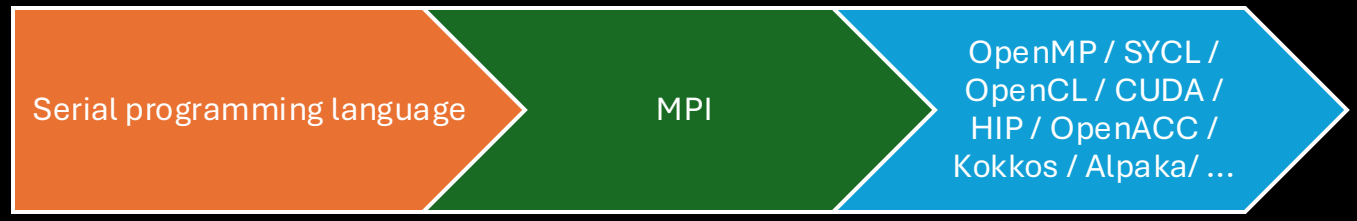
C++ is (nearly) all you need for HPC

Dr Tom Deakin
Senior Lecturer in Advanced Computer Systems
Head of HPC Research Group
University of Bristol, UK

What heterogeneous programming model should I use?

Many options, many pros/cons:

- Are they an open-standard?
- Are they an open-source project?
- Are they cross-vendor?
- Do they support CPUs and GPUs?
- Is the syntax the same for CPUs and GPU?
- …




FORTRAN
FORTRAN
High-Level Language

C++ Features

- mdspan
- algorithm/numeric and execution
- ranges and views
- linalg
- senders/receivers (execution)

- Not mentioned, but useful:
  - atomic (C++11) and atomic_ref (C++20)

mdspan (C++23)

```
template<
    class T,
    class Extents,
    class LayoutPolicy = std::layout_right,
    class AccessorPolicy = std::default_accessor<T>
> class mdspan;
```

Non-owning view of the data (like std::span)

Defines type of data, extents (amount of data in each dimension), layout (data ordering), and accessors (how to access data given a 1D index)

https://en.cppreference.com/w/cpp/container/mdspan.html

```
void init_mat(
  double *A, std::size_t N, std::size_t M) {

    std::mdspan matrix {A, N, M};

    for (std::size_t i = 0; i < matrix.extent(0); ++i)
      for (std::size_t j = 0; j < matrix.extent(1); ++j)
        matrix[i,j] = 1.1;
}
```

Constructs an mdspan view of the data
CTAD helps figure out the class
The extents are dynamic (known at runtime)

```
void init_mat(
  double *A, std::size_t N, std::size_t M) {

    // std::mdspan matrix {A, N, M};
    using exts =
      std::extents<size_t,
        std::dynamic_extent, std::dynamic_extent>;
    std::mdspan<double, exts> matrix {A, N, M};


    for (std::size_t i = 0; i < matrix.extent(0); ++i)
      for (std::size_t j = 0; j < matrix.extent(1); ++j)
        matrix[i,j] = 1.1;
}
```

```
void init_mat(
  double *A, std::size_t N, std::size_t M) {

    std::mdspan matrix {A, N, M};

    for (std::size_t i = 0; i < matrix.extent(0); ++i)
      for (std::size_t j = 0; j < matrix.extent(1); ++j)
        matrix[i,j] = 1.1;
}
```

Query the mdspan structure for the extents in each dimension
Loop order should map layout
layout_right by default i.e., right-most extent is stride 1

```
void init_mat(
  double *A, std::size_t N, std::size_t M) {

    std::mdspan matrix {A, N, M};

    for (std::size_t i = 0; i < matrix.extent(0); ++i)
      for (std::size_t j = 0; j < matrix.extent(1); ++j)
        matrix[i,j] = 1.1;
}
```

Access data using multi-dimensional subscript operator[]

C++20 deprecated using comma operator in subscript operator

C++23 added the multi-dimensional subscript operator, using the comma in natural way

https://en.cppreference.com/w/cpp/language/operator_other.html

algorithm/numeric and execution (C++17)

- Known as “stdpar”, brings execution policies to algorithm/numeric library functions
- Wide range of possible algorithms, the most common for HPC are:
  - std::transform
  - std::transform_reduce
  - std::for_each / std::for_each_n
- Applies functor or lambda function to elements
- Data-centric (container iterators) or index-centric (range iterators)
- Synchronous execution

```
#include <algorithm>
#include <execution>

int N = ...;
double *X, *Y; // initalise these!

// DAXPY: Y += a * X
std::transform(std::execution::par_unseq,
    X, X+N, Y, Y,
    [=](auto &x, auto &y){return x*alpha + y;}
);
```

Transform algorithm with 2 inputs, 1 output
Range of the transform given by the pointer iterators of the first input

```
#include <algorithm>
#include <execution>

int N = ...;
double *X, *Y; // initalise these!

// DAXPY: Y += a * X
std::transform(std::execution::par_unseq,
    X, X+N, Y, Y,
    [=](auto &x, auto &y){return x*alpha + y;}
);
```

Lambda function defines with two arguments containing the data
Capture variables by value (i.e., alpha)
Computes the daxpy kernel on one element of the array
Returns the value for storage in the output iterator

```
#include <algorithm>
#include <execution>

int N = ...;
double *X, *Y; // initalise these!

// DAXPY: Y += a * X
std::transform(std::execution::par_unseq,
    X, X+N, Y, Y,
    [=](auto &x, auto &y){return x*alpha + y;}
);
```

Parallelise with an *execution policy*
Depending on the compiler, this will be run on the CPU or GPU

Execution Policies

Policy	
std::execution::seq	Sequential execution in the calling thread.
std::execution::unseq	SIMD execution in the calling thread. Concurrent iterations can be interleaved.
std::execution::par	Parallel execution in multiple threads. Iterations can be re-ordered, but must run to completion. Allows the use of atomic/atomic_ref/mutex (starvation-free algorithms)
std::execution::par_unseq	Both par and unseq.

Note! The C++ wording for concurrent, parallel, etc is not-aligned with typical use of those words in HPC.

User-responsibility to ensure no data-races and deadlocks.

https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t.html

ranges (C++20)

- Data-centric has limitations (e.g., annoying for stencil codes).
- Instead, really want access to the loop iteration variable
- Ranges give an abstraction over iterators (i.e., has begin() and end())
  - Bring their own version of some algorithms (but can't use execution policy!)
- Views give you a lightweight range
  - Lazy evaluation, O(1) storage
  - Think Python's yield
- Two key routines for HPC:
  - std::views::iota
  - std::views::cartesian_product

Roughly like this

```
void init_mat(
  double *A, std::size_t N, std::size_t M) {

    std::mdspan matrix {A, N, M};

    auto cp = std::views::cartesian_product(
      std::views::iota(0ul, matrix.extent(0)),
      std::views::iota(0ul, matrix.extent(1)),
    );
    std::for_each(std::execution::par_unseq,
        cp.begin(), cp.end(), [=](auto &id){
        auto [i,j] = id;
        matrix[i,j] = 1.1;
    });
}
```

linalg (C++26) [coming soon]

- Brings BLAS (Level 1, Level 2, and Level 3) to C++
- Builds on mdspan data structures
  - Powerful abstractions for transpose, complex conjugate, upper/lower triangular, diagonal
- Builds on algorithms for parallelism with execution policies

https://en.cppreference.com/w/cpp/numeric/linalg.html
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1673r12.html

```
#include <linalg>

std::mdspan A {pA, M, P};
std::mdspan B {pB, P, N};
std::mdspan C {pC, M, N};

// compute C = A * B
std::matrix_product(A, B, C);



// oneMKL equivalent
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
      M, N, P, 1.0, A, P, B, N, 0.0, C, N);
```

```
#include <linalg>

std::mdspan A {pA, M, P};
std::mdspan B {pB, P, N};
std::mdspan C {pC, M, N};

// compute C = A * B
std::matrix_product(std::execution::par_unseq, A, B, C);
```

```
// oneMKL equivalent
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
      M, N, P, 1.0, A, P, B, N, 0.0, C, N);
```

std::execution – senders/receivers (C++26)

- Allows for **asynchronous** execution of a task graph
- Provides a scheduler for where things should run:
  - NB: still in flux what is standard
  - e.g., thread_pool, inline_scheduler
  - Also vendor provided e.g., gpu_scheduler
- Provides senders (the tasks) and mechanisms for programmatically building a graph
  - A bit like C++ futures: returns a value and an error.
  - Describes work to do, to be scheduled later.
  - Can compose them with a pipe | syntax
- Hidden from user code are receivers
  - Handles what senders return (the value and any error)
- In user code, see:
  - Sender adaptors: take one/more sender(s) and return another sender, e.g., **then(), bulk()**
  - Sender factories: entry point for computation, e.g., scheduler()
  - Sender consumers: take a sender and start work, e.g., sync_wait()

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();                    // 1
sender auto begin = schedule(sch);                               // 2
sender auto hi = then(begin, []{                                 // 3
  std::cout << "Hello world! Have an int.";                      // 3
  return 13;                                                     // 3
});                                                              // 3
sender auto add_42 = then(hi, [](int arg) { return arg + 42; }); // 4

auto [i] = this_thread::sync_wait(add_42).value();               // 5
```

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r6.html#example-hello-world

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();                          // 1
sender auto begin = schedule(sch);                                     // 2
sender auto hi = then(begin, []{                                       // 3
  std::cout << "Hello world! Have an int.";                           // 3
  return 13;                                                           // 3
});                                                                    // 3
sender auto add_42 = then(hi, [](int arg) { return arg + 42; });       // 4

auto [i] = this_thread::sync_wait(add_42).value();                     // 5
```

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r6.html#example-hello-world

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();                    // 1
sender auto begin = schedule(sch);                               // 2
sender auto hi = then(begin, []{                                 // 3
  std::cout << "Hello world! Have an int.";                      // 3
  return 13;                                                     // 3
});                                                              // 3
sender auto add_42 = then(hi, [](int arg) { return arg + 42; }); // 4

auto [i] = this_thread::sync_wait(add_42).value();               // 5
```

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r6.html#example-hello-world

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();                    // 1
sender auto begin = schedule(sch);                               // 2
sender auto hi = then(begin, []{                                 // 3
  std::cout << "Hello world! Have an int.";                      // 3
  return 13;                                                     // 3
});                                                              // 3
sender auto add_42 = then(hi, [](int arg) { return arg + 42; }); // 4

auto [i] = this_thread::sync_wait(add_42).value();               // 5
```

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r6.html#example-hello-world

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();                          // 1
sender auto begin = schedule(sch);                                     // 2
sender auto hi = then(begin, []{                                       // 3
  std::cout << "Hello world! Have an int.";                           // 3
  return 13;                                                           // 3
});                                                                    // 3
sender auto add_42 = then(hi, [](int arg) { return arg + 42; });       // 4

auto [i] = this_thread::sync_wait(add_42).value();                     // 5
```

i is set to the result (55)

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2300r6.html#example-hello-world

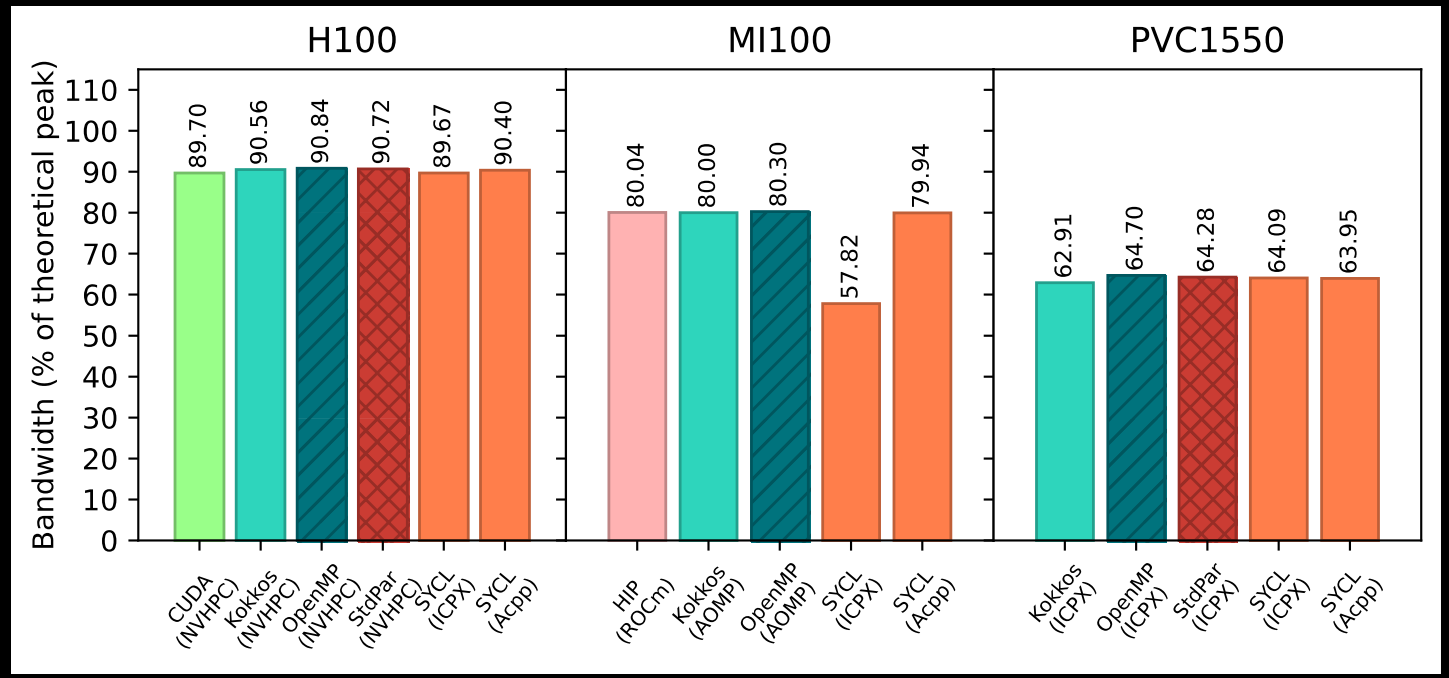
Ecosystem support

- https://github.com/NVIDIA/stdexec
- https://github.com/ericniebler/range-v3
- https://github.com/kokkos/mdspan
- https://github.com/kokkos/stdBLAS

- Support for stdpar (algorithm/numeric) is good:
  - NVIDIA: nvc++ -stdpar=gpu|multicore
  - Intel (par_unseq on GPU): icpx –fsycl –fsycl-pstl-offload=cpu | gpu
  - Clang (CPU): clang++ -stdlib=libc++ -fexperimental-library
  - GNU (CPU): -ltbb
  - Along with AdaptiveCpp, Intel oneDPL, AMD roc-stdpar (all par_unseq only)

Research using C++ Features

Boring graphs are good for performance portability



NB: single two tile device

On latest GPUs from NVIDIA, AMD, and Intel, performance portability for BabelStream possible in most mainstream performance portable programming models:

- ISO C++ stdpar, OpenMP, SYCL, Kokkos
- Same performance as “native” CUDA/HIP

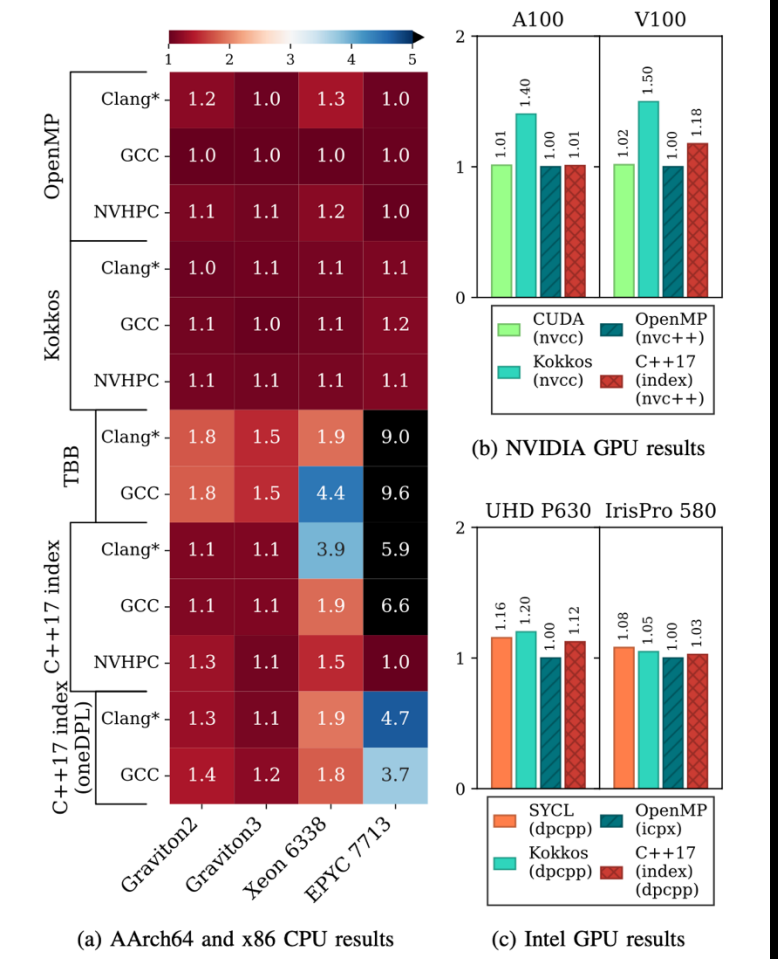


Fig. 5: CloverLeaf results as normalised runtime *per platform*; **lower is better**

More graphs showing comparable performance for CloverLeaf (left, memory bandwidth bound) and miniBUDE (right, compute bound)

Results improved with performance bugs fixed inside oneDPL and NVHPC (see paper)

Just uses algorithms with par_unseq and iota view

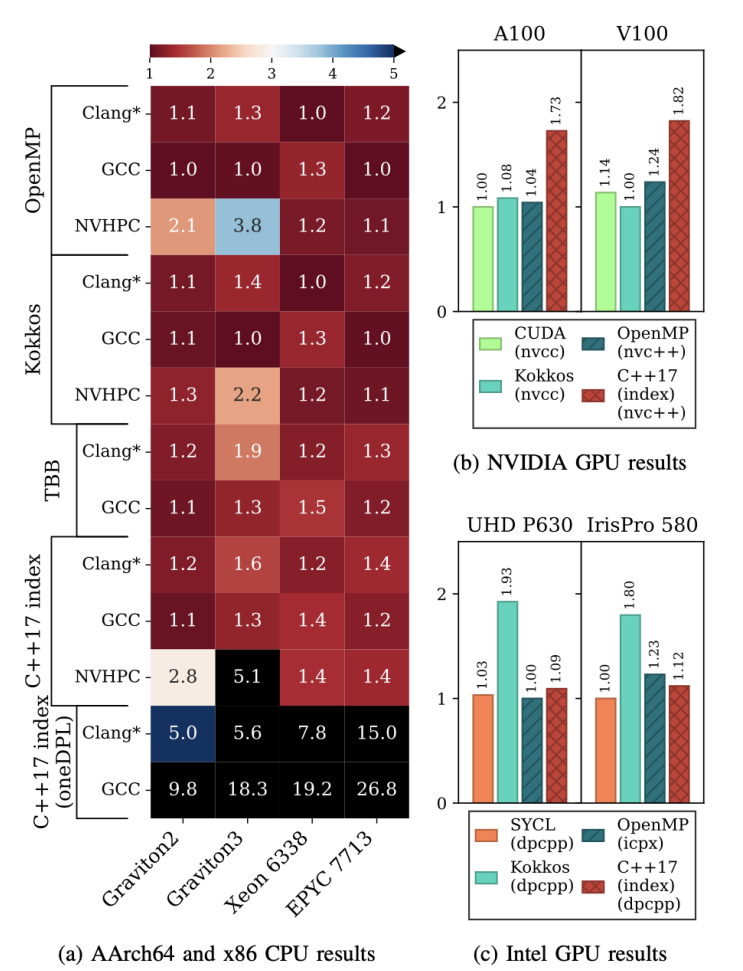
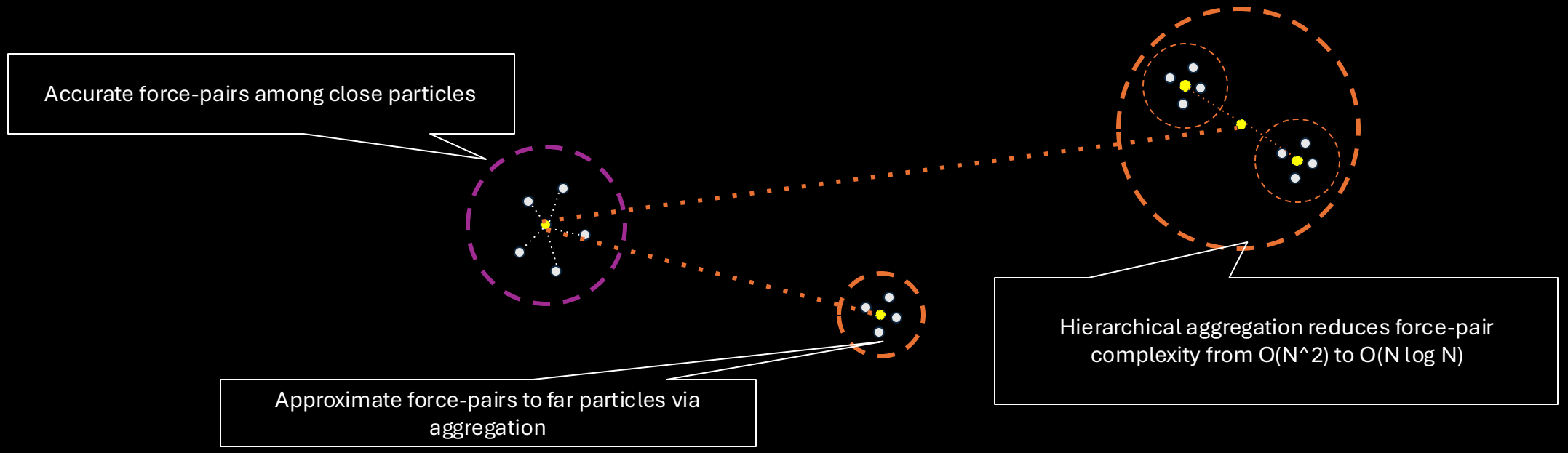


Fig. 3: miniBUDE results as normalised runtime *per platform*; **lower is better**

Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems
Lin, Deakin, and McIntosh-Smith, PMBS, Supercomputing, 2022.
https://doi.org/10.1109/PMBS56514.2022.00009

Barnes-Hut Approximation for N-Body

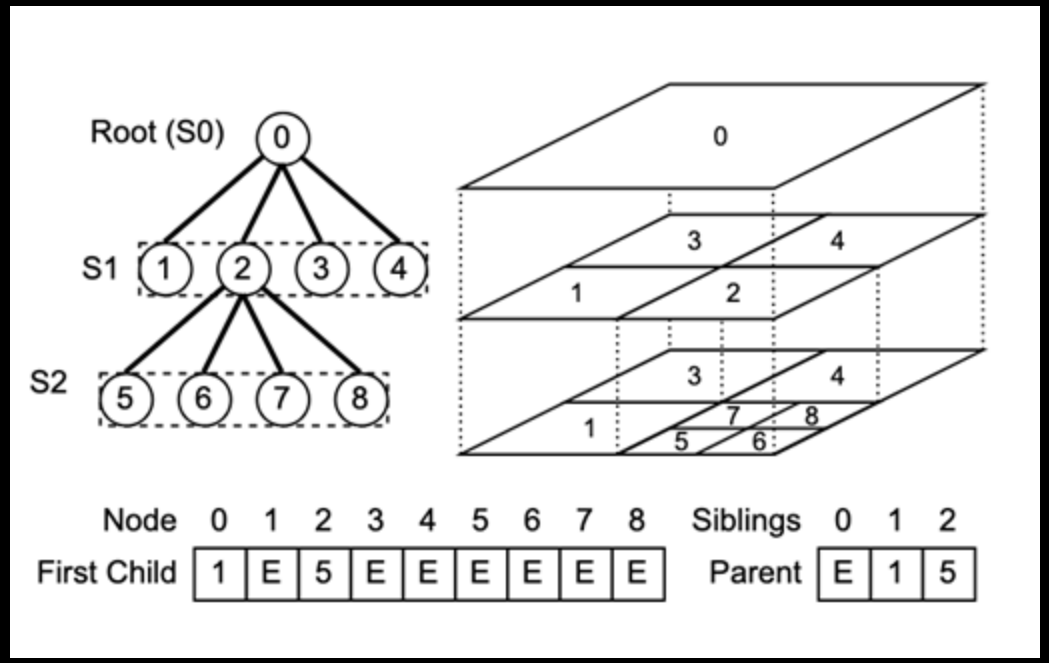


- Introduces *extra work* to aggregate particles and *extra space* (tree data-structure)
- Savings in practice because most execution time spent in Force Calculations

Efficient Tree-based Parallel Algorithms for N-Body Simulations Using C++ Standard Parallelism
Lane Cassell, T., et al, IA^3, Supercomputing, 2024, https://doi.org/10.1109/SCW63240.2024.00099

A C++ stdpar implementation

- Construction of Octree on GPU
  - Threads insert bodies into tree, refining tree if already inserted
  - Uses std::execution::par, std::for_each, atomic operations and locks
- Calculate multipoles (mass centres)
  - Find the leaf nodes by early exit of non-leaf thread, and traverses to root
  - Last child there to there calculates the mass
- Update forces uses par_unseq



Efficient Tree-based Parallel Algorithms for N-Body Simulations Using C++ Standard Parallelism
Lane Cassell, T., et al, IA^3, Supercomputing, 2024, https://doi.org/10.1109/SCW63240.2024.00099

Small Galaxies across devices & algorithms

Higher is better
“Small” 10^5 bodies

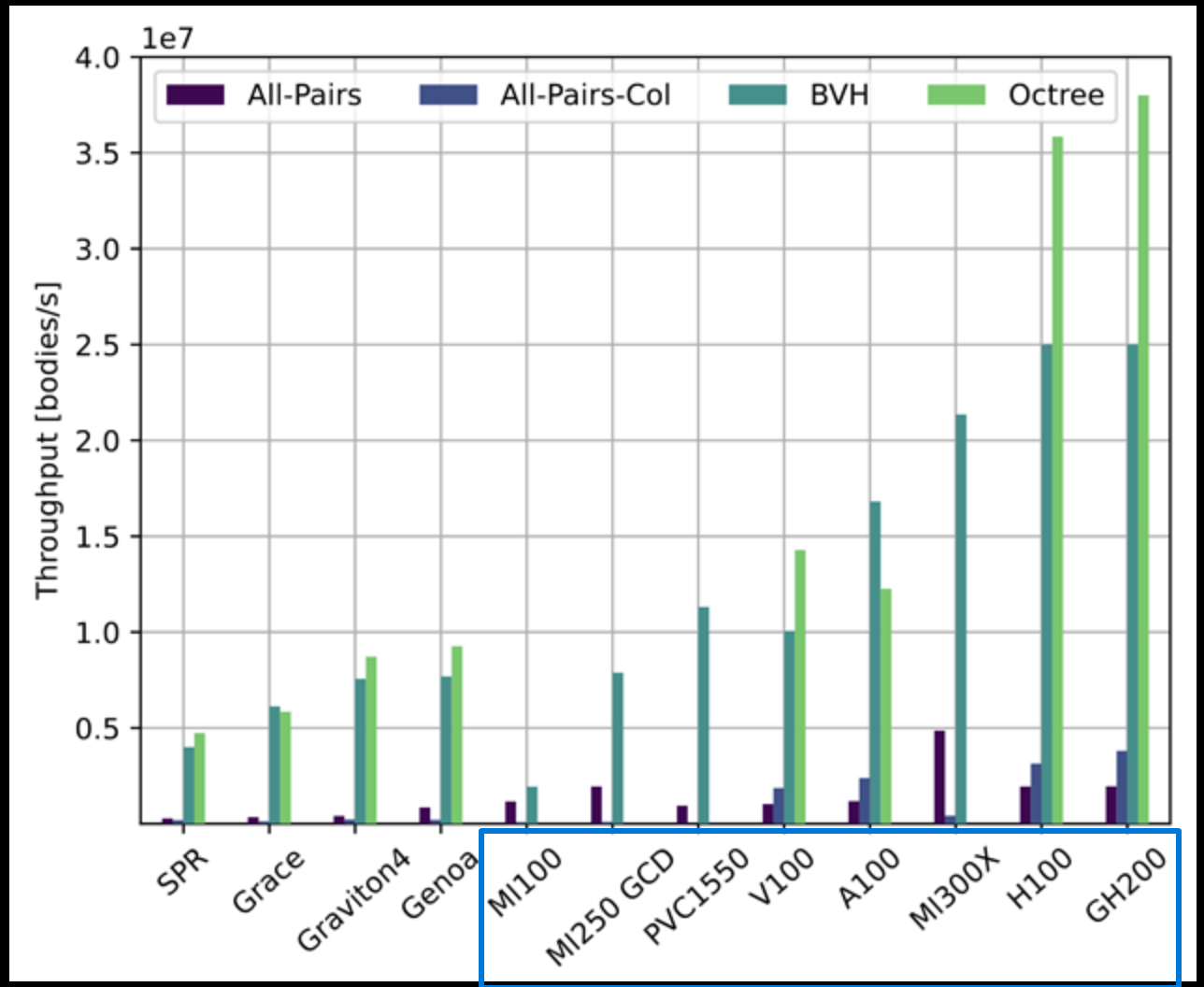
Same code on CPUs and GPUs: Portability

Better algorithms are faster (go figure!)

On CPUs & NVIDIA GPUs: octree ~1.5x faster than LBVH for fixed distance threshold, due to fewer force-pair computations.
(also holds for larger problems, see paper)

Exceptions: A100 L2 cache partitioning increases latency for atomics (used by Octree).

Compared performance against state of the art, Thüring, et al. version which uses **two-phase approach** in SYCL [1], on NASA 1M bodies data-set and H100:
- Octree is 3.3X faster than BVH.
- BVH is 5.3X faster than Thüring, et al. [1]



[1] https://doi.org/10.1145/3624062.3624604

Limitations of C++

And what to do about them

Device discovery and control

Data location and movement in discrete memory spaces

Expressing concurrent and parallel work

Device discovery and control

Data location and movement in discrete memory spaces

Expressing concurrent and parallel work

No introspection,
but control with
std::execution::scheduler

Requires implicit
unified shared
memory, and unlikely
to ever change

execution policies
algorithm
numeric

		C++17 StdPar	SYCL	OpenMP target	HIP/CUDA	OpenCL	Kokkos	Julia
	Code portability	Compiler/ Compiler flags	Compiler/ Compiler flags	Compiler/ Compiler flags	Not portable	Runtime	Compiler/ Compiler flags	Partial* * library dependent
	Device portability	CPU,GPU	CPU,GPU,FPGA	CPU,GPU	GPU (CPU via third-party impl.)	CPU,GPU, FPGA	CPU,GPU	CPU,GPU
	Supported platform	Intel/AMD/NVIDIA	Intel/AMD/NVIDIA	Intel/AMD/NVIDIA	Vendor-only	Intel/AMD/NVIDIA	Intel/AMD/NVIDIA	Intel/AMD/NVIDIA
	Format	Single-source	Single-source	Single-source	Single-source	Multi-source	Single-source	Single-source
"How" data access	Data movement	Implicit: USM	Explicit: accessors Implicit: USM	Explicit: pragmas Implicit: USM	Explicit: vendor API Implicit: USM	Explicit: buffers Implicit: SVM	Explicit: views	Explicit: library API

"How/What" parallel	Traversal	std::for_each
		std::for_each_n
		std::transform
			queue.submit([&](auto &h) {
			h.parallel_for(...);
			});
				# OpenMP >= 5.0
				omp loop
				omp target teams distributed \
				parallel for
					__global__ void kernel(...) {...}
					// ...
					kernel<<<N>>>(...)
						(> 10 lines, in two files)	Kokkos::parallel_for	Yes*
								*library dependent

"How/What" parallel	Reduction	std::transform_reduce
		std::reduce
		std::accumulate
			queue.submit([&](sycl::handler &h) {
			h.parallel_for(
			sycl::reduction(),...)
			});
				omp reduction(inscan,..) {
				omp scan inclusive(...)
				omp scan exclusive(...)
				}
					(> 10 lines, "roll your own")	(> 10 lines, "roll your own")	Kokkos::parallel_reduce	Yes*
								*library dependent

"When" parallel	Task asynchrony/ scheduling	No control (Ongoing proposals for C++26)	Command queues	#pragma omp nowait/depends(...) (Blocking by default)	Streams	Command queues	Futures (C++ like)	Yes* *library dependent
"Where" parallel	Affinity	No control (Ongoing proposals for C+26)	Device API	#pragma omp device(...) (Host is also a device)	Vendor API	Device API	Device API	Yes* *library dependent

From my PhD student: Wei-Chen Lin

More limitations

- Algorithms/numeric and senders/schedulers are different incompatible
  - Several proposals to bring these together (P3300, P2500)
- No reductions for senders
- Not all devices can support std::execution::par
  - CPUs and NVIDIA GPUs can (post Volta in 2017)
  - Other GPUs do not
  - But all can support par_unseq which is still useful

What to do when C++ isn’t enough?

Combine open-standards together in compatible way.

- C++ + SYCL, C++ + OpenMP.

Lots of ongoing work to make sure OpenMP and SYCL can “speak C++”.

Lots of ongoing work inside compilers to make programming models interoperate:

- liboffload library in LLVM
- NVHPC has good interoperability between models.

J. Doerfert, et al, Breaking the Vendor Lock: Performance Portable Programming through OpenMP as Target Independent Runtime Layer, https://doi.org/10.1145/3559009.3569687

References, links and more info

Efficient Tree-based Parallel Algorithms for N-Body Simulations Using C++ Standard Parallelism
Lane Cassell, T., et al, IA^3, Supercomputing, 2024, https://doi.org/10.1109/SCW63240.2024.00099

Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems
Lin, Deakin, and McIntosh-Smith, PMBS, Supercomputing, 2022. https://doi.org/10.1109/PMBS56514.2022.00009

A Metric for HPC Programming Model Productivity
Lin, Deakin, and McIntosh-Smith, P3HPC, Supercomputing, 2024 https://doi.org/10.1109/SCW63240.2024.00160