(https://developer.nvidia.com/) ACCELERATED COMPUTING (https://developer.nvidia.com/accelerated-computing)

Downloads (https://developer.nvidia.com/accelerated-computing-toolkit)

Training (https://developer.nvidia.com/accelerated-computing-training) Ecosystem (https://developer.nvidia.com/tools-ecosystem)

Forums (https://devtalk.nvidia.com)

Register Now (https://developer.nvidia.com/accelerated-computing-developer) Login (https://developer.nvidia.com/user)

← Previous (https://devblogs.nvidia.com/parallelforall/power-cpp11-cuda-7/) Next → (https://devblogs.nvidia.com/parallelforall/cudnn-v2-higher-performance-deep-learning-gpus/)

C++11 in CUDA: Variadic Templates

Share:
(https://devblogs.nvidia.com/parallelforall/cplusplus-11-in-cuda-variadic-templates/)

Posted on **March 26, 2015 (https://devblogs.nvidia.com/parallelforall/cplusplus-11-in-cuda-variadic-templates/)** by **Mark Harris (https://devblogs.nvidia.com/parallelforall/author/mharris/)** | **6 Comments (https://devblogs.nvidia.com/parallelforall/cplusplus-11-in-cuda-variadic-templates/#disqus_thread)**
Tagged **C++ (https://devblogs.nvidia.com/parallelforall/tag/c/)**, **C++11 (https://devblogs.nvidia.com/parallelforall/tag/c11/)**, **CUDA (https://devblogs.nvidia.com/parallelforall/tag/cuda/)**, **CUDA 7 (https://devblogs.nvidia.com/parallelforall/tag/cuda-7/)**

CUDA 7 adds C++11 feature support to nvcc, the CUDA C++ compiler. This means that you can use C++11 features not only in your host code compiled with `nvcc`, but also in device code. In my post "The Power of C++11 in CUDA 7 (http://devblogs.nvidia.com/parallelforall/power-cpp11-cuda-7/)" I covered some of the major new features of C++11, such as lambda functions, range-based for loops, and automatic type deduction (`auto`). In this post, I'll cover variadic templates (http://en.cppreference.com/w/cpp/language/parameter_pack).

There are times when you need to write functions that take a variable number of arguments: variadic functions. To do this in a typesafe manner for polymorphic functions, you really need to take a variable number of types in a template. Before C++11, the only way to write variadic functions was with the ellipsis (`...`) syntax and the `va_*` facilities (http://en.cppreference.com/w/cpp/utility/variadic). These facilities did not enable type safety and can be difficult to use.

As an example, let's say we want to abstract the launching of GPU kernels. In my case, I want to provide simpler launch semantics in the Hemi (http://devblogs.nvidia.com/parallelforall/developing-portable-cuda-cc-code-hemi/) library. There are many cases where you don't care to specify the number and size of thread blocks—you just want to run a kernel with "enough" threads to fully utilize the GPU, or to cover your data size. In that case we can let the library decide how to launch the kernel, simplifying our code. But to launch arbitrary kernels, we have to support arbitrary type signatures. Well, we can do that like this:

```
template <typename... Arguments>
void cudaLaunch(const ExecutionPolicy &p,
                void(*f)(Arguments...),
                Arguments... args);
```

Here, `Arguments...` is a "type template parameter pack". We can use it to refer to the type signature of our kernel function pointer `f`, and to the arguments of `cudaLaunch`. To do the same thing before C++11 (and CUDA 7) required providing multiple implementations of `cudaLaunch`, one for each number of arguments we wanted to support. That meant you had to limit the maximum number of arguments allowed, as well as the amount of code you had to maintain. In my experience this was prone to bugs. Here's the implementation of `cudaLaunch`.

```
// Generic simplified kernel launcher
// configureGrid uses the CUDA Occupancy API to choose grid/block dimensions
template <typename... Arguments>
void cudaLaunch(const ExecutionPolicy &policy,
                void (*f)(Arguments...),
                Arguments... args)
{
    ExecutionPolicy p = policy;
    checkCuda(configureGrid(p, f));
    f<<<p.getGridSize(), p.getBlockSize(), p.getSharedMemBytes()>>>(args...);
}

// and a wrapper for default policy -- i.e. automatic execution configuration
template <typename... Arguments>
void cudaLaunch(void(*f)(Arguments... args), Arguments... args)
{
    cudaLaunch(ExecutionPolicy(), f, args...);
}
```

Here you can see how we access the types of the arguments (`Arguments...`) in the definition our variadic template function, in order to specify the type signature of the kernel function pointer `*f`. Inside the function, we unpack the parameters using `args...` and pass them to our kernel function when we launch it. C++11 also lets you query the number of parameters in a pack using `sizeof...()`.

Using `hemi::cudaLaunch`, I can launch any `__global__` kernel, regardless of how many parameters it has, like this (here I'm launching my `xyzw_frequency` kernel from my post The Power of C++11 in CUDA 7 (http://devblogs.nvidia.com/parallelforall/power-cpp11-cuda-7/).

```
hemi::cudaLaunch(xyzw_frequency, count, text, int n);
```

Here we leave the launch configuration up to the runtime, and if we write our kernel in a portable way (http://devblogs.nvidia.com/parallelforall/developing-portable-cuda-cc-code-hemi/), this code can be made fully portable. This simplified launch code is currently available in a development branch of Hemi, which you can find on Github (https://github.com/harrism/hemi/tree/apk).

Variadic Kernels

Of course, you can also define kernel functions and `__device__` functions with variadic arguments. I'll finish up with a little program that demonstrates a few things. The `__global__` function `Kernel` is a variadic template function which just forwards its parameter pack to the function `adder`, which is where the really interesting use of variadic templates happens. (I borrowed the `adder` example from an excellent post on variadic templates (http://eli.thegreenplace.net/2014/variadic-templates-in-c/) by Eli Bendersky.)

`adder` demonstrates how a variadic parameter pack can be unpacked recursively to operate on each parameter in turn. Note that to terminate the recursion we define the "base case" function `template <typename T> adder(T v);`, so that when the parameter pack is just a single parameter it just returns its value. The second `adder` function unpacks one argument at a time because it is defined to take one parameter and then a parameter pack. Clever trick, and since all the recursion happens at compile time, the resulting code is very efficient.

We define a utility template function `print_it` with various specializations that print the type of an argument and its value. We launch the kernel with four different lists of arguments. Each time, we vary the type of the first argument to demonstrate how our variadic adder can handle multiple types, and the output has a different type each time. Note another C++11 feature is used here: `static_assert` and type traits. Our `adder` only works with integral and floating point types, so we check the types at compile time using `static_assert` to check if an arithmetic type is used. This allows us to print a custom error message at compile time when the function is misused.

```
#include <type_traits>
#include <stdio.h>

template<typename T>
__host__ __device__
T adder(T v) {
  return v;
}

template<typename T, typename... Args>
__host__ __device__
T adder(T first, Args... args) {
  static_assert(std::is_arithmetic<T>::value, "Only arithmetic types supported");
  return first + adder(args...);
}

template<typename T>
__host__ __device__
void print_it(T x) { printf("Unsupported type\n"); }

template<>
__host__ __device__
void print_it(int x) { printf("int %d\n", x); }
template<>
__host__ __device__
```

You can compile this code with `nvcc --std=c++11 variadic.cu -o variadic`.

Note that in CUDA 7, A variadic `__global__` function template has the following (documented (http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cpp11)) restrictions:

- Only a single pack parameter is allowed.
- The pack parameter must be listed last in the template parameter list.

In practice I don't find these limitations too constraining.

Try CUDA 7 Today

The CUDA Toolkit version 7 is available now, so download it today (http://developer.nvidia.com/cuda-toolkit) and try out the C++11 support and other new features (http://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/).

RELATED POSTS

The Power of C++11 in CUDA 7 (https://devblogs.nvidia.com/parallelforall/power-cpp11-cuda-7/)

Simple, Portable Parallel C++ with Hemi 2 and CUDA 7.5 (https://devblogs.nvidia.com/parallelforall/simple-portable-parallel-c-hemi-2/)

CUDA 7 Release Candidate Feature Overview: C++11, New Libraries, and More (https://devblogs.nvidia.com/parallelforall/cuda-7-release-candidate-feature-overview/)

Accelerating Bioinformatics with NVBIO (https://devblogs.nvidia.com/parallelforall/accelerating-bioinformatics-nvbio/)

‖ ∀

Share:

About Mark Harris

Mark is Chief Technologist for GPU Computing Software at NVIDIA. Mark has fifteen years of experience developing software for GPUs, ranging from graphics and games, to physically-based simulation, to parallel algorithms and high-performance computing. Mark has been using GPUs for general-purpose computing since before they even supported floating point arithmetic. While a Ph.D. student at UNC he recognized this nascent trend and coined a name for it: GPGPU (General-Purpose computing on Graphics Processing Units), and started GPGPU.org to provide a forum for those working in the field to share and discuss their work.

Follow @harrism on Twitter (https://twitter.com/intent/user?screen_name=harrism)
View all posts by Mark Harris → (https://devblogs.nvidia.com/parallelforall/author/mharris/)

6 Comments **Parallel Forall** 1 **Login**

Recommend Share Sort by Best

Join the discussion…

Sergi • 2 years ago
Useful examples Mark! I had written a C pre-processor variadic macro for something similar as your first cudaLaunch example. It works fine. Variadic Kernels look very useful. I believe this could help simplify heterogenous programming and portability. Let's read quite a few more times your posting ... It seems the clearest exposition about it ... One question: is the CUDA toolkit updated with what you tell us? More examples? After a quick search, I only found E.2.9.8. __global__ functions and function templates and PTX ISA stuff.
1 • Reply • Share ›

Mark Harris Mod Sergi • 2 years ago
"is the CUDA toolkit updated with what you tell us"? Not sure what you mean. As the post says this is supported in CUDA 7, and the documentation link in the post links to the restrictions on variadic __global__ function template parameters. So I think the answer is "yes".
• Reply • Share ›

Peter V./Vienna/Austria/Europe • 2 years ago
How does the compiler resolve lambda functions in CUDA 7.0? Are they inline or is it really a function call?
I need this information in order to estimate the performance. When all lambda functions are resolved as real function calls then this will affect very negative the performance.
• Reply • Share ›

Mark Harris Mod Peter V./Vienna/Austria/Europe • 2 years ago
Hi Peter,
The compiler uses the same strategy to inline lambdas that it uses for any other function call. Where the lambda is called directly the inliner will try to inline as usual. But if the lambda is stored into an instance of nvstd::function (from the "nvfunctional" header), and the compiler is unable to figure out the underlying function at the call site, it will not be inlined.
• Reply • Share ›

raphasch • 2 years ago
I would like to also mention perfect forwarding of arguments, as no discussion of function templates in C++11 is complete without it:
http://eli.thegreenplace.ne...
• Reply • Share ›

Devid • 2 years ago
How about OpenCL 2.1 support ?
• Reply • Share ›

ALSO ON **PARALLEL FORALL**

Recursive Neural Networks with PyTorch
3 comments • a month ago•
Avataraldub wedding — Come to the GPU Technology Conference, May 8-11 in San Jose, California, to learn more …

NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge
5 comments • 2 months ago•
AvatarMahesh Khadtare — Wow, Nice Article, Thanks!

Deep Learning in a Nutshell: Reinforcement Learning
3 comments • 8 months ago•
AvatarLiza Loop — Thanks for the comments, Carl. I don't think the data we need exists yet in useable form because a) we …

Image Segmentation Using DIGITS 5
33 comments • 6 months ago•
AvatarAlexander Kindziora — Thank you for this great article! Is there any chance to get your pretrained model with …

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

ACCELERATED COMPUTING	GET STARTED	LEARN MORE	GET INVOLVED
	About CUDA	Training and Courseware	Forums (https://devtalk.nvidia.com/)

(HTTPS://DEVELOPER.NVIDIA.COM
/ACCELERATED-COMPUTING)

GAMEWORKS
(HTTPS://DEVELOPER.NVIDIA.COM
/GAMEWORKS)

EMBEDDED COMPUTING
(HTTPS://DEVELOPER.NVIDIA.COM
/EMBEDDED-COMPUTING)

DESIGNWORKS
(HTTPS://DEVELOPER.NVIDIA.COM
/DESIGNWORKS)

(https://developer.nvidia.com
/about-cuda)

Parallel Computing
(https://developer.nvidia.com
/accelerated-computing-training)

CUDA Toolkit
(https://developer.nvidia.com/cuda-
toolkit)

CUDACast (http://www.youtube.com
/playlist?list=PL5B692fm6--vScfBaxgY89IRWFzDt0Khm)

(https://developer.nvidia.com/cuda-
education-training)

Tools and Ecosystem
(https://developer.nvidia.com/tools-
ecosystem)

Academic Collaboration
(https://developer.nvidia.com
/academia)

Documentation (http://docs.nvidia.com
/cuda/index.html)

Parallel Forall Blog
(https://devblogs.nvidia.com
/parallelforall/)

Developer Program
(https://developer.nvidia.com/cuda-
registered-developer-program)

Contact Us
(https://developer.nvidia.com/contact)