p\n", &x[0]); // CPU: e.g. prints 0x612710
#pragma omp target map(tofrom:x[:N])
{
    printf("%p\n", &x[0]); // GPU: e.g. prints 0x2aaae5afa000
}
```

Executing our simple example on the GPU

```
int N = 16384;
double *x = malloc(N * sizeof(double));
for (int i=0; i<N; ++i) x[i] = 0.0;

#pragma omp target teams distribute parallel for map(tofrom:x[:N])
for (int i=0; i<N; ++i) x[i] += 1.0;
```

Variable	Explanation of variable in device data environment
x	Explicitly mapped variable of length N
N	Firstprivate scalar variable of value 16384
i	Private scalar variable which is uninitialized

Keeping data on the GPU

- The family of target data directives may be used to keep data on the GPU for multiple GPU kernels

```
#pragma omp target enter data map(to:x[:N])

#pragma omp target teams distribute parallel for
for (int i=0; i<N; ++i) x[i] += 1.0; // GPU kernel #1

#pragma omp target teams distribute parallel for
for (int i=0; i<N; ++i) x[i] += 1.0; // GPU kernel #2

#pragma omp target exit data map(from:x[:N])
```

No maps
needed!

Be aware that a map clause does not always cause data movement

- The OpenMP runtime reference counts mapped data
 - This avoids expensive data movement

```
for (int i=0; i<N; ++i) x[i] = 0.0;
#pragma omp target enter data map(to:x[:N])
for (int i=0; i<N; ++i) x[i] = 2.0; // update on host

#pragma omp target map(to:x[:N]) ❌
{
    // Mistake: x[0] != 2.0 on device
}
```

Ensuring consistent data environments: method 1

- The target update directive transfers data between host and device data environments

```
for (int i=0; i<N; ++i) x[i] = 0.0;
#pragma omp target enter data map(to:x[:N])
for (int i=0; i<N; ++i) x[i] = 2.0; // update on host

#pragma omp target update to(x[:N]) ✓
#pragma omp target
{
    // Success: x[0] == 2.0 on device
}
```

Ensuring consistent data environments: method 2

- The always modifier in the map clause transfers data irrespective of the variable reference count

```
for (int i=0; i<N; ++i) x[i] = 0.0;
#pragma omp target enter data map(to:x[:N])
for (int i=0; i<N; ++i) x[i] = 2.0; // update on host

#pragma omp target map(always, to:x[:N]) ✓
{
    // Success: x[0] == 2.0 on device
}
```

OpenMP GPU-offload on Perlmutter

- We recommend the NVIDIA compiler for C, C++ and Fortran OpenMP applications
 - The Clang compiler will be available soon on Perlmutter for C and C++ applications
- Please see “Building and running GPU applications on Perlmutter” slides on Day 1 (Jan 5 2022)
- Also see <https://docs.nersc.gov/performance/readiness/#openmp>

OpenMP “loop” directive for performance

- The “loop” directive workshares loop iterations and also asserts that loop iterations are independent
 - Can provide a performance advantage with the NVIDIA compiler, especially when there are multiple parallel loops

OpenMP-4.5:

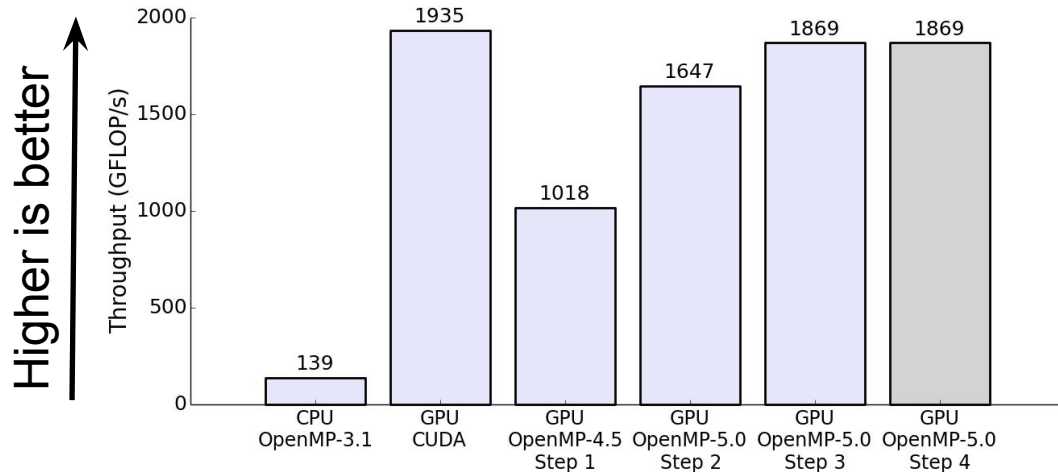
```
#pragma omp teams distribute
for (int j=0; j<N; ++j)
{
    #pragma omp parallel for
    for (int i=0; i<N; ++i)
    {
        x[j+N*i] += 1.0;
    }
}
```

OpenMP-5.0 loop:

```
#pragma omp teams loop
for (int j=0; j<N; ++j)
{
    #pragma omp loop bind(parallel)
    for (int i=0; i<N; ++i)
    {
        x[j+N*i] += 1.0;
    }
}
```

An OpenMP Case Study with SU3 benchmark

- We achieved 97% of CUDA performance on an A100 GPU using OpenMP and the NVIDIA compiler



OpenMP steps:

1. Convert CUDA to OpenMP-4.5
2. Use the “loop” directive
3. Remove “num_teams” and “thread_limit” clauses
4. Simplify by automatically collapsing loops

From “Accelerating Applications for NERSC’s Perlmutter Supercomputer using OpenMP and NVIDIA HPC SDK. GPU Technology Conference (GTC)”, April 13 2021

OpenMP has some advantages over CUDA

- Portable to the CPU and other vendor's GPUs
- Data management is simpler, especially when considering complicated data structures
- Loop iterations can be trivially workshared between threads
- Loops can be trivially fused using the collapse clause
- Data can be reduced over threads trivially using the reduction clause

A quick note about OpenACC

- OpenACC is an alternative directive-based approach
 - Similar directives, occasionally with a different name
- More restrictive programming approach than OpenMP, e.g. no thread ID and no thread synchronizations
 - The OpenMP “loop” directive provides similar restrictions
- Easier to get high performance than OpenMP, however, NERSC/NVIDIA have demonstrated that a suite of NERSC OpenMP applications achieve $\geq 90\%$ of OpenACC performance:

<https://dl.acm.org/doi/10.1145/3458817.3476213>

Questions

Google doc for questions

Check NERSC User slack for relevant channels

#mpi #openmp #kokkos #sycl #fortran and more!

Get it touch with us via help.nersc.gov

Keep an eye out for events focused on specific models and toolchains!

<https://www.nersc.gov/users/training/events/>