High Energy Physics

Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

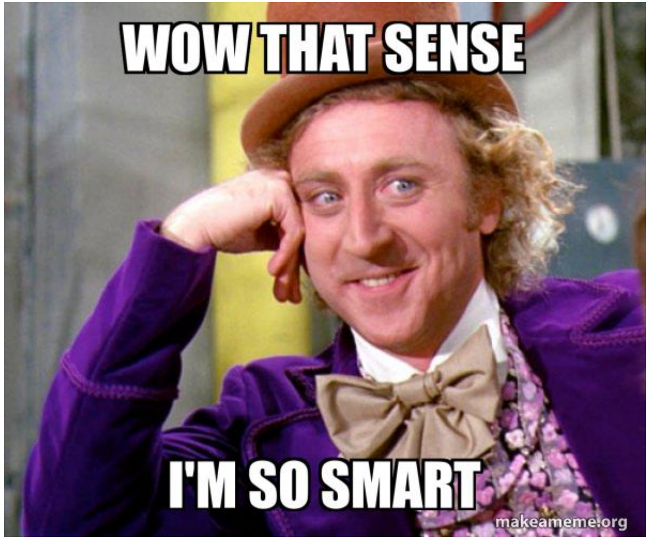
GPU programming module

Week 6 : Advanced topics :
NVIDIA HPC STANDARD LANGUAGE PARALLELISM, C++

Lecture 10 - October 15th 2024

What we learnt last week

- We discussed about CUDA streams, went over the basics of the default and the non-default streams
- We discussed about the differences between pinned and paged memory
- We learnt about CUDA events and the different levels of synchronization between streams



Today

Today we will hear about a slightly different topic:

- How can we run parallel code using C++ standards!

HPC programming in ISO C++

What is High Performance Computing

- High-Performance Computing utilizes supercomputers and parallel processing to handle complex computations

Exascale class supercomputer already used in HEP. Image taken from [i]

What is High Performance Computing

- High-Performance Computing utilizes supercomputers and parallel processing to handle complex computations
  - Crucial for complex simulations or when handling a large amount of data
  - Enables solving complex problems that are infeasible for conventional computers.
  - Critical in fields like scientific research, simulations, and big data analysis.

Exascale class supercomputer already used in HEP.
Image taken from [i]

What is High Performance Computing

- High-Performance Computing utilizes supercomputers and parallel processing to handle complex computations
  - Crucial for complex simulations or when handling a large amount of data
  - Enables solving complex problems that are infeasible for conventional computers.
  - Critical in fields like scientific research, simulations, and big data analysis.
- Modern HPC systems often combine multiple CPUs and GPUs to maximize performance.

Exascale class supercomputer already used in HEP. Image taken from [i]

What is ISO C++

- The C++ language was standardized by ISO in 1998 :
  - Subsequent updates and revisions have happened over the years
  - Ensure that C++ code is portable and consistent across different compilers and platforms.
- Latest C++ standard is C++23

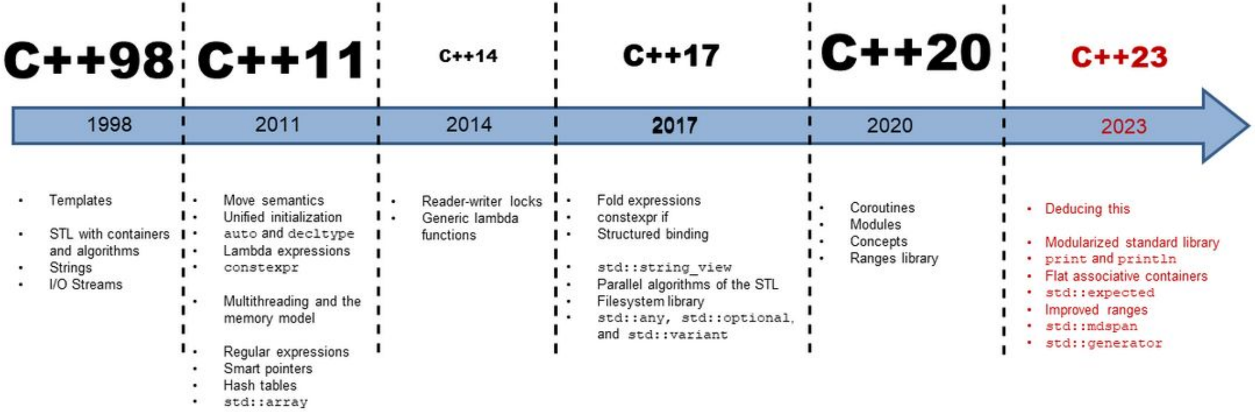
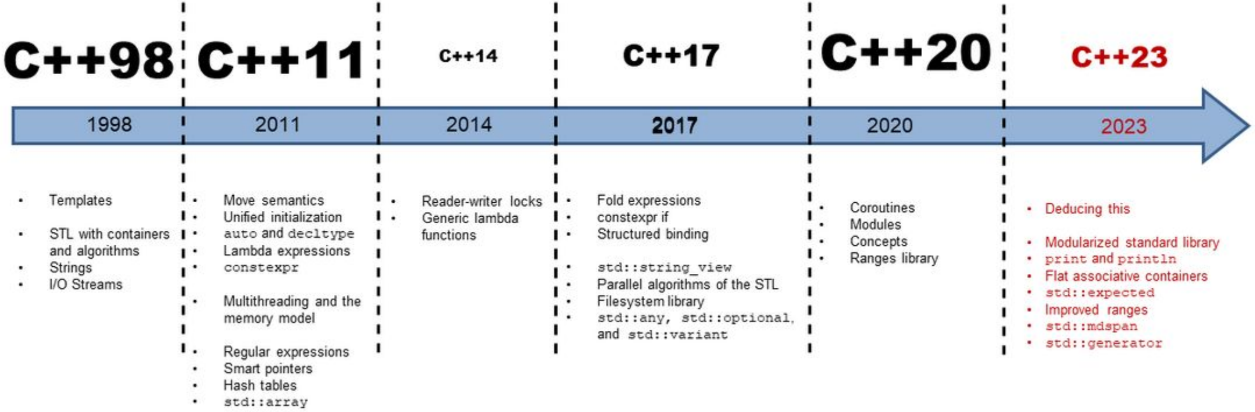


Image taken from [ii]

What is ISO C++

- The C++ language was standardized by ISO in 1998 :
  - Subsequent updates and revisions have happened over the years
  - Ensure that C++ code is portable and consistent across different compilers and platforms.
- Latest C++ standard is C++23



Each new ISO introduces new features and improvements!

Image taken from [ii]

Using ICO C++ for parallel programming

```
std::sort(std::execution::par, c.begin(), c.end());
std::for_each(std::execution::par, c.begin(), c.end(), func);
```

- Introduced in C++ 17
- Parallel and vector concurrency via execution policies
  - std::execution::par
  - std::execution::par_unseq
  - std::execution::seq

Using ICO C++ for parallel programming

```
std::sort(std::execution::par, c.begin(), c.end());
std::for_each(std::execution::par, c.begin(), c.end(), func);
```

- Introduced in C++ 17
- Parallel and vector concurrency via execution policies
  - **std::execution::seq**
  - std::execution::par
  - std::execution::par_unseq

> - This execution policy specifies that the algorithm should be executed sequentially.
> - It behaves like a traditional loop, ensuring that operations are performed in the order they appear.
> - Suitable for small datasets

Using ISO C++ for parallel programming

```
std::sort(std::execution::par, c.begin(), c.end());
std::for_each(std::execution::par, c.begin(), c.end(), func);
```

- Introduced in C++ 17
- Parallel and vector concurrency via execution policies
  - std::execution::seq
  - **std::execution::par**
  - std::execution::par_unseq

- This execution policy allows for parallel execution of the algorithm.
- It may use multiple threads to perform operations concurrently.
- Suitable for larger datasets where operations can be performed independently.
- The order of execution is not guaranteed, meaning results can be produced out of order.

Using ICO C++ for parallel programming

```
std::sort(std::execution::par, c.begin(), c.end());
std::for_each(std::execution::par, c.begin(), c.end(), func);
```

- Introduced in C++ 17
- Parallel and vector concurrency via execution policies
  - std::execution::seq
  - std::execution::par
  - **std::execution::par_unseq**

> - This execution policy allows for both parallel execution and vectorization.
> - It can take advantage of SIMD operations, which can further enhance performance on supported hardware.
> - Best for data that can be processed in parallel without dependency between operations.
> - The order of execution is not guaranteed.

Using ICO C++ for parallel programming

```
std::sort(std::execution::par, c.begin(), c.end());
std::for_each(std::execution::par, c.begin(), c.end(), func);
```

- Introduced in C++ 17
- Parallel and vector concurrency via execution policies
    - std::execution::seq
    - std::execution::par
    - **std::execution::par_unseq**

Offers the highest performance potential among the three policies!

- This execution policy allows for both parallel execution and vectorization.
- It can take advantage of SIMD operations, which can further enhance performance on supported hardware.
- Best for data that can be processed in parallel without dependency between operations.
- The order of execution is not guaranteed.

Using ICO C++ for parallel programming

```
std::sort(std::execution::par, c.begin(), c.end());
std::for_each(std::execution::par, c.begin(), c.end(), func);
```

- Introduced in C++ 17
- Parallel and vector concurrency via execution policies
- NVC++ (since 20.7): automatic CPU or GPU acceleration of C++17 parallel algorithms
  - Levarages CUDA unified memory

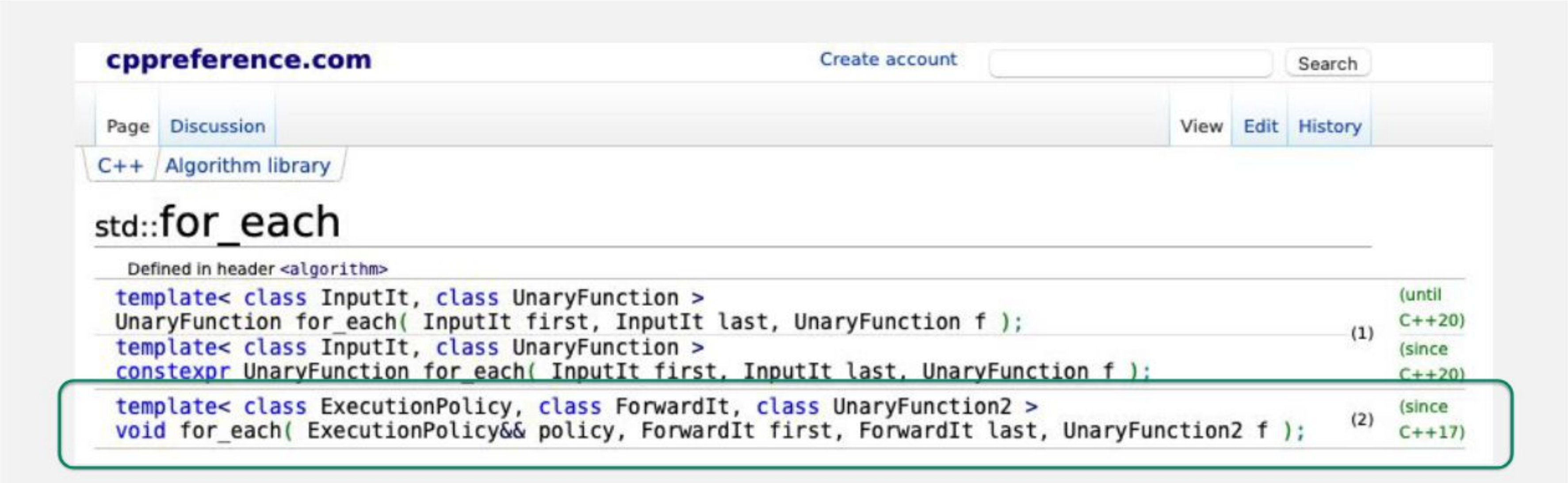
Using ISO C++ for parallel programming

```
std::sort(std::execution::par, c.begin(), c.end());
std::for_each(std::execution::par, c.begin(), c.end(), func);
```

- Introduced in C++ 17
- Parallel and vector conc
- NVC++ (since 20.7): au -17 parallel algorithms
  - Levarages CUDA un

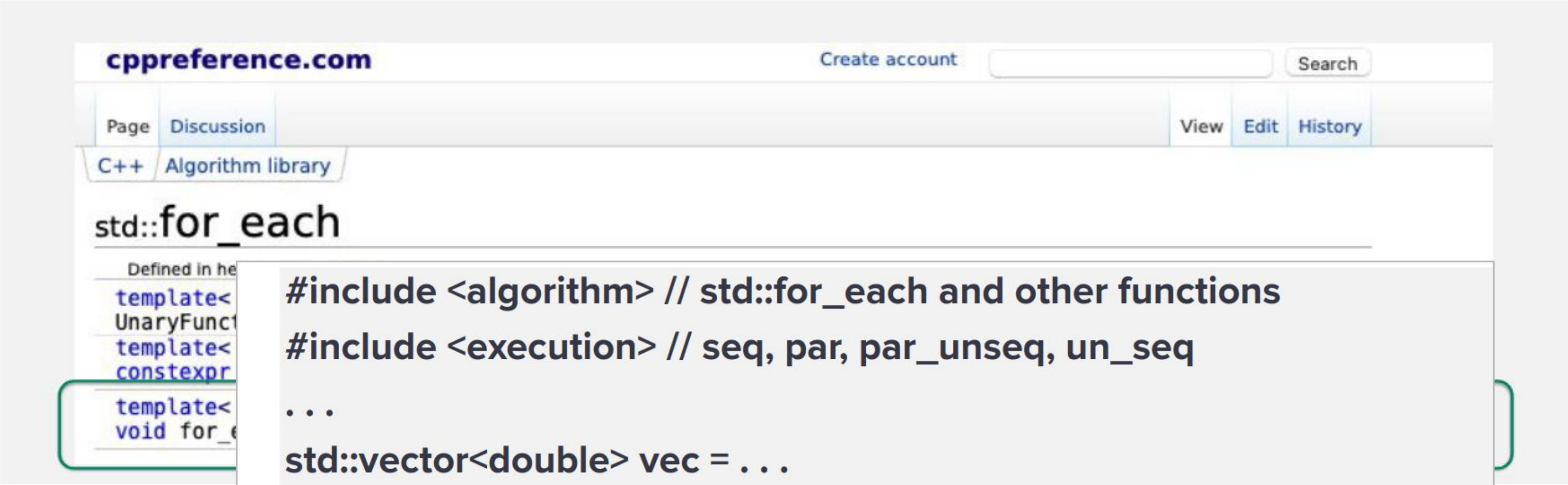
```
Aside: Cuda Unified Memory
float *x, *y;

 // Allocate Unified Memory -- accessible from
CPU or GPU
 cudaMallocManaged(&x, N*sizeof(float));
```

std::for_each



- New overload ExecutionPolicy added to enable parallel execution

std::for_each




```
#include <algorithm> // std::for_each and other functions
#include <execution> // seq, par, par_unseq, un_seq
. . .
std::vector<double> vec = . . .
std::for_each(std::execution::par, vec.begin(), vec.end(), [=](auto i){
        . . . // doing work for each element in the vector
});
```

- New

A simple example

- Include the necessary libraries

```
#include <stdio.h>
#include <vector>
#include <execution>
#include <algorithm>
#include <ranges>

int main(){

   printf("Hello world from main " );
   auto v = std::views::iota(0,9);
   std::for_each(std::execution::par,
v.begin(), v.end(),
    [=](int i) {
        printf("%d,",i);
    });

   printf("\n");
}
```

A simple example

- Include the necessary libraries
- Use the **for_each** loop specifying the execution policy

```
#include <stdio.h>
#include <vector>
#include <execution>
#include <algorithm>
#include <ranges>

int main(){

   printf("Hello world from main " );
   auto v = std::views::iota(0,9);
   std::for_each(std::execution::par,
v.begin(), v.end(),
    [=](int i) {
        printf("%d,",i);
    });

   printf("\n");
}
```

A simple example

- Include the necessary libraries
- Use the **for_each** loop specifying the execution policy
- Function here is a simple lambda but one can use any other function

```
#include <stdio.h>
#include <vector>
#include <execution>
#include <algorithm>
#include <ranges>

int main(){

    printf("Hello world from main " );
    auto v = std::views::iota(0,9);
    std::for_each(std::execution::par,
v.begin(), v.end(),
    [=](int i) {
        printf("%d,",i);
    });

    printf("\n");
}
```

A simple example

- Include the necessary libraries
- Use the **for_each** loop specifying the execution policy
- Function here is a simple lambda but one can use any other function
- To compile :

nvc++ -stdpar=gpu -Minfo=stdpar --std=c++20 test.cpp

```
#include <stdio.h>
#include <vector>
#include <execution>
#include <algorithm>
#include <ranges>

int main(){

   printf("Hello world from main " );
   auto v = std::views::iota(0,9);
   std::for_each(std::execution::par,
v.begin(), v.end(),
    [=](int i) {
        printf("%d,",i);
    });

   printf("\n");
}
```

A simple example

- Include the necessary libraries
- Use the **for_each** loop specifying the execution policy
- Function here is a simple lambda but one can use any other function
- To compile :

nvc++ -stdpar=gpu -Minfo=stdpar --std=c++20 test.cpp

nvc++ is a **C++ compiler** designed to leverage NVIDIA GPUs for high-performance computing applications
More info here

```
#include <stdio.h>
#include <vector>
#include <execution>
#include <algorithm>
#include <ranges>

int main(){

   printf("Hello world from main " );
   auto v = std::views::iota(0,9);
   std::for_each(std::execution::par,
v.begin(), v.end(),
    [=](int i) {
        printf("%d,",i);
    });

   printf("\n");
}
```

A simple example

- Include the necessary libraries
- Use the **for_each** loop specifying the execution policy
- Function here is a simple lambda but one can use any other function
- To compile :

nvc++ **-stdpar=gpu** -Minfo=stdpar --std=c++20 test.cpp

Flag that specifies parallel execution on a GPU.

```
#include <stdio.h>
#include <vector>
#include <execution>
#include <algorithm>
#include <ranges>

int main(){

   printf("Hello world from main " );
   auto v = std::views::iota(0,9);
   std::for_each(std::execution::par,
v.begin(), v.end(),
    [=](int i) {
        printf("%d,",i);
    });

   printf("\n");
}
```

A simple example

- Include the necessary libraries
- Use the **for_each** loop specifying the execution policy
- Function here is a simple lambda but one can use any other function
- To compile :

nvc++ -stdpar=gpu -**Minfo=stdpar** --std=c++20 test.cpp

Flag that instructs the compiler to produce messages that give information about optimization decisions made during compilation

```
#include <stdio.h>
#include <vector>
#include <execution>
#include <algorithm>
#include <ranges>

int main(){

   printf("Hello world from main " );
   auto v = std::views::iota(0,9);
   std::for_each(std::execution::par,
v.begin(), v.end(),
   [=](int i)  {
        printf("%d,",i);
   });

   printf("\n");
}
```

Some considerations

- When using the parallel execution policy, **make sure there are no data races or deadlocks**

Some considerations

- When using the parallel execution policy, **make sure there are no data races or deadlocks**

A data race occurs when two or more threads access the same shared resource simultaneously, and at least one of the accesses is a write operation.

A deadlock is the situation where two or more threads are unable to proceed because each is waiting for the other to release a resource.

Some considerations

- When using the parallel execution policy, **make sure there are no data races or deadlocks**
- stdpar execution on GPU leverages CUDA Unified Memory, data needs to reside in heap memory

Some considerations

- When using the parallel execution policy, **make sure there are no data races or deadlocks**
- stdpar execution on GPU leverages CUDA Unified Memory, data needs to reside in heap memory
- std::vector works but std::array does not

Some considerations

- When using the parallel execution policy, **make sure there are no data races or deadlocks**
- stdpar execution on GPU leverages CUDA Unified Memory, data needs to reside in heap memory
- std::vector works but std::array does not
- Unlike CUDA C++, functions do not need the __device__ annotation

Some considerations

- When using the parallel execution policy, **make sure there are no data races or deadlocks**
- stdpar execution on GPU leverages CUDA Unified Memory, data needs to reside in heap memory
- std::vector works but std::array does not
- Unlike CUDA C++, functions do not need the __device__ annotation
- Execution on GPU requires random access iterators

Some considerations

- When using the parallel execution policy, **make sure there are no data races or deadlocks**
- stdpar execution on GPU leverages CUDA Unified Memory, data needs to reside in heap memory
- std::vector works but std::array does not
- Unlike CUDA C++, functions do not need the __device__ annotation
- Execution on GPU requires random access iterators
- -stdpar currently has two options,
  - -stdpar=gpu (which is the default when not given an option) for parallel execution on GPU
  - –stdpar=multicore for parallel execution on CPU

Tips and tricks: Considering parallel execution

Problem

There is a std::vector I want to sort

```
std::vector<int> vec1;

{0,4,2,9,5,35,7,43,6}
```

Tips and tricks: Considering parallel execution

Problem

There is a std::vector I want to sort

```
std::vector<int> vec1;

{0,4,2,9,5,35,7,43,6}
```

→

Solution

Using standard algorithm std::sort

```
std::sort(vec1.begin(), vec1.end());

{0,2,4,5,6,7,9,35,43}
```

Tips and tricks: Considering parallel execution

Problem		**Solution**
There is a std::vector I want to sort	→	Using standard algorithm std::sort
`std::vector<int> vec1;`		`std::sort(vec1.begin(), vec1.end());`
{0,4,2,9,5,35,7,43,6}		{0,2,4,5,6,7,9,35,43}

But can I parallelize?

Tips and tricks: Considering parallel execution

Problem

There is a std::vector I want to sort

```
std::vector<int> vec1;

{0,4,2,9,5,35,7,43,6}
```

Solution

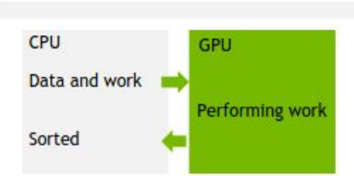
Using standard algorithm std::sort

```
std::sort(vec1.begin(), vec1.end());

{0,2,4,5,6,7,9,35,43}
```

Solution with potential performance improvement

Using parallel execution and –stdpar to offload work and data to GPU

```
std::sort(std::execution::par, vec1.begin(),vec1.end());
nvc++ -stdpar=gpu ./main.cpp

{0,2,4,5,6,7,9,35,43}
```



Tips and tricks: Considering parallel execution

Problem

There is a std::vector I want to sort

std::vector<int> vec1;

{0,4,2,9,5,35,7,43,6}

Solution

Using standard algorithm std::sort

std::sort(vec1.begin(), vec1.end());

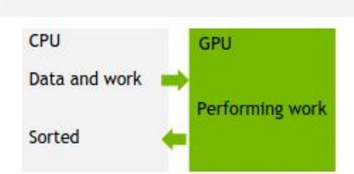
{0,2,4,5,6,7,9,35,43}

Solution with potential performance improvement

Using parallel execution and –stdpar to offload work and data to GPU

std::sort(std::execution::par, vec1.begin(),vec1.end());
nvc++ -stdpar=gpu ./main.cpp

{0,2,4,5,6,7,9,35,43}



Something to keep in mind!
Not all problems do benefit from parallelizing. Keep in mind that there is an overhead for data transfers to and from the GPU

Tips and tricks: Using -Minfo for compile time info

We already got introduced to some of the compiler flag options of nvc++ :

nvc++ -stdpar=gpu -Minfo=stdpar --std=c++20 test.cpp

The output would look something like :

main:

13, stdpar: Generating NVIDIA GPU code

13, std::for_each with std::execution::par_unseq policy parallelized on GPU

<u>The messages can include useful information :</u>

- about vectorization
- loop transformations
- how the compiler decides to parallelize certain operations

Tips and tricks: Use std::Views::iota for easy iterator

We already got introduced to some of the compiler flag options of nvc++ :

nvc++ -stdpar=gpu -Minfo=stdpar --std=c++20 test.cpp

The output would look something like :

main:

13, stdpar: Generating NVIDIA GPU code

13, std::for_each with std::execution::par_unseq policy parallelized on GPU

The messages can include useful information :

- about vectorization
- loop transformations
- how the compiler decides to parallelize certain operations

- Available in C++20
- Introduces low-level operations that are faster than manually incrementing values in a loop
- Generates values on the fly
- More info in cpp reference

```
auto v = std::views::iota(0, 9);
std::for_each(std::execution::par_unseq, v.begin(), v.end(),
[=](int i){
        printf("%d, ", threadIdx.x);
        printf("%d, ", blockIdx.x);
})
```

Note: Can access cuda specific variables if running on the GPU

Wrapping-up

Overview of today's lecture

- C++ has been introducing ISOs standards over the past 25 years which ensure consistency & portability across different compilers and platforms

- Today we went over some of the new features of C++ HPC ISO standards

- We can achieve parallel and vector concurrency via execution policies
    - nvc++ is the C++ compiler used which is provided by NVIDIA

- Careful evaluation of whether our algorithm would benefit from parallelization is still needed since we still have the overheads of data-transfers

Thursday

- We will hear a lot about CUDA managed memory by a guest lecturer from Fermilab!

Back-up

Resources

1. NVIDIA Deep Learning Institute material link
2. 10th Thematic CERN School of Computing material link
3. Nvidia turing architecture white paper link
4. CUDA programming guide link
5. CUDA runtime API documentation link
6. CUDA profiler user's guide link
7. CUDA/C++ best practices guide link
8. NVidia DLI teaching kit link
9. https://tac-hep.org/assets/pdf/uw-gpu-fpga/2023_Stdpar_Cpp.pdf
10. Cpp reference https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t