Accelerating Standard C++ with GPUs Using stdpar | NVIDIA Developer Blog

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

David Olsen Fri Jun, 11 03:53

Historically, accelerating your C++ code with GPUs has not been possible in Standard C++ without using language extensions or additional libraries:

- CUDA C++ requires the use of `__host__` and `__device__` attributes on functions and the triple-chevron syntax for GPU kernel launches.
- OpenACC uses #pragmas to control GPU acceleration.
- Thrust lets you express parallelism portably but uses language extensions internally and only supports a limited number of CPU and GPU backends. The portability of the application is limited by the portability of the library.

In many cases, the results of these ports are worth the effort. But what if you could get the same effect without that cost? What if you could take your Standard C++ code and accelerate on a GPU?

Now you can! NVIDIA recently announced NVC++, the NVIDIA HPC SDK C++ compiler. This is the first compiler to support GPU-accelerated Standard C++ with no language extensions, pragmas, directives, or non-standard libraries. You can write Standard C++, which is portable to other compilers and systems, and use NVC++ to automatically accelerate it with high-performance NVIDIA GPUs. We built it so that you can spend less time porting and more time on what really matters—solving the world's problems with computational science.

C++ Standard Parallelism

C++11 introduced a memory model, concurrent execution model, and concurrency library, providing a standard way to take advantage of multicore processors. However, until recently, Standard C++ lacked higher-level facilities for parallel programming.

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++ defines four execution policies:

- `std::execution::seq` : Sequential execution. No parallelism is allowed.
- `std::execution::unseq` : Vectorized execution on the calling thread (this execution policy was added in C++20).
- `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 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::unseq` and `std::execution::par_unseq` policies, 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 that the user requests are honored. For example, `std::execution::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 modern GPU processors.

NVIDIA HPC SDK

The NVIDIA HPC SDK is a comprehensive suite of compilers, libraries, and tools used to GPU accelerate HPC modeling and simulation applications. With support for NVIDIA GPUs and x86-64, OpenPOWER, or Arm CPUs running Linux, the NVIDIA HPC SDK provides proven tools and technologies for building cross-platform, performance-portable, and scalable HPC applications.

The NVIDIA HPC SDK includes the new NVIDIA HPC C++ compiler, NVC++. NVC++ supports C++17, C++ Standard Parallelism (stdpar) for CPU and GPU, OpenACC for CPU and GPU, and OpenMP for CPU.

NVC++ can compile Standard C++ algorithms with the parallel execution policies `std::execution::par` or `std::execution::par_unseq` for execution on NVIDIA GPUs. 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 use GPU acceleration for C++ Parallel Algorithms with NVC++. However, there are some restrictions and limitations, which we explain later in this post.

Enabling C++ Parallel Algorithms with the -stdpar option

GPU acceleration of C++ Parallel Algorithms is enabled with the `-stdpar` command-line option to NVC++. If `-stdpar` is specified, almost all algorithms that use a parallel execution policy are compiled for offloading to run in parallel on an NVIDIA GPU:

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

Simple examples

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 such as the following:

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

Assuming that the comparison class `CompareByLastName` is thread-safe, which is true for most comparison functions, then 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 example:

```
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();
```

Coding guidelines for GPU-accelerating C++ Parallel Algorithms

GPUs are not simply CPUs with more threads. To take advantage of the massive parallelism available on GPUs, it is typical for GPU programming models to put some limitations on code to be executed on the GPU.

The NVC++ implementation of C++ Parallel Algorithms is no exception in this regard. In this post, we list the major limitations that apply at the time of publication. As always, NVIDIA is working across both hardware and software teams to eliminate as many of these restrictions as possible for future releases.

Topics covered in this section include:

- C++ Parallel Algorithms and device function annotations
- C++ Parallel Algorithms and CUDA Unified Memory
- C++ Parallel Algorithms and function pointers
- Random-access iterators

- Interoperability with the C++ Standard Library
- No exceptions in GPU code

C++ Parallel Algorithms and device function annotations

Unlike CUDA C++, functions 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.

C++ Parallel Algorithms and CUDA Unified Memory

Most GPU programming models allow or require that movement of data objects between CPU memory and GPU memory be managed explicitly by the user. For example, CUDA provides `cudaMemcpy` and OpenACC has `#pragma acc` data directives to control the movement of data between CPU memory and GPU 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. Dynamic allocations from CPU code that was 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 object that is referenced within a C++ Parallel Algorithm invocation must refer to the CPU heap. 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 `std::vector` in a C++ 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 `std::array` won't work unless the `std::array` object 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.

C++ 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 support the use of function pointers automatically and seamlessly across CPU and GPU code boundaries, but it is not supported in the current implementation.

Function pointers can't be passed to C++ 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 C++ Parallel Algorithms is not currently supported, as virtual tables are implemented using function pointers.

Random-access iterators

The C++ Standard requires that the iterators passed to most C++ Parallel Algorithms be forward iterators. However, stdpar on GPUs only works with random-access iterators. Passing a forward iterator or a bidirectional iterator to a GPU-accelerated C++ 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.

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 CUDA C++.

No exceptions in GPU code

As with most other GPU programming models, throwing and catching C++ exceptions is not supported within C++ 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 executed. Exceptions in CPU code work without restrictions.

Larger example: LULESH

The LULESH hydrodynamics mini-app was developed at Lawrence Livermore National Laboratory to stress test compilers and to model the expected performance of their full-sized production hydrodynamics applications. It is about 9,000 lines of C++ code, of which 2,800 lines are the core computation that should be parallelized. The app has been ported to many different parallel programming models, including MPI, OpenMP, OpenACC, CUDA C++, RAJA, and Kokkos.

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 in this post, here are some key sections 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 like the following code example:

```
#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 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.

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 c1s = Real_t(2.0)/Real_t(3.0) ;
  bvc[i] = c1s * (compression[i] + Real_t(1.));
  pbvc[i] = c1s;
}
```

This function was rewritten as the following:

```
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);
```

There are a few cases where the improvements in code clarity, and therefore productivity, are dramatic. The function `CalcHydroConstraintForElems` finds the minimum value from a set of computed values.

The OpenMP version of the function is 52 lines long, with a temporary array to hold the minimum value found by each thread. This can be written in Standard C++ and run in parallel on the GPU with a single call to `std::transform_reduce`, letting the compiler take care of all the complications of doing a parallel reduction. This change shrinks the previous 52 lines into the following 12 lines:

```
dthydro = std::transform_reduce(std::execution::par,
  counting_iterator(0), counting_iterator(length),
  dthydro, [](Real_t a, Real_t b) { return a < b ? a : b; },
  [=, &domain](Index_t i) {
     Index_t indx = regElemlist[i];
     if (domain.vdov(indx) == Real_t(0.0)) {
      return std::numeric_limits<Real_t>::max();
     } else {
      return dvovmax /
             (std::abs(domain.vdov(indx)) + Real_t(1.e-20));
     }
  });
```

When the C++ Parallel Algorithm version of LULESH is compiled for a single A100 GPU, the application runs almost seven times faster than when the same code is compiled to run on all 40 CPU cores on a dual-socket Skylake system, as shown in Figure 1. The performance of the parallel algorithm version on A100 is almost identical to the OpenACC version on A100.

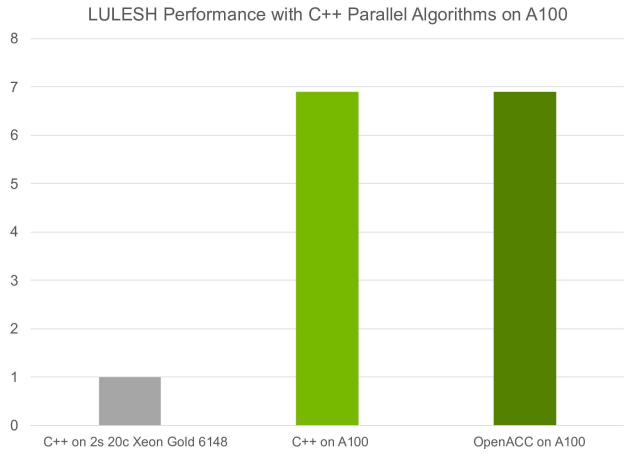

Figure 1. Relative performance of different implementations of LULESH. The left column is the Standard C++ Parallel Algorithms version run on all the CPU cores of a 40-core dual-socket Skylake system. The other two columns were run on a single A100 GPU. The middle column is the Standard C++ Parallel Algorithms version, the same code as the left column. The right column is the OpenACC version of LULESH.

Getting started with C++ 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_64/20.5/compilers/bin` to your path.
- On an OpenPOWER or Arm CPU-based system, replace `Linux_x86_64` with `Linux_ppc64le` or `Linux_aarch64`, respectively.

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 the CPU. Only algorithms with the `std::execution::par_unseq` policy can 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=cc`*`XX`* command-line option. Currently, the compiler can generate executables targeted for only one GPU architecture. The use of multiple `-gpu=cc`*`XX`* options in a single compilation results in an error from the compiler.

Supported CUDA versions

The NVC++ compiler is built on CUDA libraries and technologies and uses CUDA to accelerate C++ Parallel Algorithms to NVIDIA GPUs. A GPU-accelerated system on which NVC++-compiled applications are to be run must have a CUDA 10.1 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 the 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=cuda`*`X.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 11.0 toolchain.

Productivity, performance, and portability

The recently announced NVC++ compiler included in the NVIDIA HPC SDK enables you, for the first time, to program NVIDIA GPUs using completely standard and fully portable C++ constructs. C++ Parallel Algorithm invocations instrumented with appropriate execution policies are automatically parallelized and offloaded to NVIDIA GPUs. The resulting programming environment increases GPU programmer productivity, provides an easy on-ramp to GPU computing for new users, and delivers performance portability across a wide variety of Standard C++ implementations.