CuPP – A framework for easy CUDA integration

Jens Breitbart
Research Group Programming Languages / Methodologies
Universität Kassel
Kassel, Germany
jbreitbart@uni-kassel.de

Abstract

This paper reports on CuPP, our newly developed C++ framework designed to ease integration of NVIDIAs GPGPU system CUDA into existing C++ applications. CuPP provides interfaces to reoccurring tasks that are easier to use than the standard CUDA interfaces. In this paper we concentrate on memory management and related data structures. CuPP offers both a low level interface – mostly consisting of smartpointers and memory allocation functions for GPU memory – and a high level interface offering a C++ STL vector wrapper and the so-called type transformations. The wrapper can be used by both device and host to automatically keep data in sync. The type transformations allow developers to write their own data structures offering the same functionality as the CuPP vector, in case a vector does not conform to the need of the application. Furthermore the type transformations offer a way to have two different representations for the same data at host and device, respectively. We demonstrate the benefits of using CuPP by integrating it into an example application, the open-source steering library OpenSteer. In particular, for this application we develop a uniform grid data structure to solve the k-nearest neighbor problem that deploys the type transformations. The paper finishes with a brief outline of another CUDA application, the Einstein@Home client, which also requires data structure redesign and thus may benefit from the type transformations and future work on CuPP.

1. Introduction

In the last years graphics processing units (GPUs) have evolved into a compelling platform for high performance computing, as they offer both high computing power and memory bandwith for a reasonable price. Furthermore highly programmable GPUs become more and more available in end-user pc-systems, where they can be used as an additional processing unit.

In contrast to CPUs, GPUs have been specifically designed for compute-intensive highly-parallel computations with predictable memory accesses. This fundamental difference between CPUs and GPUs necessitates new programming systems, one of them being NVIDIAs CUDA.

CUDA offers the programmability of the GPU in the familiar C programming language in conjunction with a set of rudimentary libraries for e. g. GPU memory management. The current CUDA system works fine when the interface between the CPU part of the application and the GPU part can be written in plain C, but integrating it into a C++ application is far less straightforward. Furthermore transparent integration of CUDA into applications has up to now not been supported, and developers are required to explicitly move data from CPU memory to GPU memory and vice versa. Our novel CuPP framework allows developers to directly integrate CUDA into C++ applications and to write data structures that automatically manage data in both memory domains. Automatic memory management eliminates the need for developers to take care of reoccurring tasks, which by itself may not be difficult to solve, but nonetheless are a source of error. For example, Gillam [1] shows that there are various subtle details to consider when writing exception-save memory management code in C++. Based on the support for data structures, CuPP allows developers to define two independent representations for the same data, one of them for use at the CPU and the other at the GPU. CuPP translates between the two representations when data is moved from one memory domain to the other.

In this paper we demonstrate the benefit of using CuPP by integrating CUDA into an example application, OpenSteerDemo. OpenSteerDemo is the demo application of the OpenSteer steering library, where steering refers to life-like navigation of autonomous characters, so-called agents used for instance in computer games [2]. OpenSteerDemo offers multiple plugins each simulating different steering behaviors. We decided to work with the Boids plugin, which simulates flocks of birds. Working with multiple plugins would have been almost identical regarding memory management and CUDA integration, but would have required us to implement more steering behaviors with CUDA. From a performance viewpoint, the major component of Boids is the so-called k-nearest-neighbor problem (*k-nn*), in which for each simulated entity (*agent*) the k nearest neighbors are to be found. Based on their positions, each agent decides where to move next such that, e.g. collisions are prevented. The

existing Boids plugin uses a brute force approach to solve k-nn, meaning that each agent looks at all other agents to find its neighbors. We use the same algorithm in our first CUDA version, but implement a uniform grid in a second one. The grid implementation uses CuPP to internally work with two data representations, being optimized for CPU and GPU, respectively.

The paper is organized as follows. First, Section 2 gives a brief overview of CUDA and the problems that may arise when integrating it into C++ applications. The main part (Section 3) describes the functionality of the CuPP framework and explains how it solves the problems outlined before. Our experiences with the development of the CUDA-based OpenSteer plugins are described in Section 4. In Section 5 we give a brief overview of our experience with a CUDA port of the Einstein@Home application, in which we have found similar problems as in OpenSteerDemo – however, as the Einstein@Home client is a C application we did not use CuPP for this work. Section 6 discusses related work, while Section 7 summarizes the paper and gives a brief outline on future work regarding CuPP.

2. CUDA

CUDA is a general-purpose programming system for NVIDIA GPUs and was first publicly released in the end of 2007. By using CUDA, the GPU (called *device*) is exposed to the CPU (called *host*) as a co-processor. It executes a function (called *kernel*) in the SPMD model, which means that a user-configured number of threads runs the same program on different data. From the host's point of view, kernel invocations are asynchronous function calls. Synchronisation is done explicitly by calling a synchronisation function, or implicitly when the host tries to access memory on the device. In both cases, synchronization takes the form of a barrier that blocks the calling host thread until all previously called kernels have been finished.

The device is designed to process tens of thousands (or even more) threads at the same time. It uses this massive parallelism to hide the costs of memory accesses by efficient thread scheduling, i.e., threads are removed from a processor while waiting for a read from memory to complete. The memory at the device is called global memory and can be accessed by both the host and all processors of the device.

One of the major benefits of CUDA as compared to other GPU programming systems is its use of a C dialect, such that an original C function for a CPU can often be transformed into a CUDA kernel with only slight modifications. Furthermore CUDA provides to developers C libraries that expose all device functionalities needed to integrate CUDA into a C program. We give a brief outline of the basic functions next, a detailed description can be found in [3].

The so-called host runtime component of the CUDA library offers both `malloc` and `free`-like functions to allocate and free global memory. If a kernel should access the memory allocated with the CUDA malloc call, a pointer to this memory must be passed to it as a parameter. Furthermore the host runtime component offers a `memcpy`-like function to copy data from main memory to global memory or vice versa. As the device itself can not access main memory, but only memory located on the graphics board, global memory must be used to exchange data between host and device. Therefore an average CUDA program has the following work flow:

1) Allocate global memory (host)
2) Copy data from main memory to global memory (host)
3) Execute kernel and store the newly calculated values in global memory (device)
4) Copy calculated results from global to main memory (host)
5) Free global memory (host)

This approach is expected to work fine with all C programs. In C++, however, using memcpy is problematic at best, as copies of C++ objects are designed to have their copy constructor called at creation and the byte-wise copy done by memcpy is not guaranteed to be fully functional. The same holds true for objects containing pointers, since the pointer points to main memory, which is not accessable from the device. On the other hand, mixing C and C++ imposes some problems independent from the CUDA realm [4], [5]. For example, directly using a class in a C application requires to add a structure with the same internal memory layout as the class. It is possible to create such structures for C++ classes not using virtual inheritance, as their memory layout is identical to that of C structures. However this is only possible if the developer has access to the class definition. We propose CuPP as a solution to these problems and offer advanced mechanisms to ease the development for the host part of the application.

3. CuPP

CuPP is a C++ framework built up on CUDA and the Boost libraries. It can be roughly divided into five highly interwoven components that will be further explained below:

- device management
- memory management
- C++ kernel call
- support for classes
- data structures

Further information regarding CuPP can be found at the CuPP website [6] and more technical details of the implementation can be found in [7].

3.1. Device management

In CUDA a host thread can only control one device. CuPP device management is designed to remove this limitation.

Although in the current version of CuPP this functionality is not fully implemented, it is planned to be added in an upcoming version and therefore is already part of the existing API. The foundation of CuPP device management is a device handle that must be passed to all functions interacting with the device – e. g. allocating global memory or executing a kernel. The constructor of the device handle allows the client to specify which kind of device should be used. For example, the client may request a device with support for atomic operations. Furthermore the device handle can be used for barrier synchronization between CPU thread and GPU kernels.

3.2. Memory management

CuPP offers two abstraction levels for direct memory management. The lower level is used by CuPP itself and is similar to CUDA memory management except being adapted to C++. For instance the function allocating memory throws an exception if memory allocation fails and expects as parameter the number of elements to be allocated instead of the amount of bytes. One of the major benefits as compared to CUDA is that CuPP provides a smart pointer implementation that takes care of freeing global memory if there is no further pointer referring to it.

Nevertheless, we hardly expect application programmers to directly use this functionality in their programs, since the higher-level functionality of memory objects is typically more convenient. These are objects managing data in global memory. If a memory object is created, it automatically allocates memory at the device. The memory can be filled with data by passing a STL conform iterator or a pointer to a structure stored in main memory. When the memory object is destroyed, global memory is freed; when the memory object is copied, a new block of global memory is allocated and data is transferred from the old memory object to the new one. The behavior of copying memory objects may look uncommon at first, but is useful to implement data structures that automatically store their data on both host and device. A copy of an object is expected to have its own copy of the data instead of referring to the old one. When using memory objects, the developer must just take care that the copy constructor is called. The vector implementation described in section 3.5 and the grid data structure described in section 4 use memory objects.

3.3. C++ kernel call

With a CUDA kernel call, only C-conform data types can be passed. CuPP provides a C++-like function call including support for call-by-value and call-by-reference semantics, in a way that only parameters passed by reference can be changed by the kernel. The CuPP kernel call is implemented as a functor, which is a class providing `operator()`. The CuPP kernel functor uses `operator()` to issue the CUDA kernel call. A kernel functor object is created by passing a function pointer of the CUDA kernel function to the constructor. The number of threads, which should execute the kernel, can be passed to the kernel constructor as well, but can also be set afterwards.

Specifying if a parameter to the kernel should be passed by value or by reference is identical to the C++ syntax and all parameters to be passed by reference are prefixed with an ampersand. This is possible, since all CUDA files are preprocessed with a C++ compiler, which transforms references into automatically dereferenced pointers.

The call-by-value variant is almost identical to the standard C++ mechanism – parameters are passed by executing the following four steps:

1) A copy of the parameter is generated by calling its copy constructor (host)
2) The generated copy is byte-wise copied to the kernel stack on the device (host)
3) The kernel is executed (device)
4) The destructor of the generated copy is called after the kernel is called (host)

This method differs from the one of C++ only by the time the destructor is called. The destructor is called not after the kernel is executed, but may be called before. This is done to prevent any unneeded synchronisation between host and device. We are not aware of any case in which synchronization is required, as memory transfers from and to device memory are not done when a kernel is active and the destructor can therefore not influence the kernel in any way.

Call-by-reference is as well implemented in a similar way to C++:

1) The parameter to be passed to the kernel is byte-wise copied to global memory (host)
2) The address of the parameter in global memory is passed to the kernel (host)
3) The kernel is executed (device)
4) The copy in global memory is byte-wise copied back to main memory overwriting the original object (host)

Step four includes a synchronisation between device and host as global memory is accessed. To prevent unneeded synchronizations and to reduce the number of memory transfers, the CuPP kernel analyzes the kernel declaration and omits the last step for any references defined as constant using the `const` keyword. This analysis is done using the boost function traits [8] and self-written template metaprogramming code. The fact that a reference is declared as constant does not necessarily mean no data will be changed at the device – e. g. the developer may remove the constant with a cast later – but we expect this to be the exception and bad programming style in general.

3.4. Support for classes

The techniques described in the last section offer a more C++-like way of calling kernels, but it is still impossible to pass a data structures containing a pointer to a kernel. This problem results from the fact that each parameter passed to the kernel is byte-wise copied, so pointers are no longer useable at the device. To solve the problem, both the call-by-value and the call-by-reference behaviors can be enhanced by defining the following three functions as member functions of the object to be passed as a parameter:

- `T transform (device_handle&)`
- `device_reference<T> get_device_reference (device_handle&)`
- `void dirty (device_reference<T>)`

If it is not possible to change the class definition and add these functions, it is also possible to specialize a CuPP template and have non-member functions called instead. For now, `T` should be considered to be the same type as the class in which it is defined. The function `transform()` is called by the framework, when an object of this class is passed by value to a kernel. This function may be used to copy additional data to global memory and must return an object, which can be byte-wise copied to the device. For example, when considering an object containing pointers the `transform()` function can be used to transfer additional data to global memory and return an object with adjusted pointers, so that they are valid at the device.

`get_device_reference` is called by the framework when the object is passed by reference and has to return an object of type `device_reference <T>`. `device_reference <T>` is a reference to an object of type `T` located on the device. When created, it automatically copies the object passed to its constructor to global memory. The `device_references` are needed, as the CUDA kernel expects a pointer for all parameters passed by reference, so the parameters must be stored in global memory and a pointer to that memory must be passed to the kernel. CuPP provides for a default implementation, which creates a `device_reference` for the object returned by `transform()`. It is expected that the default implementation is sufficient in most cases.

Objects passed to a kernel as a non-`const` reference can be changed by the device. These changes have to be transferred back to the object on the host side and the object stored in the host memory has to be notified. This notification is done by calling the `dirty()` function of the object stored in main memory. A `device_reference` to the changed object in global memory is passed to this function, so the changed data on the device can be accessed and can be used to update the old host data.

It is optional to implement any of these three functions. The CuPP framework employs template metaprogramming to detect whether a function is declared or not. If it is not declared, a default implementation that behaves identical to the call-by-value and call-by-reference semantics is used. We expect that the functions `transform()`, `get_device_reference()` and `dirty()` must only be implemented when pointers or type transformations – as described next – are used.

The above described functionality allows to directly pass structures containing pointers to a kernel, however it is still not possible to pass an arbitrary class. We introduce a CuPP feature called type transformations to circumvent the problem. It is not possible to directly solve it due to CUDA limitations itself. The CuPP type transformations allow the client to define two independent types that get transformed into one another when transferred from one memory domain to the other. The type used by the host is called *host type*, whereas the type used by the device is called *device type*. The matching is expressed by adding two `typedef`s to both types, which name the host- and device types, respectively. If it is not possible to change the class definition of one of the types, the binding between host and device can be established by specializing a template.

The matching between the two types has to be a 1:1 relation. The transformation of the two types is not only done if the objects are passed by value but also if the objects are passed by reference. The transformation has to be done by the `transform()`, `get_device_reference()` and `dirty()` functions and the type denoted as `T` in the function definitions must be the device type. The host type can be any legal C++ class and the device type must be CUDA compliant. The next section gives an example of how the type transformations can be used.

3.5. Data structures

CuPP currently only includes a vector data structure, other data structures will be included in an upcoming version. The vector implementation is mostly a wrapper class of the C++ STL vector [9] and provides for an example implementation of a feature called *lazy memory copying*. The host type of the vector offers the same functionality as the C++ STL vector. The device type suffers from the problem that it is not possible to dynamically allocate memory in a CUDA kernel, so the size of the vector cannot be changed at the device. The type transformation is not only done to the vector itself, but also to the type of the values stored in the vector. Therefore the device type of `vector<T>` is identical to `vector<T::device_type>::device_type`. This kind of transformation makes it possible to pass e.g. two- and multidimensional vectors (`vector< vector<T> >`) to a kernel.

As vectors are often used to store a large amount of data, transfers of vectors from or to global memory should be minimized. CuPP vectors use lazy memory copying to reduce the amount of copies done. The technique is based

on some special behavior of the functions `transform()`, `get_device_reference()` and `dirty()`.

- `transform()` and `get_device_reference()` copy the vector data to global memory if the data is out of date or no data has been copied to the device before.
- `dirty()` marks the host data to be out of date.
- Any host functions or operators that read or write data check if the host data is up to date, and if not copy the data from the device.
- Any host functions or operators changing the state of the vector mark the data on the device to be out of date.

Using this concept, the developer may pass a vector directly to one or multiple kernels, without the need to think about how memory transfers are minimized, since the memory is only transferred if it is really needed. The example application described in the next section uses this technique to improve performance.

4. OpenSteerDemo

OpenSteer is a steering library written in C++ by Reynolds in 2002. *Steering* refers to life-like navigation of so-called agents. An *agent* is an autonomous character used in games or other interactive media. The navigation of an agent is defined by a *steering behavior*, which depends solely on the environment of the agent. OpenSteer offers a demo application, called OpenSteerDemo, which we use throughout our work. OpenSteerDemo offers different types of plugins simulating steering behaviors. To demonstrate the benefits of using CuPP, we have integrated CUDA into the Boids plugin, which simulates flocking of birds [2].

The architecture of OpenSteerDemo is similar to the one of games. It runs a main loop, in which each iteration simulates a discrete time step (*update stage*) and then draws the new state to the screen (*draw stage*). In a parallel variant of the application [10], which we take as our basis, the update stage is again split into a *simulation substage* and a *modification substage*. The simulation substage calculates the next agent state for all agents, but does not change the current states. The agent states are afterwards updated in the modification substage. In both the simulation and modification substages, the calculations for every agent can be executed in parallel and only a barrier synchronization is required between both substages.

Our CUDA implementation uses a thread for each simulated agent and two kernel calls, one for each substage of the update stage.

The simulation kernel takes position and direction of every agent as input parameters and outputs a steering vector, which represent how the agents wants to move. The modification kernel takes the steering vectors as input and updates position, heading and some misc. data – e. g. acceleration during the last simulation step – for every agent. Furthermore the modification kernel outputs matrices representing the position where the agents should be rendered at the screen. We are using multiple CuPP vectors to store all data named before – for example we have a vector for position data and one for the heading of agents. Each entry of a vector stores the data for one agent. This design is almost identical to that of the CPU-based OpenSteer plugin, except that the latter uses STL vectors instead of the CuPP ones.

To be able to access all agent data uniformly at both host and device, we only needed to replace the STL vectors with CuPP vectors and define three device types – one for the class storing the misc. agent data, one for the matrix, and one for the (mathematical) vector implementation of OpenSteer. Replacing the STL vector with the CuPP one did not harm compatibility with the original OpenSteer implementation and switching between the CPU and the device implementation can be done easily. Moreover, we still use all not performance-critical functions of the original CPU plugin – for example, the reset of the plugin, which sets all agents to random coordinates within the world is still the original CPU code. From a developer viewpoint, data stored in a CuPP vector is transparently accessable at both the host and the device.

From an internal viewpoint, an OpenSteerDemo run has the following data flow:

1) The agents are created by storing initial data in the CuPP vectors.
2) Prior to the CUDA kernel call, the data stored inside the vectors is automatically transferred to the device. The vectors storing the results of the kernel calls must be resized to the correct size before the kernel call, as memory cannot be allocated inside a CUDA kernel.
3) In the CPU-based draw stage, the render positions are transferred to main memory when the vector is first accessed.

Steps 2 to 3 are repeatedly executed until the simulation stops. Note that after the first two kernel calls, no data is transferred to the device, as the data is not changed by the host, and only the render positions are read. The data transfer of the render positions to the host is not done directly after the kernel call, but postponed until the data is needed. Since kernel calls are asynchronous and we have no implicit synchronization, everything done internally by OpenSteerDemo between our two kernel calls and the draw stage is done in parallel. The developer does not need to explicitly consider this parallelism between CPU and GPU, as it is automatically managed by CuPP and CUDA. Furthermore if a developer would remove the draw stage and output the result after some number of simulation steps, no memory transfer back to main memory would be issued in-between. The usage of CuPP vectors also allows a developer to easily check any data for debugging or testing purposes,

as the data inside a vector can be accessed at any time. If the debug code is removed unneeded memory transfers are automatically removed, as well.

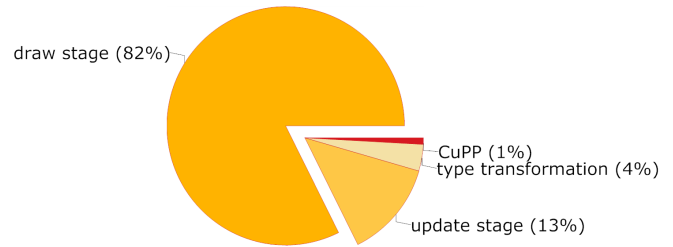
Moreover CuPP allows the developer to easily implement double buffering, so even more calculations are done in parallel at host and device. As the update stage is completely executed at the device, we can run the $n+1$-st update stage while the result of the n-th iteration is being drawn to the screen. This can be done by using two vectors storing the render positions of the n-th and $n+1$-st states, respectively. The vectors are swapped in every loop iteration, so that when state $n+1$ is drawn, the data of step n are overwritten with the data for step $n+2$. The swap must be done by the developer and includes a manual memory synchronisation before the kernels are executed. This synchronisation is necessary as otherwise the access to the vector during the draw stage would block the host thread until the previously called CUDA kernels have finished. The host thread is blocked since memory accesses to global memory block the calling host thread if a kernel is active, as already described in Section 2. The manual memory synchronisation will no longer be needed in an upcoming CuPP version, where we will provide a data structure with an interface explicitly designed for double buffering.

The CUDA plugin described above uses the same brute force algorithm to solve k-nn as the CPU implementation. To further improve the performance of the simulation, we changed this algorithm to utilize a grid data structure. A grid subdivides the world into small regions called *cells*. We refer to our grid implementation as a *static grid*. It subdivides the world into cubic cells of same size. The number of cells cannot be changed after a grid has been created and is identical for each dimension, so the overall shape of the static grid is a cube as well.

In a grid, agents are assigned to cells based on their current position, so a cell contains all agents within its range. The grid improves neighbor search performance, as an agent does not need to look at all other agents to find its neighbors, but only at those in cells within its search radius. The search inside these cells is done with the brute force algorithm, again.

Our static grid is created on the host and then passed to the simulation kernel as a parameter. We choose this built up as the device is designed for compute-intense tasks and generating the grid mostly consists of memory accesses. The creation of the grid is done before the simulation kernel is executed and redone for every simulation step. We use the CuPP type transformations to work with two different data representations.

The host type of the static grid is an aggregation of multiple C++ STL vectors, each representing a cell. A cell stores the indices of all agents within the cell. All cell vectors are stored in another vector, so the grid itself is a vector of vectors storing agent indices. The benefit of this approach is



Performance overview of the static grid based Boids simulating 4096 agents at about 102 frames per second, which results in 204 kernel calls per second.

Figure 1. Performance overview

that adding elements to the grid is a $O(1)$ operation. To add an element we must calculate the index of the cell vector and append the element.

Transferring one large memory block to global memory is to be preferred over transferring multiple smaller blocks, as CUDA memory transfers are DMA transfers with a rather high initialization cost. Based on this knowledge, we designed the device type so that it only consists of two linear global memory blocks. One of them contains the data of all cell vectors ordered by their index, i.e., all data from cell vector 0 followed by all data from cell vector 1 and so on. The other memory block contains the indices to locate the beginning of different cell vectors within the first memory block, i.e., the first value in this memory block is 0, the next is the size of cell vector 0, the next the size of vector 0 plus the size of vector 1 and so on. This device type requires only two memory transfers for all data, whereas a direct transfer of the host type to global memory would require one memory transfer per cell.

By using CuPP for implementing the static grid, we could independently choose two rather easy and straightforward data structures, whereas devising a single structure with support for both easy filling and fast transfers would have been more complicated. Even though such an implementation may provide a faster overall performance, we believe that for projects not striving for maximum performance, but looking for good performance with reasonable programming effort, the CuPP approach is a good compromise.

Measuring the CuPP runtime overhead compared to solely using CUDA is hardly possible, as CuPP consists of a high amount of rather small functions, which are inlined by default. Nonetheless, Figure 1 provides a rough performance overview of OpenSteerDemo when compiled *without* function inlining, so it is possible to measure the performance cost of these functions with a profiler. When compiling without function inlining, the performance of the update stage is reduced by more than 7% and the overall performance of the application is reduced by about 2%; therefore the performance numbers should only be considered a rough estimation. We measured the performance of our plugin us-

ing the static grid in a scenario with only 4096 agents, so the calculations done by the GPU don't become the performance bottleneck. In this scenario CuPP required less than 1% of the overall runtime and was capable of issuing more than 200 kernel calls per second. The type transformation itself requires about 4% of the runtime, but there is no runtime overhead compared to manually transforming the data structure before the kernel call. However, the type transformations impose a rather high compile time overhead as determining the corresponding types uses template metaprogramming – compiling the original Boids plugin required about 3.1 seconds, whereas compiling the CuPP based plugin requires 7.3 seconds.

Using the original CPU data structure is not only problematic for our grid implementation, but we also have the same problem in the implementation of another CUDA-based application, the Einstein@Home client. We give a brief outline of the problems with this application in the next section.

5. Einstein@Home client

Einstein@Home is a distributed computing project searching for so-called gravitational waves emitted from particular stars (*pulsars*), by running a brute force search for different waveforms at a huge dataset. The calculations done by the Einstein@Home application can be roughly divided into two parts. The first part is the so-called F-statistics, and the second is a Hough transformation. We only ported the F-statistics to CUDA.

The data structures used by the CPU-based Einstein@Home client are deeply-nested and pointer-based. We tried to directly copy these structures to the device, however this increased the runtime of the application by about a factor of 100. A new data structure, which arranges all data in one linear memory block and uses a second one with indices, similar to the one described above, reduces the time required for memory transfers by a factor of about 5000. Moreover, with the new data structure, it was finally possible to reduce the overall runtime of the application to about one third of the original CPU runtime, even though the kernel running at the device is hardly more than a copy-and-paste of the original CPU code. CuPP is not used in this work as the Einstein@Home client is written in C and not C++. Nonetheless, our experience with the Einstein@Home client shows that the problems experienced with OpenSteerDemo are not unique to this application and the type transformation solution offered by CuPP would also work well in the Einstein@Home application. A more detailed description of our work on the Einstein@Home client can be found in [11].

6. Related work

NVIDIA provides an example of how CUDA can be integrated into a C++ application as part of the CUDA SDK, however it discusses none of the problems outline above and expects the interface between CUDA and C++ to be C compliant. The CUDA Data Parallel Primitives Library (*CUDPP*) [12] designed by Harris et.al. tries to improve programmability by providing a set of kernel implementations for data-parallel algorithms such as parallel-prefix-sum or parallel-reduction. In contrast to CuPP, this approach frees the developer of writing a CUDA kernel, but host code and data transfers must still be managed by the developer.

7. Conclusion / Future work

In this paper we have introduced our novel framework CuPP, which is designed to ease the integration of CUDA into C++ applications. CuPP provides both predefined data structures that transparently support access to their data at GPU and CPU, and an interface to easily write own data structures with the same functionality. These data structures work with any kind of data flow and free the developer from taking care when the memory between the host and the device should be synchronized. Furthermore CuPP allows direct CUDA integration into C++ applications, which is complicated without CuPP when there is no C compatible interface between the two parts. As we have demonstrated in Sections 4 and 5, directly using a CPU data structure is problematic and may sometimes completely prevent any performance increase. The CuPP type transformations are designed to overcome this problem, by providing the possibility to easily use two independent data representations for the same data.

We have already noted some future developments for CuPP in both Sections 3 and 4, however there are some more future developments not yet named. We plan to extend the device management by adding a system that keeps track of the devices currently being used. Moreover CuPP would benefit from an advanced kernel queue that automatically distributes multiple kernels across available devices including transparent moves of data between the devices. Implementing such a system would require dataflow analysis to determine how data is passed to all kernels as well as the development of non-trivial efficient scheduling algorithms. Upcoming versions of CuPP may also support texture memory, so data can be easily stored not only in global but also in texture memory.

A problem when using CuPP is that it is based on a high amount of template metaprogramming. Since template metaprogramming is carried out during compilation, the compile time is increased – in the case of the Boids scenario, the compile time was more than doubled with the introduction of CuPP. Due to restrictions of the standard

C++ language, e. g. missing reflections support, template metaprogramming must be used to analyze the kernel definitions and we cannot avoid this problem.

Acknowledgment

We are grateful to Claudia Fohry and Michael Lesniak for proof reading and all members of the moderated C++ newgroup (comp.lang.c++.moderated) for some enlighting template metaprogramming snippets. We thank NVIDIA for providing hardware for testing and developing CuPP.

References

[1] R. Gillam, "The anatomy of the assignment operator," pp. 305–336, 2000.

[2] C. W. Reynolds, "Steering behaviors for autonomous characters," in *Proc. Game Developer Conference*, 1999, pp. 763–782. [Online]. Available: http://www.red3d.com/cwr/papers/1999/gdc99steer.html

[3] N. Corporation, "NVIDIA CUDA compute unified device architecture programming guide version 2.0," NVIDIA Corporation, 2008.

[4] S. Meyers, *More effective C++: 35 new ways to improve your programs and designs*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] M. P. Cline, M. Girou, and G. Lomow, *C++ FAQs*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[6] "CuPP website," http://www.plm.eecs.uni-kassel.de/plm/index.php?id=cupp, 2008. [Online]. Available: http://www.plm.eecs.uni-kassel.de/plm/index.php?id=cupp

[7] J. Breitbart, "A framework for easy CUDA integration in C++ applications," Diplomarbeit, University of Kassel, 2008.

[8] Adobe Systems Inc, D. Abrahams, S. Cleary, B. Dawes, A. Gurtovoy, H. Hinnant, J. Jones, M. Marcus, I. Maman, J. Maddock, T. Ottosen, R. Ramey, and J. Siek, "Boost type traits library," http://www.boost.org/doc/html/boost_typetraits.html, 2008. [Online]. Available: http://www.boost.org/doc/html/boost_typetraits.html

[9] B. Stroustrup, *The C++ programming language, fourth edition*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[10] B. Knafla and C. Leopold, "Parallelizing a real-time steering simulation for computer games with OpenMP," 2007, parCo.

[11] J. Breitbart, "Case studies on gpu usage and data structure design," Master's thesis, University of Kassel, 2008.

[12] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "CUDPP: CUDA data parallel primitives library," http://www.gpgpu.org/developer/cudpp/, 2008. [Online]. Available: http://www.gpgpu.org/developer/cudpp/