HyCuda

A Hybrid Framework Code-Generator for CUDA

Joren Heit

December 17, 2013

Abstract

This article descibes the HyCuda code-generator, that generates a Hybrid algorithm-framework that can be integrated in any C++(11) project. The framework implements a template metaprogramming mechanism that takes care of all memory-transfers, based on an input-file that describes the specifics of the algorithm. No extra care has to be taken when it is decided that a subroutine needs to be executed on a different device. Also, due to the compile-time nature of template metaprogramming, there is no overhead in deciding which routines (CPU- or GPU-code) to call (i.e. no runtime conditionals).

1 Introduction

CUDA (Compute Unified Device Architecture), is a computational framework by NVidia, now implemented on all modern NVidia GPU's (Graphics Processing Units). The framework allows for programming non-graphical applications that run on the highly parallel GPU architecture, using many different processes (threads) to process the data. The programming-language, CUDA-C, is like C or even partial C++, with some additional syntax.

Because of the insanely parallel nature of GPU's, some calculations can be accellerated by executing them as a kernel on the GPU instead of the CPU. Often, these calculations are subroutines to some larger algorithm, containing many more substeps. However, it is not always clear in advance which routines should best be executed on the CPU, and which on the GPU. Switching between devices is nontrivial, because data has to be up to date and ready to be read/written to by the device in question. These difficulties call for a mechanism that allows the programmer to try different combinations (paths) of devices without much effort, in order to choose the best option.

The HyCuda code-generator generates such a framework, based on a specification file that describes some properties of the algorithm. It generates C++11 header- and source-files, only a few of which have to be modified by the programmer in order to implement the algorithm itself. The structure of the specification-file and options to HyCuda, as well as the generated template mechanism, will be described in this document.

2 Specification-File

The specification-file tells HyCuda what you want to name the generated class, in which namespace you want to embed it, what the type of your input-data is, how you want to allocate memory and most importantly, what your algorithm looks like. It consists of 4 sections, seperated by two consecutive percent-signs (`%%`):

1. Directives
2. Memory
3. Routines
4. Order of execution

Each of the sections above will now be discussed in more detail. Keywords specific to HyCuda are prefixed by a single percent-sign, in order to be able to distinguish them from C++ keywords. There are two types of comment available in the HyCuda specification-file. Both end-of-line comments (`// single-line comment`) as C-style comments (`/* multi-line`

Keyword	Default
`namespace`	global namespace
`class-name`	`HyCudaAlgorithm`
`parameters`	`Params`
`memory`	`%dynamic`
`include`	no pre-includes

Table 1: Allowed keywords in the first (directives) section of the specification file, and their default values.

`comment */`) are supported. The internals of the spec-file will be discussed in the coming sections. Those who want to see what the final product looks like, can scroll to Section 4.

2.1 Directives

The syntax of the directives-section is

```
% keyword: value
```

where `keyword` is one of those listed in Table 1. When a keyword is not listed, the default-value will be taken.

2.1.1 namespace

When a namespace is specified, all HyCuda classes will be defined within this namespace. When no namespace is specified, all HyCuda-related classes will be declared in the global namespace.

2.1.2 class-name

The name of the class that will eventually be used to run the algorithm is specified by `class-name`. By default, the resulting class will be called `HyCudaAlgorithm`. For the remainder of this document, the resulting class will be refered to as such.

2.1.3 parameters

The algorithm assumes that a set of paramaters might be necessary to determine its behavior. These parameters will be listed in a struct, declared in the specification following the `parameters`-keyword. An instance of this struct will be passed to the algorithm, where its contents can be passed to the subroutines where necessary. Its syntax is regular C++, omitting the `struct` keyword and closing semicolon. For example:

```
% parameters: Params { // opening '{' must be on the same line
  float x, y, z;
  int q;
}
```

If the parameters are omitted, an empty struct called `Params` will be generated.

2.1.4 memory

There are two memory-allocation modes available, indicated by the keywords `%static` and `%dynamic`. When the mode is set to *static*, HyCuda generates code to allocate all memory when the final `HyCudaAlgorithm`-object is instantiated, and frees the memory when the object is destroyed. For this to be possible, the number of elements in each block of memory must be known in advance, and be independent of runtime input. The memory-sizes (see Section 2.2) can still be symbolic values, but these should be defined (e.g. in an `enum` or by `#define`'s) by the user and pre-included (e.g. using `%include`) in order to be seen by the allocation functions.

When the allocation-mode is set to `%dynamic`, array-sizes may depend on the runtime input. Only when the `process`-member is called on the object, to which the input and output parameters are passed (Section 2.2.1), will the memory be allocated. More on this on Section 2.2.

2.1.5 include

The include keyword allows for inclusion of header files by the algorithm. These header files may be standard-library files, within angle brackets (<file>), or user-generated files within double quotes ("file").

2.2 Memory

The memory-section contains information about all memory that is potentially 'shared' between the CPU and GPU. Based on this section, HyCuda will generate code that allocates the memory on both devices. The syntax is as follows:

```
identifier [(io-specifiers)]: data-type, elements
```

Here, identifier is the name of the variable (pointer) that points to the memory, io-specifiers is an optional sequence of input (i) and output (o) specifications, data-type is the type of the variable (omitting the asterix even though they will be pointers eventually), and elements is the number of elements in the array. The latter can both be a hardcoded numerical value, or a symbolic variable.

2.2.1 Input/Output

The io-specifiers tell HyCuda which variables will be used as input or output. When a variable is marked as input (i), output (o) or both (io), HyCuda assumes that these variables already exist on the CPU. Two structs will be declared, Input and Output, containing pointers that should be initialized by the programmer and passed to process to execute the algorithm. HyCuda will make sure that the output will be readily available (on the CPU) when the algorithm returns.

More precisely, both Input and Output contain std::pair's of pointers and sizes (size_t). For example:

```
struct Input
{
    std::pair<float*, size_t> vec1;
    // ...
}
```

The value passed as vec1.second will be assigned to the symbolic memorysize specified in the HyCuda spec-file as the size of vec1. See also Section 4 for a detailed example.

2.2.2 Memory-Access

HyCuda will generate a Memory class, which will be one of the base-classes of HyCudaAlgorithm. Memory owns two structures, hostMem and devMem, which contain pointers to all memory on the host and device respectively. Through these pointers, all memory can be accessed. Also, an instance of devMem is copied to the device itself. A pointer to this struct of device-pointers, dev_devMem, can be passed to the kernels instead of having to pass all necessary pointers individually.

2.3 Routines

The 3rd section contains information about every subroutine. For each subroutine, the programmer needs to specify on which memory this routine depends, and in what way (read, write or both). This helps HyCuda to determine which data has to be available on which device at any point in the algorithm. The syntax of the routine-section looks like this:

```
function-identifier: memory-identifier (permissions), ...
```

For example, a function fun1 that will read from array1 and will both read and write to array2 will be declared as

```
fun1: array1 (r), array2 (rw)
```

The memory-identifiers should match those declared in the memory-section, and all functions that are part of the algorithm should be declared here.

2.4 Order of Execution

The subroutines listed in the routine-section of the specification do not have to be declared in the order in which they are intended to be executed. For extra flexibility, it is possible to define the specific order of execution in the final section. The syntax of this final section is:

```
% order: $function-number, $function-number, ...
```

Here, `function-number` corresponds to the position (top = 1, downwards) of the function in the previous section. When the order of execution is identical to the order in the 3rd section, the keyword `%default` may be used:

```
% order: $1, $2, $3, $4
      or
      % order: %default
```

3 Using HyCuda

3.1 Files and Options

3.1.1 Files

When the specification-file has been constructed, it will be fed to HyCuda as its first input argument. HyCuda can be called from the commandline, expecting the following syntax:

```
hycuda spec-file [options]
```

When HyCuda is called without any options, the files listed below are generated in the current directory, where `X` denotes the class-name as specified in the specification (all lowercase).

```
X.h
X_algorithm.h
X_cudastopwatch.cuh    X_cudastopwatch.ih   X_cudastopwatch.cu
X_default_base.h
X_default_cpu.h        X_default_cpu.ih     X_default_cpu_impl.cc
X_default_gpu.h        X_default_gpu.ih     X_default_gpu_impl.cu
X_devicepolicies.h
X_hybrid.h
X_inputpolicy.h        X_inputpolicy.ih     X_inputpolicy_impl.cc
X_kernels.h            X_kernels.cu
X_memory.h             X_memory.ih          X_memory.cu / X_memory.cc
X_stopwatch.h
X_timer.h              X_timer.ih           X_timer.cc

Makefile
```

There exist many common source-file-extensions in the world of C++; `.cpp`, `.cc`, `.cxx` or `.C` for source-files and `.h` or `.hpp` for header-files. In addition to the familiar source- and header-files, for which I have chosen the `.cc` and `.h` extension, there are a few other extensions visible in the list of files above. The uncommon `.ih`-files are so-called *implementation headers*, which should by convention only be included by source-files. The purpose of these implementation headers is to include other headers that are only needed by the implementation, or contain `using`-directives (e.g. `using namespace std;`) for convenience of the programmer.

The CUDA sources have the `.cu` extension, and header-files that are exclusively seen by the CUDA compiler (`nvcc`) have the `.cuh` extension. Even though these are a lot of files, the user will only have to be familiar with a few to be able to implement his own algorithm (Section 3.3.

3.1.2 `-generate_main (-m)`

Optionally, an additional file (`X_main.cc`), containing the `main` function, can be generated using the `-generate_main (-m)` flag. This file also contains the instan tiation of the resulting object, and a call to the algorithm (which is implemented as a sequence of empty functions).

3.1.3 `-regenerate_implementations (-r)`

To prevent modified implementation-files to be overwritten when HyCuda is re-run, some user-implementated are not regenerated when they already exist (these files are listen in Section 3.3). This default behavior may be overruled by using the `regenerate_implementations (-r)` flag.

3.1.4 `-folder (-f)`

By default, the files will be generated in the current directory. The `-folder=... (-f)` flag specifies a different (pre-existing) directory for the files to be put into.

3.2 Members

This section will provide an overview of the members that the resulting class has access to, some of which have already been mentioned previously in the text. Not all of the members actually belong to the final class. Instead, they belong to one of the baseclasses up in the hierarchy. For more information about the class-hierarchy, the user is referred to Section 6.1. Table 2 lists the members that could be helpful in defining the implementation of your routines.

Member	Access	Type	Class	Description
hostMem	private	HostMem	Memory	Structure containing **host**-pointers.
devMem	private	DevMem	Memory	Structure containing **device**-pointers.
dev_devMem	private	DevMem*	Memory	Same as devmem, but points to a struct on the device.
resetHostMem	private	void (*)(unsigned char)	Memory	Sets every byte in the **host**-memory to the character specified.
resetDeviceMem	private	void (*)(unsigned char)	Memory	Sets every byte in the **device**-memory to the character specified.
runtimes	private	std::vector<double>	Timer	When the macro NO_TIMERS has *not* been defined, all the runtimes are stored here.
memcpytimes	private	std::vector<double>	Timer	When the macro NO_TIMERS has *not* been defined, all the (cuda)memcpy-times are stored here.
elapsedRunTime	public	double(*)(int n) std::vector<double>(*)()	Timer	Returns the time taken to execute the n'th routine or the complete vector (== runtimes)
elapsedMemcpyTime	public	double(*)(int n) std::vector<double>(*)()	Timer	Returns the time taken to execute the memcpy's prior to the n'th routine or the complete vector (== memcpytimes)
process	public	void (*)(Input const &, Output const &)	Algorithm	Executes the routines in the specified order.
printTimes	public	void (*)()	Algorithm	Prints a complete list of runtimes and memcpy-times

Table 2: Members of the resulting algorithm-class and the (base-)class in which they are declared. Note: the access-rights are with respect to the final class. This means that a member that is public in its own class can be private in the inheriting class.

3.2.1 init() and close()

Two other members, which are not listed in Table 2, should be mentioned. These are the functions init() and close(). These functions are called by process() *before* and *after* the routines have been executed. Each of the classes Hybrid_, DefaultCPU and DefaultGPU are equipped with these members, but the former merely calls those of its parents. That is, Hybrid_'s init() and close() members will call those of DefaultCPU and DefaultGPU, in that order.

By default, the init() implementation in DefaultCPU only calls the private member setMemSizes(), which sets the memory-size variables based on the input. This means that when the size of an input or output-variable is bound in the specification-file to some symbol (vectorSize in the example of Section 4), the symbol will be assigned the value corresponding to the size of the input. The default implementation of DefaultGPU::init() does absolutely nothing.

The default implementation of DefaultGPU::close() however, makes sure that the device and host are synchronized before returning from process() by calling cudaDeviceSynchronize(). In contrast, DefaultCPU::close() is an empty function by default.

3.3 User Implementations

When all files have been generated, it is up to the programmer to implement the actual algorithm. Several files *may* be modified to do so:

`X.h`	Specify the device-order.
`X_default_base.h`	Declare additional members, used by the `DefaultCPU`- and `DefaultGPU`-classes.
`X_default_cpu_impl.cc`	Implement the CPU-routines.
`X_default_cpu_impl.ih`	CPU implementation header.
`X_default_cpu_impl.cu`	Implement the GPU-routines.
`X_default_cpu_impl.cc`	GPU implementation header.
`X_kernels.h`	Declare the GPU-kernels.
`X_kernels.cu`	Implement the GPU-kernels.
`X_main.cc`	Optionally implement `main()`.

3.3.1 Main Header: `X.h`

The final header-file, `X.h` (where `X` still denotes the class-name), contains only some `typedef`'s, and may look something like this:

```
/* hycudaalgorithm.h */

#include "hycudaalgorithm_algorithm.h"
#include "hycudaalgorithm_hybrid.h"
#include "hycudaalgorithm_devicepolicies.h"

namespace Example {

// Specify which device to use for each of the routines (CPU/GPU)
typedef DevicePolicies <
    Fun1Device< CPU >,
    Fun2Device< CPU >,
    Fun3Device< CPU >

> CustomPolicy;

typedef Hybrid_< CustomPolicy > Hybrid;
typedef HyCudaAlgorithm_< Hybrid > HyCudaAlgorithm;

} // namespace Example
```

The very last definition is defining the class that the end-user will actually instantiate (`HyCudaAlgorithm`). For each subroutine, the device that will execute it is defined in the first definition. In this example, the subroutines were named `fun1`, `fun2` and `fun3` respectively. HyCuda has generated corresponding template classes `Fun1Device`, `Fun2Device` and `Fun3Device`, from which the template argument (either `CPU` or `GPU`) tells the mechanism which function to call.

3.3.2 Baseclass Header: `X_default_base.h`

To avoid having to modify the `DefaultCPU` and `DefaultGPU` header-files, and allow data to be shared between these classes, they are both virtually derived from `DefaultBase`. Whenever the programmer feels that it is necessary to declare additional datamembers (that are not suitable as parameters), this interface can be used to do so. Keep in mind though that this data will not be available on the GPU, unless copied manually. The example (Section 4) illustrates one such application.

3.3.3 Default CPU-implementations: `X_default_cpu_impl.cc`

This is where you should implement the CPU-routines. Because the `DefaultCPU` has both `DefaultBase` and `Memory` as its base-classes, it has access to each of the memory-locations in the `hostMem` member (owned by `Memory`), and to additional members defined in the `DefaultBase` interface.

3.3.4 Default GPU-implementation: `X_default_gpu_impl.cu`

This file is meant for the CPU-side of the GPU-implementation. It is where you call your kernels, your Thrust-algorithms, or any other CUDA-related functions. The implementation-header `X_default_gpu_.ih` includes the declarations of your kernels (`X_kernels.h`), which should be defined in a file called `X_kernels.cu`. If you need to include, for example, Thrust-headers, it is recommended to do this in the implementation header as well.

3.3.5 Main: `X_main.cc`

If the `-m` option was passed to HyCuda, an additional file containing the `main` function will be generated. In this file, one has to assign the appropriate parameters to the `Params` object, and initialize the contents of `Input` and `Output`. Section 4 will show how exactly this goes about.

4 Example

4.1 Specification-File

This section will provide an example specification-file and implementation, implementing a simple 3-step algorithm, assuming the existence of two vectors, `vec1` and `vec2`:

1. Multiply `vec1` and `vec2` by the parameter `m1`.
2. Add the two vectors together, and store the result in `vec3`.
3. Multiply `vec3` by another parameter `m2`.
4. Calculate the sum of `vec3`.

These three steps can easily be factored into one, but for the sake of illustration, they will be kept seperately. The input vectors `vec1` and `vec2` will be stored in some file, and will serve as the input of our algorithm. The specification-file can now be defined as follows:

```
/* spec.hycuda */

// Directives
% namespace: Example class-name: ExampleAlgorithm parameters: Params {
    float m1;
    float m2;
}

%% // Memory-Section
vec1 (i)     : float, vectorSize
vec2 (i)     : float, vectorSize
vec3         : float, vectorSize
sum (o)      : float, 1

%% // Routine-dependencies
multiplyM1: vec1 (rw), vec2 (rw)
multiplyM2: vec3 (rw)
addVec1Vec2: vec1 (r), vec2 (r), vec3 (w)
sumVec3: vec3 (r), sum (w)

%% // Routine-order
% order: $1, $3, $2, $4
```

We have chosen for the default memory-allocation scheme (dynamic), because we cannot know in advance how large the vectors in the file are. Another option would have been to set a maximum vector size, and check if `vectorSize` is within this maximum. However, because we only process one run of data, the dynamic scheme will not cause any unnecessary overhead.

To generate the framework, call HyCuda from the commandline:

```
$ hycuda spec.hycuda -m
```

4.2 Implementation

4.2.1 main

Because we identified `vec1`, `vec2` and `sum` as input and output, these will be assumed to exist on the CPU and will not be allocated anymore (on the host). The `main`-function will therefore be responsible for reading the input from a file, and passing pointers to the input and their size to the `process` member.

```
/* examplealgorithm_main.cc */

#include "examplealgorithm.h"
using namespace Example
using namespace std;

size_t readVectorsFromFile(char const *filename,
                           float **v1, float **v2);

int main(int argc, char **argv)
{
    // Initialize parameters
    Params params;
    params.m1 = 2;
    params.m2 = 3;

    // Initialize vectors
    float *v1, *v2;
    size_t vectorSize = readVectorsFromFile(argv[1], &v1, &v2);

    // Initialize input
    Input in;
    in.vec1 = {v1, vectorSize};
    in.vec2 = {v2, vectorSize};

    // Initialize output
    float sum;
    Output out;
    out.sum = {&sum, 1};

    // Process
    ExampleAlgorithm alg(params);
    alg.process(in, out);

    // Output
    cout << "Sum: " << sum << '\n';
}
```

4.2.2 Baseclass header

We slightly modify the baseclass header to contain the number of blocks and threads that will be used by the kernels. This will prevent us from having to recalculate these values in each of the GPU routines again and again. Strictly speaking, these values should be stored in the `DefaultGPU`-class, as they're not being used by the CPU implementation. However, the `DefaultGPU`-header will be regenerated when we call HyCuda again for some reason, thereby losing any changes we made. This is not true for the baseclass header.

```
#ifndef ExampleAlgorithm_DEFAULT_BASE_INCL
#define ExampleAlgorithm_DEFAULT_BASE_INCL
#include "examplealgorithm_algorithm.h"

namespace Example {

class DefaultBase
{
protected:
```

```
    uint blocks;
    uint threads;
};

} // namespace Example
#endif // ExampleAlgorithm_DEFAULT_BASE_INCL
```

4.2.3 GPU Implementation

To implement the GPU routines, we modify the `examplealgorithm_gpu_impl.cu` file, which will call our kernels. The implementation details are irrelevant, and will not be discussed in detail. The previous section concerned the addition of two members, `threads` and `blocks`, to the baseclass. These will be initialized by `DefaultGPU`'s `init()` member. For the sake of completeness, the entire GPU implementation is listed below, including the kernel declarations and implementations.

```
/* examplealgorithm_kernels.h */

#ifndef ExampleAlgorithm_KERNELS_INCLUDED
#define ExampleAlgorithm_KERNELS_INCLUDED

namespace Example {

__global__ void vectorMultiply(float *v1, uint size, float mul);
__global__ void vectorAdd(float *v1, float *v2, float *dst, uint sz);
__global__ void vectorSum(float *v1, uint size, float *result);

} // namespace Example


#endif // ExampleAlgorithm_KERNELS_INCLUDED
```

```
/* examplealgorithm_gpu_impl.cu */

#include "examplealgorithm_default_gpu.ih"

void DefaultGPU::init(Input const &in, Output const &out)
{
    uint const maxThreadsPerBlock = 512;
    blocks = (vectorSize + maxThreadsPerBlock - 1) /
        maxThreadsPerBlock;
    threads = vectorSize / blocks + 1;
}

void DefaultGPU::close()
{
    cudaDeviceSynchronize();    // Wait for GPU to be finished before
        returning
}

void DefaultGPU::multiplyM1_()
{
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);

    vectorMultiply <<< blocks, threads, 0, stream1 >>>
        (devMem.vec1, vectorSize, d_params.m1);
    vectorMultiply <<< blocks, threads, 0, stream2 >>>
        (devMem.vec2, vectorSize, d_params.m1);
}

void DefaultGPU::multiplyM2_()
{
```

```
    vectorMultiply <<< blocks, threads >>> (devMem.vec3, vectorSize,
        d_params.m2);
}

void DefaultGPU::addVec1Vec2_()
{
    vectorAdd <<< blocks, threads >>> (devMem.vec1, devMem.vec2,
        devMem.vec3, vectorSize);
}

void DefaultGPU::sumVec3_()
{
    cudaMemset(devMem.sum, 0, sizeof(float));
    vectorSum <<< blocks, threads, threads * sizeof(float) >>> (devMem
        .vec3, vectorSize, devMem.sum);
}
```

```
/* examplealgorithm_kernels.cu */

#include "examplealgorithm_kernels.h"

__global__
void Example::vectorMultiply(float *v1, uint size, float multiplier)
{
    uint tid = threadIdx.x + blockIdx.x * blockDim.x;
    uint nThreads = blockDim.x * gridDim.x;

    uint idx = tid;
    while (idx < size)
    {
        v1[idx] *= multiplier;
        idx += nThreads;
    }
}

__global__
void Example::vectorAdd(float *v1, float *v2, float *dest, uint size)
{
    uint tid = threadIdx.x + blockIdx.x * blockDim.x;
    uint nThreads = blockDim.x * gridDim.x;

    uint idx = tid;
    while (idx < size)
    {
        dest[idx] = v1[idx] + v2[idx];
        idx += nThreads;
    }
}

__global__
void Example::vectorSum(float *v1, uint size, float *result)
{
    __shared__ extern float buffer[];

    // initialize shared memory to part of the vector (#elements == #
        threads per block)
    uint tid = threadIdx.x;
    uint idx = blockIdx.x * blockDim.x + tid;
    if (idx < size)
        buffer[tid] = v1[idx];
    else
        buffer[tid] = 0;

    __syncthreads();

```

```
    uint currentArraySize = blockDim.x;
    while (currentArraySize > 1)
    {
        uint secondHalfBegin = (1 + currentArraySize) / 2;
        if (secondHalfBegin + tid < currentArraySize)
            buffer[tid] += buffer[secondHalfBegin + tid];

        __syncthreads();
        currentArraySize = secondHalfBegin;
    }

    // buffer now contains the (partial) sum of this block -> atomic
        add to result
    if (tid == 0)
        atomicAdd(result, buffer[0]);
}
```

4.2.4 CPU Implementation

The CPU routines are implemented in the `examplealgorithm_default_cpu_impl.cc`, listed below. In this case, the `init()` function can be kept as is, only calling `setMemSizes()` to set the value of `vectorSize`, which was deduced from the input. When the size of an array can nót be deduced from input, this is where you should set the size (provided it is a symbolic, non-const value in a dynamic scheme).

```
/* examplealgorithm_default_cpu.cc */

#include "examplealgorithm_default_cpu.ih"

void DefaultCPU::init(Input const &in, Output const &out)
{
    setMemSizes(in, out);
}

void DefaultCPU::close()
{
}

void DefaultCPU::multiplyM1_()
{
    float *vec1 = hostMem.vec1;
    float *vec2 = hostMem.vec2;

    float m1 = d_params.m1;
    for (size_t i = 0; i != vectorSize; ++i)
    {
        vec1[i] *= m1;
        vec2[i] *= m1;
    }
}

void DefaultCPU::multiplyM2_()
{
    float m2 = d_params.m2;
    float *vec3 = hostMem.vec3;
    for (size_t i = 0; i != vectorSize; ++i)
        vec3[i] *= m2;
}

void DefaultCPU::addVec1Vec2_()
{
    float *vec1 = hostMem.vec1;
    float *vec2 = hostMem.vec2;
    float *vec3 = hostMem.vec3;

```

```
    for (size_t i = 0; i != vectorSize; ++i)
        vec3[i] = vec1[i] + vec2[i];
}

void DefaultCPU::sumVec3_()
{
    float *vec3 = hostMem.vec3;
    float &sum = *hostMem.sum;
    sum = 0;
    for (size_t i = 0; i != vectorSize; ++i)
        sum += vec3[i];
}
```

4.2.5 Main Header

When all the hard work has been finished, it is time to determine the devices that will execute the functions. This is handled by the `DevicePolicies` typedefinition in the main header: `examplealgorithm.h`. Let's say we want to perform all routines on the GPU, except for the last one. The header should be modified to look like this:

```
#include "examplealgorithm_algorithm.h"
#include "examplealgorithm_hybrid.h"
#include "examplealgorithm_devicepolicies.h"

namespace Example {

typedef DevicePolicies <
  MultiplyM1Device< GPU >,
  MultiplyM2Device< GPU >,
  AddVec1Vec2Device< GPU >,
  SumVec3Device< CPU >

> CustomPolicy;

typedef Hybrid_< CustomPolicy > Hybrid;
typedef ExampleAlgorithm_< Hybrid > ExampleAlgorithm;

} // namespace Example
```

4.3 Building the Example

When no additional source-files are added to the project, it should now be ready to be built using the pre-generated Makefile. This Makefile was organized such that a C++11 capable compiler will compile all C++ source-files (`nvcc` cannot parse all C++11 syntax yet), `nvcc` will compile all CUDA source-files, and will link the resulting object files.

5 Manual Addition of Routines

It is not always convenient to re-run HyCuda when you need to add functionality, so this section will provide instructions on how to do this manually. The next section (Sec. 6) should contain enough background information to back this section up when needed.

5.1 Member Declarations

Let us start with declaring the new member-functions in every necessary header, assuming that you want both a CPU and a GPU routine to be implemented. In the `X_default_cpu.h` and `X_default_gpu.h` header-files, you will find function-declarations of the form `void routine()` in the public section and `void routine_()` (note the underscore) in the private section of the class interface. This is where the new routine should be declared in two-fold: with and without a trailing underscore. The non-underscored version is merely a wrapper of the underscored version, and takes care of the timing (if necessary). Each of the non-underscored routines has an inline implementation in the same file, which looks like this (same for DefaultGPU):

```
inline void DefaultCPU::routineName()
{
#ifndef NO_TIMERS
  d_sw.start();
#endif

  routineName_();

#ifndef NO_TIMERS
  d_sw.stop();
  runtimes.push_back(d_sw.read());
#endif
}
```

Assuming there are already other inline implementations present, you can simply copy-paste and change the routine-identifier.

The Hybrid_ class also contains function-declarations in its public section (non-underscored). The implementation is again a wrapper, this time of either the CPU or GPU version. Exactly which one is determined by the DevicePolicies, which we will come to later. Their inline implementation (in the same file) should look like this:

```
template <typename DevicePolicies, typename CPUType, typename GPUType>
void Hybrid_<DevicePolicies, CPUType, GPUType>::routineName()
{
    Sync<RoutineDevice, SourceDevice>(Memory::hostMem.data, Memory::
        devMem.data, sizeof(float) * Memory::vectorSize, this);

    // ... other synchronizations

    RoutineDevice:routineName();
}
```

This is also the tricky part. You have to figure out for yourself which device has the data that is used by this particular routine (SourceDevice), and make sure you synchronize with it.

5.2 DevicePolicies

In the previous section, we have already encountered the DevicePolicies that are used to synchronize data between CPU and GPU. These policies of course have to be declared somewhere, which is done in the X_devicepolicies.h file. It defines the macro ADD_DEVICE_POLICY which allows you to easily add policies. It expects 2 arguments: the name of the policy and its ID, which should be unique! I suggest you just keep incrementing this ID to keep things simple. Once your policy has been added, you can move back to the header file in which Hybrid_ is declared.

A lot is going on in the private section of Hybrid_, but we now focus on the list of typedefs of the form

```
typedef typename UseParent<typename DevicePolicies::template Get<
    RoutineDevice>::Device>::Parent RoutineDevice;
```

Here, RoutineDevice is the device used for the routine you are adding. It uses the TMP mechanism to find out which device, CPU or GPU, you have specified in the DevicePolicices. Again, just copy-paste and make sure the names are consistent.

Of course, the typedef for DevicePolicies itself should also be edited. As you might remember, it resides in the main header X.h (an example was shown in Section 4.2.5).

5.3 Adding it to the Algorithm

To finish things up, it should be called by process() in the final algorithm-class (X_algorithm.h). Simply look it up and add a call to whatever routine you have added by now. Make sure it is in the right place, corresponding to the synchronization you implemented in the `Hybrid_` class! All you need to do now is implement the new routines in the implementation files (e.g. `X_default_cpu_impl.cc`). Will it compile...?

6 Behind the Scenes: the Template Framework

6.1 Class Hierarchy

To gain some insight in the organisation of the generated framework, its class-hierarchy is depicted in Figure 1. The direction of the arrow is towards the base class.

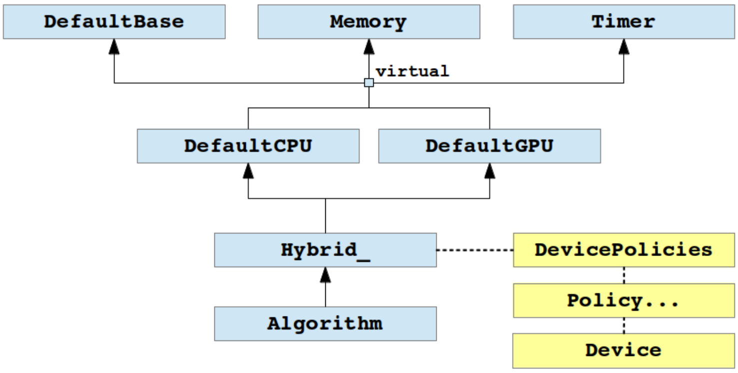

Figure 1: Class-hierarchy of the template framework generated by HyCuda. The arrows between the blue blocks represent class-inheritance, whereas the dashed lines represent template parameters.

The final `Algorithm` class is derived from `Hybrid_` which is itself multiply derived from both `DefaultCPU` and `DefaultGPU`. `Hybrid_` takes a template parameter called `DevicePolicies` to determine which of its parents, who provide the implementations, to use. The `DevicePolicies` class itself has a list of policies of the form `PolicyName<Device>`, which can be recursively searched to match a certain routine to its device.

At the root of the inheritance tree stand three classes, called `DefaultBase`, `Memory` and `Timer`. Both `DefaultCPU` and `DefaultGPU` derive virtually from these classes, such that they will share the same objects once instantiated.

The `DefaultBase` class is a convenience-class in which the user can add more data or through which the CPU and GPU implementations can communicate. It also prevents the user from having to modify the header-files corresponding to `DefaultCPU` and `DefaultGPU`, which will be regenerated and overwritten on a second call to HyCuda.

`Memory` holds pointers to both the CPU and GPU memory. Both implementations need to be aware of each memory-pool and when instantiated, the memory should only be allocated *once*, hence the virtual derivation.

For your convenience, `Timer` will time every routine and every memory-transfer, such that different paths through device-space can be compared without effort. Of course, there is a little overhead involved in timing the algorithm. Therefore, when the macro `NO_TIMERS` is defined, this will not be done.

6.2 DevicePolicies

The class-template `DevicePolicies` is designed to serve as a template parameter to the `Hybrid_` struct. It offers a metaprogramming interface which can be investigated to find out which device to use for each of the routines. Its definition is listed below. For convenience, the output from the example in Section 4 is used.

```
// Possible devices
struct CPU{};
struct GPU{};
struct Dummy{};

// TMP if-construction
```

```
template <bool val, typename TrueType, typename FalseType>
struct If
{
    typedef TrueType Type;
};

template <typename TrueType, typename FalseType>
struct If<false, TrueType, FalseType>
{
    typedef FalseType Type;
};

// Macro to easily create new device-policies
#define ADD_DEVICE_POLICY(name, id_) \
    template <typename Device_>      \
    struct name                      \
    {                                \
        typedef Device_ Device;      \
        enum { id = id_ };           \
    };

// Device-policies, names based on the routines
// id's must be unique!
ADD_DEVICE_POLICY( MultiplyM1Device,  0 );
ADD_DEVICE_POLICY( MultiplyM2Device,  1 );
ADD_DEVICE_POLICY( AddVec1Vec2Device, 2 );
ADD_DEVICE_POLICY( SumVec3Device,     3 );

// DevicePolicies class-template, expecting a list of policies (defined above)
template <typename ... PolicyList>
struct DevicePolicies
{
    enum { count = sizeof ... (PolicyList) };

    template <typename Policy, typename First, typename ... Rest>
    struct Get_
    {
        enum
        {
            id1 = Policy::id,
            id2 = First::id
        };

        typedef typename If < id1 == id2,
            typename First::Device,
            typename Get_<Policy, Rest ...>::Device
        >::Type Device;
    };

    template <typename Policy, typename First>
    struct Get_<Policy, First>
    {
        enum
        {
            id1 = Policy::id,
            id2 = First::id
        };

        typedef typename If < id1 == id2,
            typename First::Device,
            void    // policy not found! error!
        >::Type Device;
    };

```

```
    template <template <typename Device> class Policy>
    struct Get
    {
        typedef typename Get_<Policy<Dummy>, PolicyList ...>::Device
            Device;
    };
};
```

When a type of the form `DevicePolicies<...>` is passed as a template parameter to a `Hybrid_`, the `count`-value can be used to find out how many policies were specified (mostly to check for consistency). More importantly, its `Get`-member is used to get the device that should be used for a certain routine. In our case, for example `Get<SumVec3Device>::Device` will be a `typedef` for either `CPU` or `GPU`, depending on what you specified. Note that `Get` expects a *template template* argument. This prevented me from using template specialization techniques to find out which device was passed as a template argument to a certain policy. Instead, I had to equip each of the policies with a unique ID that is used to match the template-template parameter and extract its device-type.

6.3 Hybrid_

```
template <typename DevicePolicies = OnlyCPUPolicy,
          typename CPUType = DefaultCPU,
          typename GPUType = DefaultGPU>
class Hybrid_: public CPUType, public GPUType
{
    template <typename Device, typename Dummy = void>
    struct UseParent;

    template <typename Dummy>
    struct UseParent<GPU, Dummy>
    {
        typedef GPUType Parent;
    };

    template <typename Dummy>
    struct UseParent<CPU, Dummy>
    {
        typedef CPUType Parent;
    };

    template <typename DestDevice,
              typename SrcDevice,
              typename Dummy = void>
    struct Sync
    {
        Sync(void*, void*, size_t, Hybrid_<DevicePolicies>* = 0) {};
        // default: don't sync anything
    };

    template <typename Dummy>
    struct Sync<CPUType, GPUType, Dummy>
    {
        CudaStopwatch d_sw;
        Sync(void *host_ptr, void *dev_ptr, size_t size, Hybrid_<
            DevicePolicies, CPUType, GPUType> *timer = 0);
    };


    template <typename Dummy>
    struct Sync<GPUType, CPUType, Dummy>
    {
        CudaStopwatch d_sw;
        Sync(void *host_ptr, void *dev_ptr, size_t size,
             Hybrid_<DevicePolicies, CPUType, GPUType> *timer = 0);
```

```
    };

    typedef typename UseParent<typename DevicePolicies::template
        Get<MultiplyM1Device>::Device>::Parent MultiplyM1Device;
    typedef typename UseParent<typename DevicePolicies::template
        Get<MultiplyM2Device>::Device>::Parent MultiplyM2Device;
    typedef typename UseParent<typename DevicePolicies::template
        Get<AddVec1Vec2Device>::Device>::Parent AddVec1Vec2Device;
    typedef typename UseParent<typename DevicePolicies::template
        Get<SumVec3Device>::Device>::Parent SumVec3Device;

public:
    Hybrid_(Params const &params);

    void init(Input const &in, Output const &out);
    void close();


    void multiplyM1();
    void multiplyM2();
    void addVec1Vec2();
    void sumVec3();

};
```

There are several things going on here, but let's first focus on the public interface. It declares the routines as specified in the specification file, returning void and accepting no arguments at all. The implementation of these members is inherited from `Hybrid_`'s base-classes. More on this later.

6.3.1 Synchronization

The programmer has specified which data is used by each of the functions. Therefore, regardless of what happened prior to executing this function, this data should be up-to-date on the active device. The class-template `Sync` was designed to serve exactly this purpose. Let's ignore the `Dummy` parameter, as it was only necessary to specialize `Sync` within the scope of `Hybrid_` (one of the template peculiarities in C++). `Sync` expects two template parameters which should correspond to the parents of `Hybrid_` (`DefaultCPU` and `DefaultGPU` by default). The first argument corresponds to the destination device, i.e. the device executing the current routine, whereas the second argument is the device that currently has the data. By default, noting happens when a `Sync` object is being instantiated. It is however specialized for two other cases: CPU → GPU, and GPU → CPU.

The implementation of `Sync` won't be listed here, but all it does is call `cudaMemcpy`, where the direction (e.g. `cudaMemcpyHostToDevice`) depends on the specialization. An example of the implementation of one of the routines however, is listed below:

```
template <typename DevicePolicies, typename CPUType, typename GPUType>
void Hybrid_<DevicePolicies, CPUType, GPUType>::multiplyM1()
{
    Sync<MultiplyM1Device, CPUType>(Memory::hostMem.vec1,
                                    Memory::devMem.vec1,
                                    sizeof(float)*Memory::vectorSize,
                                    this);

    Sync<MultiplyM1Device, CPUType>(Memory::hostMem.vec2,
                                    Memory::devMem.vec2,
                                    sizeof(float)*Memory::vectorSize,
                                    this);
    MultiplyM1Device::multiplyM1();
}
```

Because the data-dependencies of `multiplyM1()` have been specified as *input*, they are assumed to reside on the CPU. Therefore, the `Sync` instantiation uses `CPUType` as the source and `MultiplyM1Device` as its destination for both `vec1` and `vec2`. Here, `MultiplyM1Device` is one of the `typedef`'s from `Hybrid_`, using the `Get` facility from the `DevicePolicies` parameter.

7 Contact

HyCuda was developed as a by-product of a thesis, and was therefore never used in any major project. I realise that I have probably overseen many (edge-)cases that complicate its use. I am very curious to whether this will ever be used in a serious project, so any feedback is very much appreciated! Don't hesitate to contact me through email at jorenheit@gmail.com.

Thanks!