A C++ template for decoupling the invocation of CUDA kernels from the nvcc compiler driver

Sola Aina

✉ `sola.aina@mail.com` `@_SolaAina_`

September 3, 2018

Ecrebo Limited, Reading UK

About me/this talk

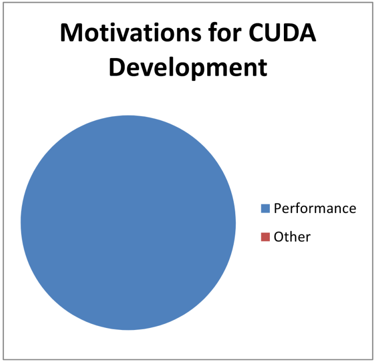
- Day job: writing code-generator for POS printers
- By night: GPU-accelerated evolutionary algorithm
- I will not be teaching CUDA
- (First talk ever)

CUDA

- A quick poll
- Why CUDA?
- GPU design philosophy
- Compiling a CUDA program
- Runtime vs Driver API

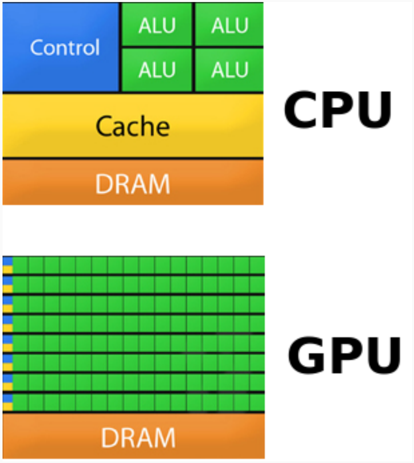
CUDA

- A quick poll
- Why CUDA?
- GPU design philosophy
- Compiling a CUDA program
- Runtime vs Driver API



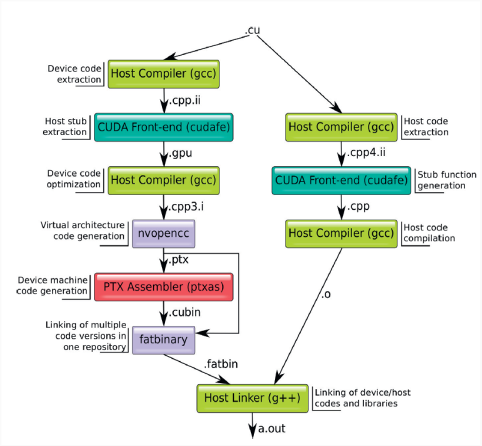
CUDA

- A quick poll
- Why CUDA?
- GPU design philosophy
- Compiling a CUDA program
- Runtime vs Driver API



CUDA

- A quick poll
- Why CUDA?
- GPU design philosophy
- Compiling a CUDA program
- Runtime vs Driver API



CUDA

- A quick poll
- Why CUDA?
- GPU design philosophy
- Compiling a CUDA program
- Runtime vs Driver API

HelloWorld.cu (Runtime API)

```
// nvcc -ccbin g++-5 -std=c++11 -g -m64 -gencode arch=compute_30,code=sm_30 \
// -o HelloWorld HelloWorld.cu
#include <stdio.h>

__global__ void TestKernel( int )
{
    printf( "Block: %d -- Thread %d\n" , blockIdx.x , threadIdx.x );
}

int main()
{
    // if constexpr( true ){} // C++17 -- will not compile
    // auto value = [](auto x) { return x; }; // C++14 -- will not compile
    TestKernel<<<8,2>>>( 42 );
    cudaDeviceSynchronize();
}
```

HelloWorld.cu (result)

```
$ ./HelloWorld
Block: 2 -- Thread 0
Block: 2 -- Thread 1
Block: 6 -- Thread 0
Block: 6 -- Thread 1
Block: 0 -- Thread 0
Block: 0 -- Thread 1
Block: 5 -- Thread 0
Block: 5 -- Thread 1
Block: 3 -- Thread 0
Block: 3 -- Thread 1
Block: 4 -- Thread 0
Block: 4 -- Thread 1
Block: 7 -- Thread 0
Block: 7 -- Thread 1
Block: 1 -- Thread 0
Block: 1 -- Thread 1
```

[Decoupling] client-code side (take 1)

```cpp
// InvokeKernel.h
template<typename ...Args>
void InvokeKernel( void(Args...) , int , int , Args ... args );

extern template void InvokeKernel<int>( void(int) , int , int , int );
extern template void InvokeKernel<int,int>( void(int , int) , int , int , int , int );
extern template void InvokeKernel<int,int,int>( void(int , int , int ) , int , int , int , int , int );
```

```cpp
// Kernel.h
// #define __global__ __attribute__((global))
// __global__ void TestKernel1( int ); // warning: 'global' attribute directive ignored
void TestKernel1( int );
void TestKernel2( int , int );
void TestKernel3( int , int , int );
```

```cpp
// main.cpp
#include "Kernel.h"
#include "InvokeKernel.h"

int main()
{
    if constexpr( true ){}  // C++17 feature
    auto value = [](auto x) { return x; }; // C++14 feature

    InvokeKernel( TestKernel1 , 8 , 2 , 42 ); // Instead of TestKernel1<<<8,2>>>( 42 )
    InvokeKernel( TestKernel2 , 8 , 2 , 42 , 43 ); // Instead of TestKernel2<<<8,2>>>( 42 , 43 )
    InvokeKernel( TestKernel3 , 8 , 2 , 42 , 43 , 44 ); // Instead of ...
}
```

[Decoupling] CUDA/library side (take 1)

```
// Kernel.cu
__global__ void TestKernel1( int )
{
    printf( "Block: %d -- Thread %d\n" , blockIdx.x , threadIdx.x );
}

__global__ void TestKernel2( int , int )
{
    printf( "Block: %d -- Thread %d\n" , blockIdx.x , threadIdx.x );
}

__global__ void TestKernel3( int , int )
{
    printf( "Block: %d -- Thread %d\n" , blockIdx.x , threadIdx.x );
}
```

```
// InvokeKernel.cu
template<typename ...Args>
void InvokeKernel( void kernel(Args...) , int n , int m , Args ... args )
{
    kernel<<<n,m>>>( args ... );
    //cudaDeviceSynchronize();
}

template void InvokeKernel<int>( void(int) , int , int , int );
template void InvokeKernel<int,int>( void(int , int ) , int , int , int , int );
template void InvokeKernel<int,int,int>( void(int , int , int ) , int , int , int , int , int );
```

[Decoupling] client-code side (take 2)

```cpp
// InvokeKernel.h
#include "Kernel.h"

template<typename> struct KernelWrapper; // Base template

template<typename ...Args>
struct KernelWrapper<void(Args...)>
{
    static void Invoke( void kernel(Args...) , int n , int m , Args ... args );
};

extern template struct KernelWrapper<decltype(TestKernel1)>;
extern template struct KernelWrapper<decltype(TestKernel2)>;
extern template struct KernelWrapper<decltype(TestKernel3)>;

template<typename K , typename ...Args>
void InvokeKernel( K k , int n , int m , Args ... args )
{
    typedef typename std::remove_pointer<K>::type KernelType;   // K is a *pointer* to a function
    KernelWrapper<KernelType>::Invoke( k , n , m , args ... );
}
```

```cpp
// main.cpp
#include "InvokeKernel.h"
int main()
{
    InvokeKernel( TestKernel1 , 8 , 2 , 42 );
}
```

[Decoupling] CUDA/library side (take 2)

```
// InvokeKernel.cu
#include "InvokeKernel.h"

template<typename ...Args>
void KernelWrapper<void(Args...)>::Invoke( void kernel(Args...)  , int n , int m , Args ... args )
{
    kernel<<<numBlocks,numThreads>>>( args ... );
    //cudaDeviceSynchronize();
}

template struct KernelWrapper<decltype(TestKernel1)>;
template struct KernelWrapper<decltype(TestKernel2)>;
template struct KernelWrapper<decltype(TestKernel3)>;
```

Fly in the ointment (two kernels same prototype)

```
// Kernel.h
void Kernel1( int );
void Kernel2( int );
```

```
// Kernel.cu
__global__ void Kernel1( int )
{
    printf( "Block: %d -- Thread %d\n" , blockIdx.x , threadIdx.x );
}

__global__ void Kernel2( int )
{
    printf( "Block: %d -- Thread %d\n" , blockIdx.x , threadIdx.x );
}
```

```
// InvokeKernel.cu
#include "Kernel.h"
...
template struct KernelWrapper<decltype(Kernel1)>;
template struct KernelWrapper<decltype(Kernel2)>; // ERROR (double instantiation)
```

Solution: outline

```
// InvokeKernel.cu
#include "Unique.hpp"
...
#define TypeList decltype(Kernel1),decltype(Kernel2)

//template struct KernelWrapper<decltype(Kernel1)>;
//template struct KernelWrapper<decltype(Kernel2)>;

template struct KernelWrapper<UniqueType<0,TypeList>>; // Type at index 0 or other some unique type
template struct KernelWrapper<UniqueType<1,TypeList>>; // Type at index 1 or other some unique type
```

Solution: Unique.hpp

```
#include <type_traits>

template<int Id1 , int Id2 , typename T , typename ... Ts>
struct Unique
{
        typedef Unique<Id1 + 1,Id2,Ts ...> Next;
        typedef typename Next::type type; // type at Id2
        static const bool value = !std::is_same<type , T>::value &&
                                  Next::value; // Is type at Id2 unique in (Id1,Id2]
};

template<int Id , typename T , typename ... Ts>
struct Unique<Id,Id,T,Ts ...>
{
        static const bool value = true;
        typedef T type;
};

template<int> struct Dummy{};

template<int I, typename ... Ts>
using UniqueType =
      typename std::conditional<
                                 Unique<0,I,Ts...>::value ,          // type at I is unique in [0,I]
                                 typename Unique<0,I,Ts...>::type ,  // type at I
                                 Dummy<I>                            // some other unique type
                               >::type;
```

Solution: one final change . . .

```
// Invoke.h

// Implement base template, for Dummy<int>
template<typename>
struct KernelWrapper
{
};

// Unchanged
template<typename ...Args>
struct KernelWrapper<void(Args...)>
{
    static void Invoke( void kernel(Args...) , int n , int m , Args ... args );
};

// Unchanged
extern template struct KernelWrapper<decltype(Kernel1)>;
extern template struct KernelWrapper<decltype(Kernel2)>;
```

Final word . . .

- Technique described can also be used to wrap CUDA libraries e.g. CUB
- By default, driver API decouples CUDA and non-CUDA code, **BUT**
- More complicated (HelloWorld $>$ 50LOC)
- nvcc better integrated with MSVC
- Recent versions of clang now compile CUDA

Thank you.

(Questions? . . . comments?)

✉ sola.aina@mail.com   @_SolaAina_