



NVIDIA[®]

HPC Compilers

C++ PARALLEL ALGORITHMS

TABLE OF CONTENTS

Chapter 1. Introduction.....	1
Chapter 2. NVC++ Compiler Parallel Algorithms Support.....	3
2.1. Enabling Parallel Algorithms with the -stdpar option.....	3
Chapter 3. Simple Example.....	4
Chapter 4. Coding guidelines for GPU-accelerating Parallel Algorithms.....	5
4.1. Parallel Algorithms and device function annotations.....	5
4.2. Parallel Algorithms and CUDA Unified Memory.....	5
4.3. Parallel Algorithms and function pointers.....	6
4.4. Random access iterators.....	8
4.5. Interoperability with the C++ Standard Library.....	8
4.6. No exceptions in GPU code.....	8
Chapter 5. Larger example: LULESH.....	9
Chapter 6. Getting started with Parallel Algorithms for GPUs.....	11
6.1. Supported NVIDIA GPUs.....	11
6.2. Supported CUDA versions.....	12

Chapter 1.

INTRODUCTION

The C++17 Standard introduced higher-level parallelism features that allow users to request parallelization of Standard Library algorithms.

This higher-level parallelism is expressed by adding an execution policy as the first parameter to any algorithm that supports execution policies. Most of the existing Standard C++ algorithms were enhanced to support execution policies. C++17 defined several new parallel algorithms, including the useful `std::reduce` and `std::transform_reduce`.

C++17 defines three [execution policies](#):

- ▶ `std::execution::seq`: Sequential execution. No parallelism is allowed.
- ▶ `std::execution::par`: Parallel execution on one or more threads.
- ▶ `std::execution::par_unseq`: Parallel execution on one or more threads, with each thread possibly vectorized.

When you use an execution policy other than `std::execution::seq`, you are communicating two important things to the compiler:

- ▶ You prefer but do not require that the algorithm be run in parallel. A conforming C++17 implementation may ignore the hint and run the algorithm sequentially, but a high-quality implementation takes the hint and executes in parallel when possible and prudent.
- ▶ The algorithm is safe to run in parallel. For the `std::execution::par` and `std::execution::par_unseq` policies, any user-provided code—such as iterators, lambdas, or function objects passed into the algorithm—must not introduce data races if run concurrently on separate threads. For the `std::execution::par_unseq` policy, any user-provided code must not introduce data races or deadlocks if multiple calls are interleaved on the same thread, which is what happens when a loop is vectorized. For more information about potential deadlocks, see the [forward progress guarantees](#) provided by the parallel policies or watch [CppCon 2018: Bryce Adelstein Lelbach “The C++ Execution Model”](#).

The C++ Standard grants compilers great freedom to choose if, when, and how to execute algorithms in parallel as long as the forward progress guarantees the user requests are honored. For example, `std::execution::par_unseq` may be implemented with vectorization and `std::execution::par` may be implemented with

a CPU thread pool. It is also possible to execute parallel algorithms on a GPU, which is a good choice for invocations with sufficient parallelism to take advantage of the processing power and memory bandwidth of NVIDIA GPU processors.

Chapter 2.

NVC++ COMPILER PARALLEL ALGORITHMS SUPPORT

The NVIDIA HPC C++ compiler, NVC++, supports C++17, C++ Standard Parallelism (stdpar), OpenACC and OpenMP for NVIDIA GPUs and multicore CPUs.

NVC++ can compile Standard C++ algorithms with the parallel execution policies `std::execution::par` or `std::execution::par_unseq` for execution on NVIDIA GPUs and multicore CPUs. An NVC++ command-line option, `-stdpar`, is used to enable GPU-accelerated C++ Parallel Algorithms. Lambdas, including generic lambdas, are fully supported in parallel algorithm invocations. No language extensions or non-standard libraries are required to enable GPU acceleration. All data movement between host memory and GPU device memory is performed implicitly and automatically under the control of CUDA Unified Memory.

It's easy to automatically GPU accelerate C++ Parallel Algorithms with NVC++. However, there are some restrictions and limitations you need to be aware of as explained below.

2.1. Enabling Parallel Algorithms with the `-stdpar` option

GPU acceleration of C++ Parallel Algorithms is enabled with the `-stdpar=gpu` command-line option to NVC++. If `-stdpar=gpu` is specified (or `-stdpar` without an argument), almost all algorithms that use a parallel execution policy are compiled for offloading to run in parallel on an NVIDIA GPU:

```
nvc++ -stdpar=gpu program.cpp -o program
```

```
nvc++ -stdpar program.cpp -o program
```

Acceleration of C++ Parallel Algorithms with multicore CPUs is enabled with the `-stdpar=multicore` command-line option to NVC++. If `-stdpar=multicore` is specified, all algorithms that use a parallel execution policy are compiled to run in on a multicore CPU:

```
nvc++ -stdpar=multicore program.cpp -o program
```

Chapter 3.

SIMPLE EXAMPLE

Here are a few simple examples to get a feel for how the C++ Parallel Algorithms work.

From the early days of C++, sorting items stored in an appropriate container has been relatively easy using a single call like the following:

```
std::sort(employees.begin(), employees.end(),
          CompareByLastName());
```

Assuming the comparison class **CompareByLastName** is thread-safe, which is true for most comparison functions, parallelizing this sort is simple with C++ Parallel Algorithms. Include **<execution>** and add an execution policy to the function call:

```
std::sort(std::execution::par,
          employees.begin(), employees.end(),
          CompareByLastName());
```

Calculating the sum of all the elements in a container is also simple with the **std::accumulate** algorithm. Prior to C++17, transforming the data in some way while taking the sum was somewhat awkward. For example, to compute the average age of your employees, you might write the following code:

```
int ave_age =
    std::accumulate(employees.begin(), employees.end(), 0,
                    [](int sum, const Employee& emp){
                        return sum + emp.age();
                    })
    / employees.size();
```

The **std::transform_reduce** algorithm introduced in C++17 makes it simple to parallelize this code. It also results in cleaner code by separating the reduction operation, in this case **std::plus**, from the transformation operation, in this case **emp.age()**:

```
int ave_age =
    std::transform_reduce(std::execution::par_unseq,
                          employees.begin(), employees.end(),
                          0, std::plus<int>(),
                          [](const Employee& emp){
                              return emp.age();
                          })
    / employees.size();
```

Chapter 4.

CODING GUIDELINES FOR GPU-ACCELERATING PARALLEL ALGORITHMS

GPUs are not simply CPUs with more threads. To effectively take advantage of the massive parallelism and memory bandwidth available on GPUs, it is typical for GPU programming models to put some limitations on code executed on the GPU. The NVC++ implementation of C++ Parallel Algorithms is no exception in this regard. The sections which follow detail the limitations that apply in the current release.

4.1. Parallel Algorithms and device function annotations

Functions to be executed on the GPU within parallel algorithms do not need any `__device__` annotations or other special markings to be compiled for GPU execution. The NVC++ compiler walks the call graph for each source file and automatically infers which functions must be compiled for GPU execution.

However, this only works when the compiler can see the function definition in the same source file where the function is called. This is true for most inline functions and template functions but may fail when functions are defined in a different source file or linked in from an external library. You need to be aware of this when formulating parallel algorithms invocations that you expect to be offloaded and accelerated on NVIDIA GPUs.

4.2. Parallel Algorithms and CUDA Unified Memory

NVC++ relies on CUDA Unified Memory for all data movement between CPU and GPU memory. Through support in both the CUDA device driver and the NVIDIA GPU hardware, the CUDA Unified Memory manager automatically moves some types of data based on usage.

Currently, only data dynamically allocated on the heap in CPU code that was compiled by NVC++ can be managed automatically. Memory dynamically allocated in GPU code

is only visible from GPU code and can never be accessed by the CPU. Likewise, CPU and GPU stack memory and memory used for global objects on most systems cannot be automatically managed. Likewise, data that is dynamically allocated in program units not compiled by NVC++ with the `-stdpar` option is not automatically managed by CUDA Unified Memory, even though it is on the CPU heap.

As a result, any pointer that is dereferenced and any C++ object that is referenced within a Parallel Algorithm invocation must refer to data on the CPU heap that is allocated in a program unit compiled by NVC++. Dereferencing a pointer to a CPU stack or a global object results in a memory violation in GPU code.

For example, `std::vector` uses dynamically allocated memory, which is accessible from the GPU when using `stdpar`. Iterating over the contents of a `std::vector` in a Parallel Algorithm works as expected:

```
std::vector<int> v = ...;
std::sort(std::execution::par,
          v.begin(), v.end()); // Okay, accesses heap memory.
```

On the other hand, `std::array` performs no dynamic allocations. Its contents are stored within the `std::array` object itself, which is often on a CPU stack. Iterating over the contents of a `std::array` won't work unless the `std::array` itself is allocated on the heap:

```
std::array<int, 1024> a = ...;
std::sort(std::execution::par,
          a.begin(), a.end()); // Fails, array is on a CPU stack.
```

Pay particular attention to lambda captures, especially capturing data objects by reference, which may contain non-obvious pointer dereferences:

```
void saxpy(float* x, float* y, int N, float a) {
    std::transform(std::execution::par_unseq, x, x + N, y, y,
                  [&](float xi, float yi){ return a * xi + yi; });
}
```

In the earlier example, the function parameter `a` is captured by reference. The code within the body of the lambda, which is running on the GPU, tries to access `a`, which is in the CPU stack memory. This results in a memory violation and undefined behavior. In this case, the problem can easily be fixed by changing the lambda to capture by value:

```
void saxpy(float* x, float* y, int N, float a) {
    std::transform(std::execution::par_unseq, x, x + N, y, y,
                  [=](float xi, float yi){ return a * xi + yi; });
}
```

With this one-character change, the lambda makes a copy of `a`, which is then copied to the GPU, and there are no attempts to reference CPU stack memory from GPU code.

4.3. Parallel Algorithms and function pointers

Functions compiled to run on either the CPU or the GPU must be compiled into two different versions, one with the CPU machine instructions and one with the GPU machine instructions.

In the current implementation, a function pointer either points to the CPU or the GPU version of the functions. This causes problems if you attempt to pass function pointers between CPU and GPU code. You might inadvertently pass a pointer to the CPU version

of the function to GPU code. In the future, it may be possible to automatically and seamlessly support the use of function pointers across CPU and GPU code boundaries, but it is not supported in the current implementation.

Function pointers can't be passed to Parallel Algorithms to be run on the GPU, and functions may not be called through a function pointer within GPU code. For example, the following code example won't work correctly:

```
void square(int& x) { x = x * x; }
void square_all(std::vector<int>& v) {
    std::for_each(std::execution::par_unseq,
                  v.begin(), v.end(), &square);
}
```

It passes a pointer to the CPU version of the function `square` to a parallel `for_each` algorithm invocation. When the algorithm is parallelized and offloaded to the GPU, the program fails to resolve the function pointer to the GPU version of `square`.

You can often solve this issue by using a function object, which is an object with a function call operator. The function object's call operator is resolved at compile time to the GPU version of the function, instead of being resolved at run time to the incorrect CPU version of the function as in the previous example. For example, the following code example works:

```
struct squared {
    void operator()(int& x) const { x = x * x; }
};
void square_all(std::vector<int>& v) {
    std::for_each(std::execution::par_unseq,
                  v.begin(), v.end(), squared{});
}
```

Another possible workaround is to change the function to a lambda, because a lambda is implemented as a nameless function object:

```
void square_all(std::vector<int>& v) {
    std::for_each(std::execution::par_unseq, v.begin(), v.end(),
                  [](int& x) { x = x * x; });
}
```

If the function in question is too big to be converted to a function object or a lambda, then it should be possible to wrap the call to the function in a lambda:

```
void compute(int& x) {
    // Assume lots and lots of code here.
}
void compute_all(std::vector<int>& v) {
    std::for_each(std::execution::par_unseq, v.begin(), v.end(),
                  [](int& x) { compute(x); });
}
```

No function pointers are used in this example.

The restriction on calling a function through a function pointer unfortunately means passing polymorphic objects from CPU code to GPU-accelerated Parallel Algorithms is not currently supported, as virtual tables are implemented using function pointers.

4.4. Random access iterators

The C++ Standard requires that the iterators passed to most C++ Parallel Algorithms be forward iterators. However, C++ Parallel Algorithms on GPUs only works with random access iterators. Passing a forward iterator or a bidirectional iterator to a GPU/CPU-accelerated Parallel Algorithm results in a compilation error. Passing raw pointers that point to the heap or Standard Library random access iterators to the algorithms has the best performance, but most other random-access iterators work correctly.

4.5. Interoperability with the C++ Standard Library

Large parts of the C++ Standard Library can be used with `stdpar` on GPUs.

- ▶ `std::atomic<T>` objects within GPU code work provided that `T` is a four-byte or eight-byte integer type. `std::atomic<T>` objects can be accessed from both CPU and GPU code provided the object is on the heap.
- ▶ Math functions that operate on floating-point types—such as `sin`, `cos`, `log`, and most of the other functions declared in `<cmath>`—can be used in GPU code and resolve to the same implementations that are used in CUDA C++ programs.
- ▶ `std::complex`, `std::tuple`, `std::pair`, `std::optional`, `std::variant`, and `<type_traits>`, are supported and work as expected in GPU code.

The parts of the C++ Standard Library that aren't supported in GPU code include I/O functions and in general any function that accesses the CPU operating system. As a special case, basic `printf` calls can be used within GPU code and leverage the same implementation that is used in NVIDIA CUDA C++.

4.6. No exceptions in GPU code

As with most other GPU programming models, throwing and catching C++ exceptions is not supported within Parallel Algorithm invocations that are offloaded to the GPU.

Unlike some other GPU programming models where try/catch blocks and throw expressions are compilation errors, exception code does compile but with non-standard behavior. Catch clauses are ignored, and throw expressions abort the GPU kernel if actually executed. Exceptions in CPU code work without restrictions.

Chapter 5.

LARGER EXAMPLE: LULESH

The **LULESH hydrodynamics mini-app** was developed at Lawrence Livermore National Laboratory to stress test compilers and model performance of hydrodynamics applications. It is about 9,000 lines of C++ code, of which 2,800 lines are the core computation that should be parallelized.

We ported LULESH to C++ Parallel Algorithms and made the port available on **LULESH's GitHub repository**. To compile it, install the **NVIDIA HPC SDK**, check out the 2.0.2-dev branch of the LULESH repository, go to the correct directory, and **run make**.

```
git clone --branch 2.0.2-dev https://github.com/LLNL/LULESH.git
cd LULESH/stdpar/build
make run
```

While LULESH is too large to show the entire source code here, there are some key code sequences that demonstrate the use of `stdpar`.

The LULESH code has many loops with large bodies and no loop-carried dependencies, making them good candidates for parallelization. Most of these were easily converted into calls to `std::for_each_n` with the `std::execution::par` policy, where the body of the lambda passed to `std::for_each_n` is identical to the original loop body.

The function **CalcMonotonicQRegionForElems** is an example of this. The loop header written for OpenMP looks as follows:

```
#pragma omp parallel for firstprivate(qlc_monoq, qqc_monoq, \
                               monoq_limiter_mult, monoq_max_slope, ptiny)
for ( Index_t i = 0 ; i < domain.regElemSize(r); ++i ) {
```

This loop header in the C++ Parallel Algorithms version becomes **the following**:

```
std::for_each_n(
    std::execution::par, counting_iterator(0), domain.regElemSize(r),
    [=, &domain](Index_t i) {
```

The loop body, which in this case is almost 200 lines long, becomes the body of the lambda but is otherwise unchanged from the OpenMP version.

In a number of places, an explicit **for** loop was changed to use C++ Parallel Algorithms that better express the intent of the code, such as the function **CalcPressureForElems**:

```
#pragma omp parallel for firstprivate(length)
for (Index_t i = 0; i < length ; ++i) {
```

```
Real_t cls = Real_t(2.0)/Real_t(3.0) ;  
bvc[i] = cls * (compression[i] + Real_t(1.));  
pbvc[i] = cls;  
}
```

This function was rewritten as follows:

```
constexpr Real_t cls = Real_t(2.0) / Real_t(3.0);  
std::transform(std::execution::par,  
    compression, compression + length, bvc,  
    [=](Real_t compression_i) {  
        return cls * (compression_i + Real_t(1.0));  
    });  
std::fill(std::execution::par, pbvc, pbvc + length, cls);
```

Chapter 6.

GETTING STARTED WITH PARALLEL ALGORITHMS FOR GPUS

To get started, download and install the [NVIDIA HPC SDK](#) on your x86-64, OpenPOWER, or Arm CPU-based system running a supported version of Linux.

The NVIDIA HPC SDK is freely downloadable and includes a perpetual use license for all NVIDIA Registered Developers, including access to future release updates as they are issued. After you have the NVIDIA HPC SDK installed on your system, the `nvc++` compiler is available under the `/opt/nvidia/hpc_sdk` directory structure.

- ▶ To use the compilers including `nvc++` on a Linux/x86-64 system, add the directory `/opt/nvidia/hpc_sdk/Linux_x86_64/23.5/compilers/bin` to your path.
- ▶ On an OpenPOWER or Arm CPU-based system, replace `Linux_x86_64` with `Linux_ppc64le` or `Linux_aarch64`, respectively.

6.1. Supported NVIDIA GPUs

The NVC++ compiler can automatically offload C++ Parallel Algorithms to NVIDIA GPUs based on the Volta, Turing, or Ampere architectures. These architectures include features -- such as independent thread scheduling and hardware optimizations for CUDA Unified Memory -- that were specifically designed to support high-performance, general-purpose parallel programming models like the C++ Parallel Algorithms.

The NVC++ compiler provides limited support for C++ Parallel Algorithms on the Pascal architecture, which does not have the [independent thread scheduling](#) necessary to properly support the `std::execution::par` policy. When compiling for the Pascal architecture (`-gpu=cc60`), NVC++ compiles algorithms with the `std::execution::par` policy for serial execution on the CPU. Only algorithms with the `std::execution::par_unseq` policy will be scheduled to run on Pascal GPUs.

By default, NVC++ auto-detects and generates GPU code for the type of GPU that is installed on the system on which the compiler is running. To generate code for a specific GPU architecture, which may be necessary when the application is compiled and run on different systems, add the `-gpu=ccXX` command-line option.

6.2. Supported CUDA versions

The NVC++ compiler is built on CUDA libraries and technologies and uses CUDA to accelerate C++ Parallel Algorithms on NVIDIA GPUs. A GPU-accelerated system on which NVC++-compiled applications are to be run must have a CUDA 11.0 or newer device driver installed.

The NVIDIA HPC SDK compilers ship with an integrated CUDA toolchain, header files, and libraries to use during compilation, so it is not necessary to have a CUDA Toolkit installed on the system.

When `-stdpar` is specified, NVC++ compiles using the CUDA toolchain version that matches the CUDA driver installed on the system on which compilation is performed. To compile using a different version of the CUDA toolchain, use the `-gpu=cudaX.Y` option. For example, use the `-gpu=cuda11.0` option to specify that your program should be compiled for a CUDA 11.0 system using the CUDA 11.0 toolchain.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, CUDA-X, GPUDirect, HPC SDK, NGC, NVIDIA Volta, NVIDIA DGX, NVIDIA Nsight, NVLink, NVSwitch, and Tesla are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2023 NVIDIA Corporation. All rights reserved.

