University of Pisa
&
Scuola Superiore Sant'Anna
Master Degree in Computer Science and Networking

Master Thesis

Optimization of Parallel Computations on Heterogeneous GPU-Based Systems

Candidate
Claudio Parisi

Supervisor
Marco Danelutto

Co-supervisor
Christoph Kessler

Academic Year 2014/2015

PREFACE

This work has been done in the framework of cooperation existing among the department of Computer Science of the University of Pisa, Italy (Prof. Danelutto) and the IDA department of the Linköping University, Sweden (Prof. Kessler).
It originated with the main goal of addressing a well know issue related to GPU programming: the loss of performance due to data transfers between main memory and device memory.
All known solutions tackling this problem often require a considerable amount of hours of additional work for the application programmer to be designed and implemented. This is the main motivation that led to this master thesis, where we have extended the existing data-parallel skeleton library SkePU to offer a fully transparent solution which has been synthesized in just one more compile flag.
The result of this work has eventually contributed to the new release of SkePU itself (v1.2) which is now both faster in executing data-parallel computations, and better optimized to support task-parallel computations on-top of it.

ACKNOWLEDGMENTS

My deepest gratitude goes to both my supervisors: Professor Marco Danelutto and Professor Christoph Kessler. The former to present me with the opportunity of a master thesis abroad and for the endless afternoons spent in video-conference analyzing my work and suggesting further ways to progress. The latter for actually making that opportunity a reality, giving me a warm hospitality in his team and following every step of my work providing plenty of ideas and, most of all, knowledge.

A special thanks goes to Antonella Mercurio, both for being my mother and for the financial support of my entire period abroad: without her I couldn't have achieved any of this.

Last but not least, thank you to my love Alessia, my “brother” Mike, Cesira, RHF, Patrik, Carlo and Davide: you simply are the best part of my life.

Contents

1. Introduction

1.1 Motivation

In the past fifteen years, the IT industry evolution has been characterized by the introduction and growth of parallelism. This new approach is gradually substituting the serial one in almost every area: the semi-conductor industry, since 2003, proposes multicore CPUs to keep up with the Moore's law, a completely new generation of manycore chips has been introduced mainly in the form of co-processor (e.g. Intel Xeon Phi) and the old Graphic Processing Unit architecture has been reviewed to support General Purpose computations (GPGPU) on thousand of cores simultaneously. But what may seem a relatively linear evolution on the hardware side, becomes a continuous challenge for programmers. The newborn *heterogeneous systems* require a deep knowledge of all the composing architectures, of their related programming model, and of all the different ways they may interact with each other to squeeze out every last bit of performance they have to offer. This rightfully is an impossible task for the programmer which tends to specialize himself on a very small subset of those architectures.

Introduced by Murray Cole in 1989 [9], *algorithmic skeletons* constitute a high-level parallel programming model which abstracts both the computation and the coordination parts of the most recurring patterns of a structured parallel application, providing the programmer with a generic sequential looking interface.

An algorithmic skeleton based parallel programming approach directly influences three software-related properties, strictly linked each other:

- **Programmability.** Exposing a sequential interface, the application programmer can use the skeleton as a more familiar sequential component and focus his efforts on the actual application logic.
- **Portability**. Being an interface, an algorithmic skeleton masks the actual implementation of the parallel pattern it models. As a consequence, a given algorithmic skeleton can be adopted in a platform-independent manner, provided that an implementation exists for each of the targeted platforms.
- **Performance**. The implementation of a skeleton, for each of the supported

platforms, can be tweaked to exploit at their best all the features of the underlying architecture.

As such, algorithmic skeletons may seem a perfect fit for the current IT scenario and, in fact, research groups all over the world are proposing several frameworks and libraries based on the algorithmic skeleton concept [8]. But, as with any high-level abstraction, advantages can't come without tradeoffs: as the programmability improves and the portability broadens, it gets more and more difficult to aim at the best possible performance.

1.2 Thesis Target

In this master thesis we consider SkePU, a data-parallel skeleton library targeting GPGPUs, which offers both a wide portability and an easy-to-program sequential interface. We analyze its performance on CUDA-based heterogeneous systems and propose an enhancement which addresses the latency increase that occurs when computation is offloaded to an external device such as a GPU.

In particular, our target was to decrease, or even better to completely eliminate, the time spent on transferring data between system memory and NVIDIA GPUs memory before the computation can actually begin. This is, in general, the highest source of overhead in heterogeneous systems, often leading to a decrease in performance even with respect to a sequential, CPU-only, execution. Depending on the implementation, the application programmer may spend hours trying to optimize his code or completely discard the usage of a co-processor if the outcome isn't worth the effort.

Exploiting the skeleton programming approach, we propose *MultiStream*, a generic optimization that adopts a pipeline pattern implemented on top of CUDA Streams. This gives SkePU the possibility to overlap data transfers with kernel executions thus decreasing, and in some cases completely masking, the overhead due to the former. This solution is applied directly to every skeleton implementation, is completely backward compatible, and has little to no impact on the level of abstraction guaranteed to the application programmer.

On the results side, *MultiStream* is able to increase the performance of every skeleton execution and its adoption is suggested with very few exceptions which depend, as will be shown, on specific and uncommon conditions. Moreover, using the stream-parallel skeleton framework FastFlow [7], we show how SkePU and *MultiStream* can be efficiently used to compute data-parallel portions of code nested in a larger stream-parallel application.

1.3 Thesis Outline

The rest of this paper is organized as follows:

- Chapter 2 describes CUDA, SkePU and all the technical background required to understand the remainder of this master thesis.
- Chapter 3 provides an in-depth analysis of the problem and the description of the *MultiStream* design and implementation. A cost model is derived for every skeleton implementation.
- Chapter 4 is reserved to the *MultiStream* performance analysis.
- Chapter 5 and 6 conclude this master thesis and discusses possible future enhancement for *MultiStream* and SkePU.

2. Background

This chapter introduces the technical background that may help the reader in understanding the contents of this master thesis. First, NVIDIA CUDA Architecture is described to underline advantages and tradeoffs relative to GPGPU usage. Afterward, SkePU is introduced as a high-level library able to provide a skeleton programming approach to GPGPU programming.

2.1 NVIDIA CUDA

2.1.1 A Brief History of GPU Computing

In the early 1990s, thanks to the growth in popularity of graphically driven operative systems (e.g. Microsoft Windows), personal computers started to be equipped with 2D display accelerators which offered hardware-assisted bitmap operations.

In 1992, Silicon Graphics released the OpenGL library, the first 3D application programming interface. The most popular were the first-person shooter games such as Doom or Quake, which ignited the spark that led to the creation of progressively more realistic 3D environments and, consequently, increasingly more powerful graphics accelerators.

The first attempts to provide a general purpose graphic parallel computing started in 2001, with the introduction of programmable vertex and pixel shaders by GPU manufacturers. In those years, researchers cleverly tricked the graphic accelerators into performing arbitrary computations on generic numerical data by making them appear as operands required to standard graphic rendering. This approach was restrictive because applications had to be written with graphic-only programming languages and there was no method to debug any code upon incorrect results or failures.

In November 2006, with the release of the GeForce 8000 GPU series, NVIDIA introduced the first general-purpose processing GPU built with CUDA Architecture.

2.1.2 CUDA Architecture

CUDA Architecture, and GPGPUs in general, provide the application programmer with a high throughput co-processor best suited for massively parallel computations.

With respect to a traditional multicore CPU, graphics accelerators make a different use of their chip area. In particular, features such as out-of-order instruction execution, speculation or branch prediction are completely absent, interprocessor communications are bound to hardware restrictions and cache memory is one order of magnitude smaller. All of these limitations leave enough area to fit hundreds of *Streaming Multiprocessors* (up to 384 on the Maxwell architecture released in 2015) which are, in turn, groups of 8 execution units, called CUDA Cores.

On GPU, performance is achieved through the SIMT architecture (Single-Instruction, Multiple-Thread) which enables a high degree of parallelism [1: 4.1]. It consists of a large number of threads sharing the same kernel which, compared to the CPU ones, have zero context-switching overhead and are suitable to operate on extremely fine-grained datasets.

Threads are organized using a two-level hierarchy [Figure 1]. At the bottom level, they are grouped into *Thread Blocks* of 1, 2 or 3 dimensions. Let us call (x, y, z) the index of a specific thread inside given thread block and (D_x, D_y, D_z) the size of that thread block, the thread can be identified by $threadIdx = x + y D_x + z D_x D_y$. At the top level, thread blocks are organized into 1, 2 or 3 dimensional *Grids* and are identified by an ID *blockIdx* similar to the thread one.

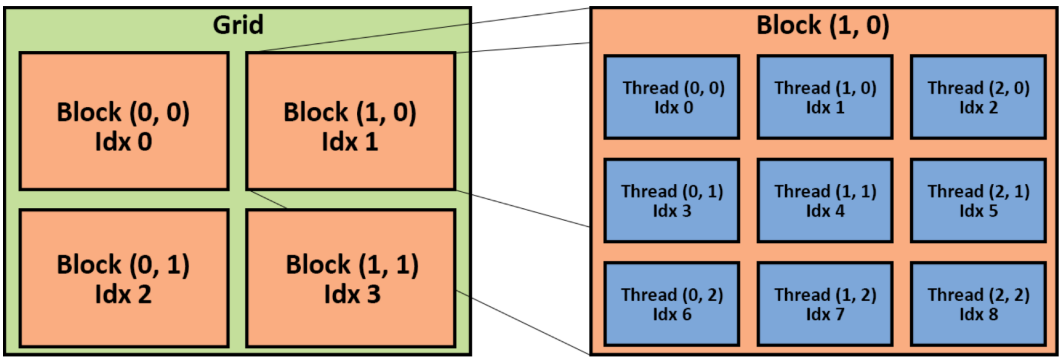

Figure 1: Thread Hierarchical Organization

Upon kernel execution, each multiprocessor gets a thread block enabling a first level of concurrency. Each multiprocessor decomposes the received thread block into groups of 32 parallel threads called *warps* which are then managed by the CUDA cores, enabling a second level of concurrency. In particular, supposing that a given instruction, common to all the threads inside a warp, takes 1 clock cycle to be executed, it will take a total of 4 clock cycles $(WarpSize / CUDACoresPerSM)$ to complete that instruction for the whole

warp. Threads inside a warp start at the same instruction address but they all have their own program counter and register state. This leaves branching and independent execution possible although, each time a thread follows a different path of execution, it gets serialized and concurrency is partially lost.

For what concerns memory management, the CUDA Architecture is composed of different types of memory, each with their own advantages and drawbacks [1: 2.3]. We will focus our attention on the subset which is actually important to the remainder of this master thesis. [Figure 2] shows the hierarchy for the Kepler architecture. Each GPU generation introduces some changes, but these differences are negligible in our analysis.

- *Constant memory* is a 64 KB read-only portion of the device memory which can be cached in an 8 KB memory present in each SM. It enables warp level broadcast.
- *Shared memory* is a programmer managed portion of the L1 cache reserved available to each SM. Can only be shared among threads of the same thread block.
- *L2 cache.* Both accesses to the device memory and system (*host*) memory pass through L2 cache.
- *Global memory* is roughly equivalent to the device memory (except for the portion reserved to constant, texture and local memories), it has the highest latency on device but provides up to 15 times higher bandwidth with respect to the PCIe interface that links the device L2 cache to the host memory. It is implemented with GDDR5 technology up to Maxwell architecture, Stacked DRAM will be introduced with Pascal in 2016.

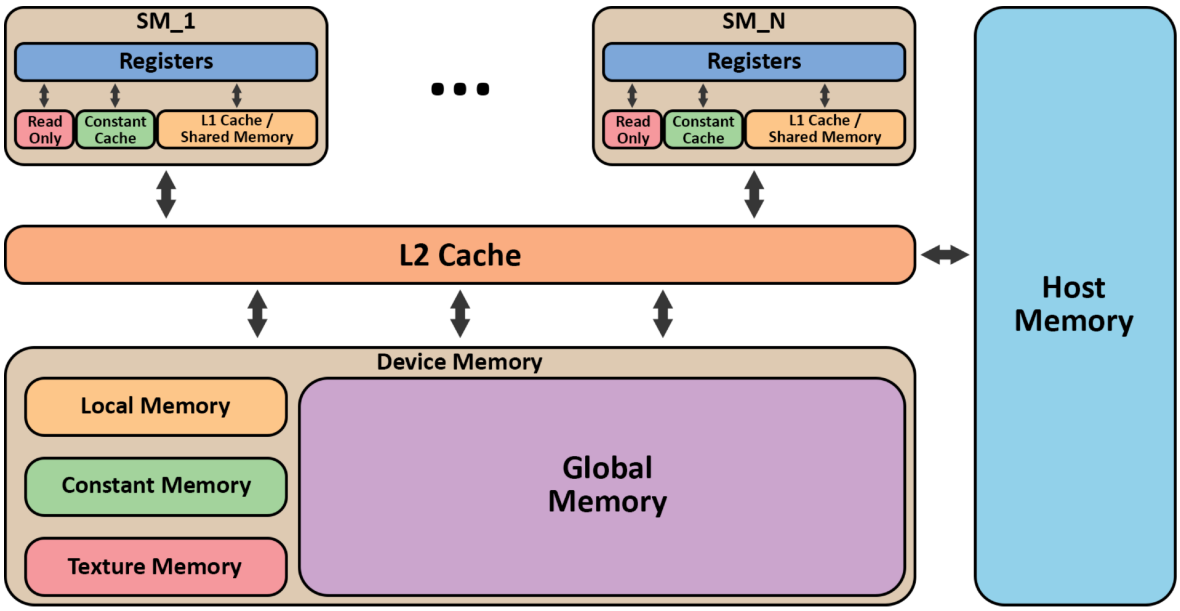

Figure 2: Kepler Memory Hierarchy

2.1.3 CUDA Parallel Programming

The first version of CUDA was based of ANSI C. With new releases, it has eventually evolved to a more complete extension of C++11 (CUDA version 7 or greater) [1].

A CUDA application can be divided in two portions: one containing the *host* code and one reserved to the *device* code. Both portions are compiled by the NVIDIA compiler *nvcc,* which internally uses a conventional C/C++ compiler for the host code, and a proprietary compiler which translates the device code to an assembly language called PTX (Parallel Thread Execution). A PTX application describes the execution of a thread inside a Cooperative Thread Array (CTA), including possible communications and synchronizations required between threads in the same CTA. In other words, a CTA is the implementation of a thread block.

Upon execution, there's a first phase called *install time* which translates the PTX code to the GPU one. Subsequent executions of the same application, in the same session, will skip this step. The actual execution then starts on the host thread which is in charge to transfer the required input data to the GPU device memory before invoking the GPU kernel code. If data are already on the GPU (e.g. as a result of a previous computation), only the kernel code will be transferred and executed. Upon completion, output data need to be transferred back to the host memory before they can actually be accessed.

As will be shown in the first part of Chapter 3, Host-to-Device (*HtD*) and Device-to-Host (*DtH*) memory transfers constitute one of the narrowest bottleneck in GPGPU because of the limited bandwidth provided by the PCIe interface that connects the GPU to the hosting system.

2.1.4 Example of a Simple CUDA Application

As an example, we consider the following code which performs the sum of two vectors:

```
01 // Device code part
02 __global__ void sum(float *v1, float *v2, float *vres){
03    // Each thread is responsible of the computation of one element
04    int i = blockIdx.x * blockDim.x + threadIdx.x;
05    vres[i] = v1[i] + v2[i];
06 }
07
08 // Host code part
09 int main(){
10    float v1[N], v2[N], vres[N];
11    float *dev_v1, *dev_v2, *dev_vres;
12
13    // Allocate memory on the GPU
14    cudaMalloc((void**)&dev_v1, N * sizeof(float));
15    cudaMalloc((void**)&dev_v2, N * sizeof(float));
16    cudaMalloc((void**)&dev_vres, N * sizeof(float));
17
18    /* Populate vectors v1 and v2 */
19
```

```
20    // Copy the vectors on the device memory
21    cudaMemcpy(dev_v1, v1, N * sizeof(float), cudaMemcpyHostToDevice);
22    cudaMemcpy(dev_v2, v2, N * sizeof(float), cudaMemcpyHostToDevice);
23
24    // Define a thread block size and a grid size – OPTIMAL is a number
25    dim3 threadsPerBlock(OPTIMAL);
26    dim3 blocksPerGrid(N/threadsPerBlock);
27
28    // Invoke the kernel
29    sum<<<blocksPerGrid, threadsPerBlock>>>(dev_v1, dev_v2, dev_vres);
30
31    // Copy the result to the host memory
32    cudaMemcpy(vres, dev_vres, N * sizeof(float), cudaMemcpyDeviceToHost);
33
34
35    // Free device memory
36    cudaFree(dev_v1);
37    cudaFree(dev_v2);
38    cudaFree(dev_vres);
39
40    /* Print the result */
41
42    return 0;
43 }
```

The example shows that the application programmer has to take care of a lot of aspects:

- memory (de)allocation on the device
- copy of the vectors from the host memory to the device one and viceversa, when a result has been computed
- defining the thread block size and the grid size by fixing a value OPTIMAL.

The last point is crucial to achieve the best performance (e.g. having a few thread blocks with many threads may leave some of the stream multiprocessors idle) and is dependent of both the specific application and the *Compute Capabilities* (i.e. architecture and features) of the targeted device.

2.2 SkePU

SkePU is data-parallel skeleton programming framework targeting heterogeneous GPU-based systems (multicore CPU with one or more GPUs) [2]. It is developed and currently maintained by the group of Professor Christoph Kessler at the IDA department of the Linköping University, Sweden.

SkePU is provided as a C++ template library that exposes a unified interface to specify data-parallel computations expressed using algorithmic skeletons.

The library provides multiple implementations of each skeleton to target different architectures: single core and multicore CPUs are supported by sequential C++ and OpenMP respectively, whereas GPUs are supported using CUDA and OpenCL. To target a

specific architecture it is enough to define a macro SKEPU_*X* where X is the target architecture (e.g. `#define SKEPU_CUDA`). It is also possible to let SkePU automatically select the expected best back-end for each skeleton call. The choice is dependent on the size of the operands used in the computation.

At the time of writing, SkePU comes in two different distributions [5]:

- stand-alone SkePU 1.2 which is the one adopted and described into this master thesis. It includes the *MultiStream* optimization, the result of our research.
- SkePU with StarPU integration, a C-based run-time system with generic scheduling falicities for heterogeneous platforms. StarPU is used by SkePU to provide asynchronous and hybrid (i.e. CPU+GPU) skeleton executions. Each skeleton call is translated into a task and dynamically scheduled by StarPU following a user-selected strategy among those available.

The three main components of the library are user functions, skeletons and smart containers (1D vector and 2D matrix).

2.2.1 User Functions

In order to define functions that can be used as skeleton parameters regardless of the target architecture, SkePU provides the application programmer with a macro language that the compiler preprocessor expands to the specific implementation once a target architecture as been selected. Macro functions are based on a *struct* containing, in turn, member functions for CPU and CUDA, and strings for OpenCL [3: 3.1].

Let us consider the same application shown in the example of section 2.1.4. The function that computes the sum in the device portion of the code can be written as a *BINARY_FUNC* that takes two operands *a* and *b* and sums them:

```
BINARY_FUNC(sum_f, float, a, b, return a+b);
```

This function expands to:

```
struct sum_f {
   // OpenCL
   skepu::FuncType ft;
   std::string func_CL;
   std::string funcName_CL;
   std::string datatype_CL;
   sum_f() {
      ft = skepu::BINARY;
      funcName_CL.append("sum_f");
      datatype_CL.append("float");
      func_CL.append("float sum_f(float a, float b) { return a+b; }\n");
   }
   // CPU
   float CPU(float a, float b) { return a+b; }
   // CUDA
   __device__ float CU(float a, float b) { return a+b; }
};
```

To get a list of the available macros for each of the SkePU supported skeleton, please refer to the SkePU documentation [5].

2.2.2 Skeletons

SkePU provides six data-parallel skeletons: Map, Reduce, MapReduce, MapArray, MapOverlap (i.e. stencil) and Scan (i.e. parallel-prefix) [3: 3.1]. They are represented using C++ objects so, in order to use them, the application programmer has to instantiate the chosen skeleton with the user function to compute as parameter. Afterward, the skeleton function can be called passing the whole containers as operands, or iterators if the computation as to be applied only on a subset of data.

Taking again into account the application of section 2.1.4 and applying the binary user function defined in section 2.2.1:

```
01 int main(){
02
03    // Initialize three vectors
04    skepu::Vector<float> v1(1000);
05    skepu::Vector<float> v2(1000);
06    skepu::Vector<float> vres(1000);
07
08    /* Populate vectors v1 and v2 */
09
10    // Instantiate the Map skeleton and perform the sum_f user function
11    skepu::Map<sum_f> sum(new sum_f);
12    sum(v1, v2, vres);
13
14    /* Print the result */
15
16    return 0;
17 }
```

As we can see the code complexity is significantly reduced: SkePU run-time support will take care of the memory management, will determine the best skeleton implementation and tune its parameters to maximize the device resources utilization [3: 3.2].

A brief description of the available skeletons follows:

Map. Given a function *f* and a vector (matrix) *v*, the Map skeleton computes a vector (matrix) *r* such that each element is $r_i = f(v_i)$. A `skepu::Map` can be initialized with a user function that takes up to three variable operands and a constant one. The CUDA implementation of the latter exploits the constant memory described in section 2.1.2.

Reduce. Given an associative, cumulative binary operator $\oplus$ and a vector (matrix) *v*, the Reduce skeleton computes a scalar result $r = v_1 \oplus v_2 \oplus \ldots \oplus v_n$. A reduce policy can be specified to perform column-wise or row-wise reduction of a matrix.

MapReduce. Combines the Map and Reduce skeletons by taking two user functions as parameter upon initialization. The computation is performed in the same kernel to avoid synchronization resulting in a speedup.

MapArray. Given a function *f*, a vector *A* and another vector (matrix) *v*, the MapArray skeleton is a variant of the Map skeleton that computes a vector (matrix) *r* such that each element is $r_i = f(A, v_i)$. This means that, whereas in the Map skeleton the user function has access to the *i*-th element of each operand to compute r_i, in the MapArray skeleton the first parameter passed to the user function is a pointer to the whole vector *A*, thus r_i can be computed as a function of v_i and any of the elements of vector *A*.

MapOverlap. Another variant of Map, where the *i*-th element of the result vector is computed by taking as user function operands several adjacent elements of the input vector. A constant parameter *d* (compile time defined) represents the radius of the neighborhood space of center *i* of the input vector to be taken into account in the user function: $r_i = f(v_{(i-d)}, v_{(i-d+1)}, \ldots, v_{(i+d-1)}, v_{(i+d)})$. The MapOverlap skeleton can also be applied to matrices and is provided with four different adjacency patterns: column-wise, row-wise, cross-wise or square-wise. The CUDA implementation makes use of the shared memory described in section 2.1.2, thus the parameter *d* is limited by both the number of threads per block and the amount of shared memory provided by the targeted GPU.
Scan. Given an associative binary operator $\oplus$, the Scan skeleton computes the prefix-$\oplus$ vector of the input operand: $r_i = v_1 \oplus v_2 \oplus \ldots \oplus v_i$.
The *MultiStream* optimization discussed in this master thesis can be applied to Map, MapReduce and MapArray skeletons.

2.2.3 Smart Containers

In order to support skeleton computations and optimize data transfers between host and device memories, SkePU provides two containers, Vector and Matrix, that can be used to organize data in 1 or 2 dimensions respectively [4]. Their interface is similar to their C++ STL counterparts, although extended with some member functions such as `randomize(min, max)` to assign a random value between [min, max) to each element of the vector (matrix), or `operator ~()` which can be used to transpose a matrix and provide the application programmer with column-wise access.
The implementation of the containers features two memory management optimizations.

Lazy memory copying delays the device to host memory data transfer until this is absolutely required. A speculative approach is applied to all the data that have been modified in the device memory by a skeleton computation: unless they are accessed by

the host thread (e.g. by reading the content of a vector element using the `[]` operator), there is no need to synchronize the copy residing in the host memory since these data could be an intermediate result that will be used as an operand of a new skeleton computation. This way the overhead due to the Device-to-Host transfer for intermediate results can be completely avoided.

[Figure 3] shows the memory management behavior that the following code would have with or without lazy memory copying:

```
01 …
02 // Initialize one vector
03 skepu::Vector<float> v(4000);
04
05 /* Populate vector v */
06
07 // Instantiate two different skeletons and execute them
08 skepu::Map<Kernel_1> k1(new Kernel_1);
09 skepu::Map<Kernel_2> k2(new Kernel_2);
10 k1(v);
11 k2(v);
12
13 // Print the result: the content of vector v is accessed by the host thread
14 std::cout << v << “\n”;
15 …
```

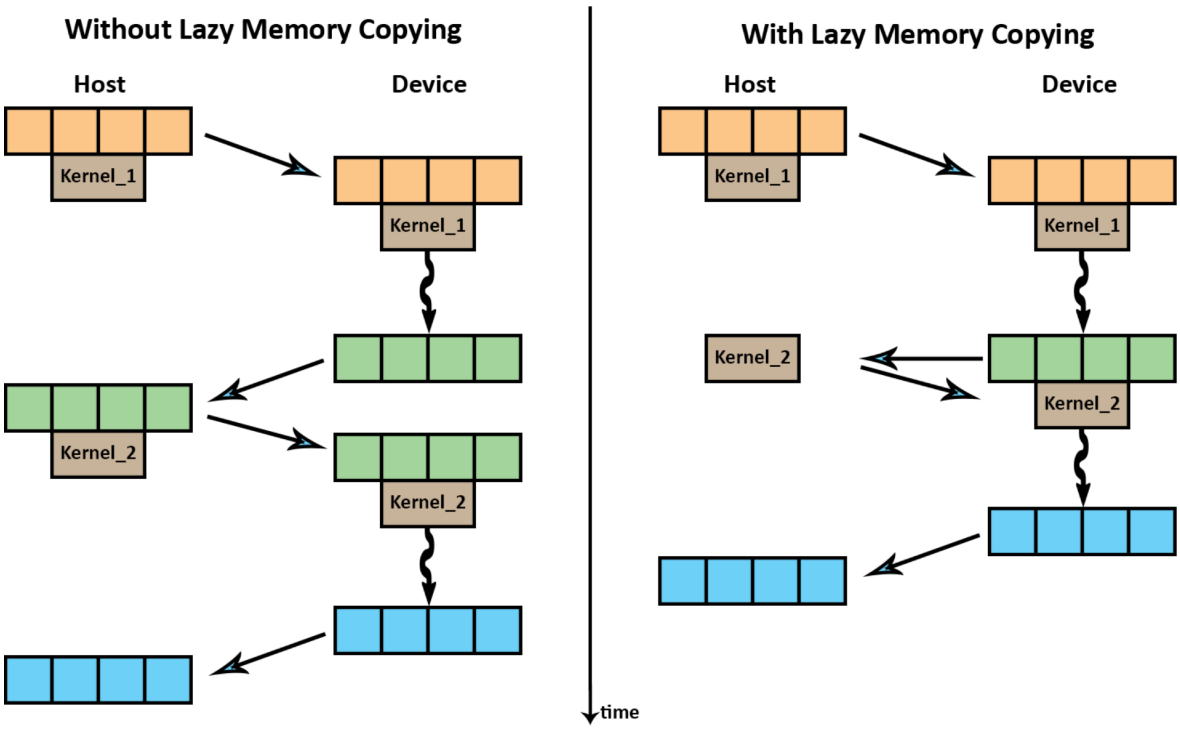

Figure 3: Lazy memory copying behavior

Subset transfers is the second memory management optimization that consists of splitting input and output data in all those cases where only a portion of them is required for skeleton computations (e.g. when work is distributed across multiple GPUs). Each subset is implemented as an additional device copy whose size is the one of the subset itself. Moreover it is coupled with a *valid* flag that tracks which portion has been modified and which one needs to be updated upon accessing. To reduce the overhead due to tracking and limit memory usage, a maximum amount of subsets is imposed.
As will be shown, the implementation of the *MultiStream* solution had to deal with both of these two features to avoid conflicts.

3. MultiStream

MultiStream is an optimization that targets the loss of performance due to data transfers between host and device memories in a GPU-based heterogeneous system. It has been designed and developed as a new feature for the data-parallel skeleton programming library SkePU, in the framework of cooperation existing among the department of Computer Science of the University of Pisa, Italy (Prof. Danelutto) and the IDA department of the Linköping University, Sweden (Prof. Kessler).

3.1 Problem Statement

In the early stages of the present work we were investigating the feasibility of using GPGPU to perform the data-parallel portion of computations of all those applications that could be expressed with a nesting of task-parallel and data-parallel skeletons.
If we run a quick search on how to optimize an application that makes use of a CUDA enabled GPU, we will surely find this result [1: 5.1]:

> Performance optimization revolves around three basic strategies:
> - Maximize parallel execution to achieve maximum utilization;
> - Optimize memory usage to achieve maximum memory throughput;
> - Optimize instruction usage to achieve maximum instruction throughput.

It is worth to further divide the second point into:

- Optimize the usage of the device memory;
- Minimize data transfers between host and device.

Providing the application programmer with a parallel skeleton programming framework, such as SkePU, should reduce the list of strategies to the last point which entirely depends on the logic of the application itself. The remaining advices should be handled by the implementation of each skeleton.
During the analysis of SkePU, in section 2.2, it has been shown how its skeletons implementations take care of achieving the maximum performance in the device scope, and how optimizations like *lazy memory copying* and *subset transfers* try to minimize communications between host and device. But would be these two optimizations enough in an application that continuously issues new tasks to the GPU from the CPU?

3.1.1 Best-Case Scenario Analysis – No MultiStream

The following example could model one of the typical behaviors of an application working on a stream of data produced on the host side. The *for-loop* is used to create new tasks that will be sent to the GPU for computation. The user function takes the result of the previous skeleton execution and reuses it as operand, coupled with the task produced on the host in the *i*-th iteration. The result is stored overwriting the *vres* vector.

```
…
skepu::Vector<float> task(1000);
skepu::Vector<float> vres(1000, 0);
skepu::Map<foo_f> foo(new foo_f);
for(uint i = 0; i < N; i++){ // N = number of tasks

   /* Populate the vector “task” */

   // Use task as second operand for the skeleton
   foo(vres, task, vres);
}
…
```

At steady state, the workflow of this application can be summarized in the following points:

a) Production of a new task by the host thread
b) Synchronous Host-to-Device transfer of vector *task*
c) Execution of the *foo* skeleton on GPU and release of the host thread
d) New iteration of the *for-loop*

It is worth to notice that between point (c) and (d) SkePU's lazy memory copying mechanism has prevented the useless Device-to-Host memory transfer of the *vres* vector. However we can still identify two sources of inefficiency: the keyword *synchronous* implies that the CPU has to wait for the Host-to-Device copy before issuing the kernel execution and start the production of a new task; the Host-to-Device transfer itself tends to be a bottleneck if we consider sufficiently large transfers over the limited bandwidth of the PCIe interface with respect to the potential throughput of a GPU device.

Let us now try to calculate the GPGPU portion of the completion time of this application.

We will adopt the following conventions:

$T_{comp}^{(CM1)}$ completion time without *MultiStream*;

Ts kernel service time;

T_{HtD}, T_{DtH} communication times required to perform a Host-to-Device or Device-to-Host transfer, respectively.

The cost model has to leave out the time required to produce a task by the host:

$$T_{comp}^{(CM1)}(N) = T_{HtD}(vres) + N * [T_{HtD}(task) + Ts] + T_{DtH}(vres)$$

Assuming *N* sufficiently large and $T_{HtD}(vres) \approx T_{HtD}(task) \approx T_{HtD}$, we obtain: $T_{comp}^{(CM1)}(N) = N * (T_{HtD} + Ts)$ or, normalizing for one task:

(1a) $T_{comp}^{(CM1)} = T_{HtD} + Ts$.

Considering the set of applications that operates on a stream of data produced on the host side, the application we have just analyzed exemplifies a best case scenario: for each task, we only send one operand to the GPU and wait for a kernel execution. The result is gathered from the host once we have exhausted all the tasks.

3.1.2 General Worst-Case Scenario

To better understand the benefits that the *MultiStream* optimization brings to SkePU, we need to analyze the cost of each of the supported skeletons in their worst-case scenario of utilization. In particular, we need to take into account the possibility of passing multiple operands to the skeleton responsible of computing each task, and we also have to consider the case in which each produced result is immediately needed on the host side for access or further computation.

The general workflow for the worst-case scenario can be summarized in the following steps:

a) Production of a new task by the host thread
b) Synchronous Host-to-Device transfer of one or more operands
c) Execution of the proper skeleton on the GPU
d) Synchronous Device-to-Host transfer of the computed result
e) Jump to point (*a*)

The GPGPU portion of this computation is represented by points (*b*, *c*, *d*).

In the following sections, we will define a cost model taking into account the actually used skeleton. In section 3.5, we will then make the same analysis with the addition of the *MultiStream* optimization which, as we will see, enables asynchronous operations over multiple CUDA Streams in order to overlap communication times and service times.

3.1.3 Smart Containers Representation in CUDA

Before moving to the cost model analysis of each skeleton, it is worth to spend a few words on how SkePU's smart containers are represented in CUDA. What follows is to be intended as a simplified version of the actual implementation which is spread across multiple pages of code and isn't required for the remainder of this master thesis.

Both the Vector and the Matrix smart containers are represented in CUDA as a one-dimension data-structure.

In particular

```
skepu::Vector<T> vect(N);
```

would translate to something like

```
T *host_vect, *dev_vect;
/* Allocate N*sizeof(T) bytes on the host memory pointed by *host_vect */
/* Allocate N*sizeof(T) bytes on the device memory pointed by *dev_vect */
```

The Matrix smart container follows the same schema

```
skepu::Matrix<T> vect(N, M);
```

would translate to something like

```
T *host_vect, *dev_vect;
/* Allocate N*M*sizeof(T) bytes on the host memory pointed by *host_vect */
/* Allocate N*M*sizeof(T) bytes on the device memory pointed by *dev_vect */
```

This common representation gives us the possibility to threat both smart containers as the same entity, thus every analysis that follows will be valid for both of them.

3.1.3 Map Cost Model – No MultiStream

As explained in section 2.2.2, the `skepu::Map` skeleton can take up to three input operands to produces one output. The cost of the GPGPU computation for one task can be written as $T_{comp}^{(CM1)} = kT_{HtD} + Ts + T_{DtH}$ with $1 \leq k \leq 3$ input operands that are actually transferred from the host memory to the device one.

As will be shown in section 3.2, the usage of page-locked memory can let us make the following assumption: $T_{comm}^{(CM1)}(N) \approx T_{HtD} \approx T_{DtH}$. This assumption will always be valid in the remainder of this master thesis since the usage of page-locked is mandatory for *MultiStream*. Moreover, if we consider that in the Map skeleton both the operands and the output are of the same length *N* and of the same type, we can approximate the previous equation to:

(2a) $$T_{comp}^{(CM1)} = (k+1)T_{comm}(N) + Ts.$$

3.1.4 MapReduce Cost Model – No MultiStream

The `skepu::MapReduce` skeleton provides the same behavior of the `skepu::Map` one with respect to the input operands. Its implementation requires that the last part of the reduce is performed by the CPU. Calling *Z* the length of the output we obtain:

(3a) $$T_{comp}^{(CM1)} = kT_{comm}(N) + Ts_{map} + Ts_{reduce} + T_{comm}(Z).$$

Considering a sufficiently large *N*, the Device-to-Host transfer cost can be omitted, thus:

$$T_{comp}^{(CM1)} = kT_{comm}(N) + Ts_{map} + Ts_{reduce}.$$

We won't use this last equation in the *MultiStream* analysis because, as will be shown, *N* will be divided by the number of streams used and *Z* will be multiplied instead.

3.1.5 MapArray Cost Model – No MultiStream

The `skepu::MapArray` skeleton accepts two input operands of different size. The first, of length *V*, is entirely accessible by all threads. The second, of length *N*, is partitioned across the threads becoming the object of the Map. The output has the same length of the second component. The cost model can be written as:

(4a) $$T_{comp}^{(CM1)} = T_{comm}(V) + 2T_{comm}(N) + Ts.$$

3.2 Page-Locked Memory

In this section, it will be shown how, in certain systems, the *communication times* T_{HtD} and T_{DtH} can be slightly optimized through the usage of page-locked memory. This optimization will not affect the cost models equations we've found, but only the communication time values since the bandwidth may increase.

All the operative systems that supports CUDA make use of *pageable* (i.e. virtualized) memory. Virtual memory segments are partitioned in pages which can be relocated, without changing their virtual address, from the main system memory (e.g. RAM) to a secondary storage device (e.g. hard disk) and viceversa. When a specific page is required by a process running on the host, the memory management unit (MMU) of the CPU is responsible of translating the virtual memory address to the physical one. If the page has been swapped to disk, the MMU signals a page fault to the virtual memory manager (VMM) of the operative system which will load the copy on the disk into the main memory and then resume the computation.

CUDA capable devices aren't equipped with an IOMMU (so far), thus they need to perform Direct Memory Access (DMA) using physical addresses. To be accessed directly, the interested pages have to reside in physical memory and marked by the VMM as locked (i.e. ineligible for eviction) [8: 3.8].

3.2.1 Asynchronous Operations

The access to data residing on the host memory by the GPU can follow two possible paths [Figure 4]:

- if the memory is pageable, data have to be copied first to a page-locked buffer by the CPU (synchronous copy)
- if the memory is already page-locked using a specific CUDA memory allocation

function, the GPU can directly fetch the required data through DMA. The CPU only needs to initialize the DMA controller with the amount of data to transfer and the memory address to use. After that, it is free to perform other operations (asynchronous copy).

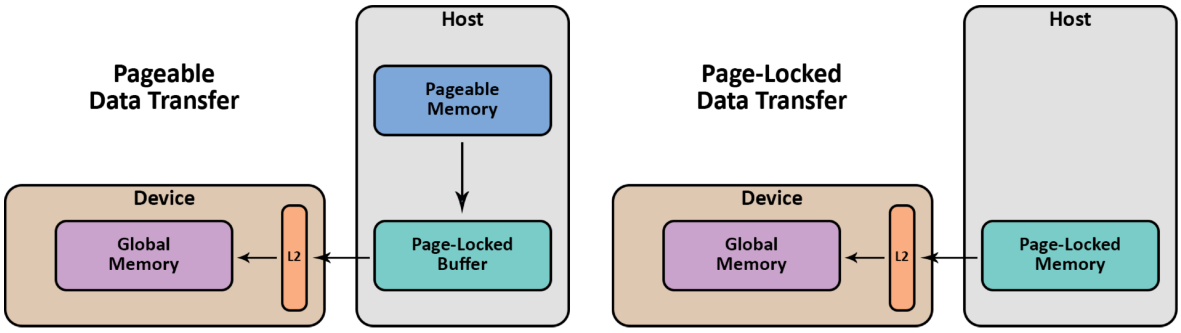

Figure 4: Pageable vs Page-Locked Data Transfer

To directly allocate page-locked memory, CUDA provides the `cudaHostAlloc()` function. The data transfer can then be issued by the `cudaMemcpyAsync()` function.

3.2.2 Performance Evaluation

The advantage of using page-locked memory isn't limited to the possibility of performing asynchronous transfers: by freeing the CPU from copying pageable memory to page-locked buffers we also increase the data transfer rates in all those systems where the bandwidth to perform a copy in the main memory is lower than the PCIe interface one.

To evaluate how the benefits of page-locked memory usage differ between different system, we have measured the bandwidth (GB/s) of a 64MB data transfer performed on two heterogeneous GPU-based machines:

A. 2 x AMD Opteron 6176 2.3 GHz CPUs with NVIDIA Tesla C2050 GPU
B. Intel Core i5-3210M 2.5 GHz CPU with NVIDIA GeForce GT630M GPU

As we can see in [Figure 5], the increase in bandwidth on the system with the more recent Intel Core i5 CPU is almost negligible with respect to the two times benefit measured in system (A).

The implementation of the SkePU smart containers already provides the application programmer with the possibility of using page-locked memory (*pinned* in CUDA's jargon) by defining the macro `USE_PINNED_MEMORY`.

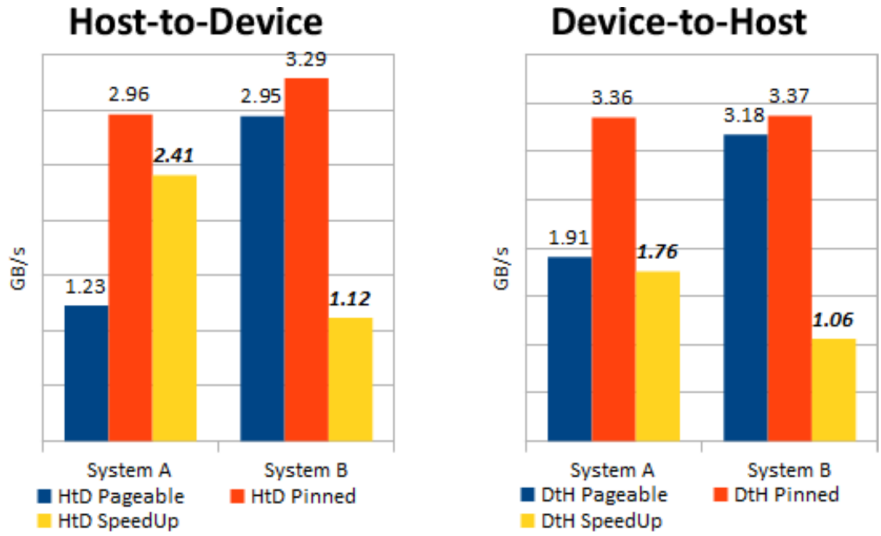

Figure 5: Page-Locked Speedup on different systems

3.2.3 Limits of Page-Locked Memory and MultiStream

The usage of page-locked memory is mandatory for the implementation of *MultiStream*. It should be noted that this requirement also limits the possibility to use *MultiStream* in very specific and relatively uncommon scenarios, namely in all those applications where the data-structures that need to be page-locked require nearly the same capacity as the physical memory present in the targeted system. In these cases the operating system would be left with insufficient physical memory to guarantee system stability and the efficient execution of the other running applications.

3.3 CUDA Streams

Before entering the details of *MultiStream* design, we need to define what a *stream* is in the CUDA parallel programming environment and how different generations of CUDA-enabled devices schedule the work submitted to them by the hosting system.

A *stream* in CUDA represents a queue of operations guaranteed to be executed on the device in the same order of issuing adopted in the host code.

3.3.1 Single Stream

When operations do not explicitly specify a stream, they are enqueued in the so-called *default stream*. The default stream is a synchronized stream: no operation inside its queue can start until all the other streams on the device have completed their execution.

Let us consider the following CUDA host code:

```
01 …
02 /* Allocate and populate a vector v of N floats */
03 /* Allocate a vector dev_v of N floats on the GPU */
04
05 cudaMemcpy(dev_v, v, N * sizeof(float), cudaMemcpyHostToDevice);
06 cuda_foo<<<1, N>>>(dev_v);
07 host_foo();
08 cudaMemcpy(v, dev_v, N * sizeof(float), cudaMemcpyDeviceToHost);
09 …
```

All the operations are issued sequentially to the GPU and are enqueued in the default stream. In particular, `cudaMemcpy` is a blocking operation (section 3.2.1) whereas `cuda_foo` is the kernel invocation which exhibits an asynchronous behavior: after its execution starts on the device, the CPU is free to perform `host_foo()`. This behavior easily enables overlapping between CUDA kernel and CPU executions.

CUDA provides the programmer with the possibility of using non-default streams. Let us change the previous code to make use of a programmer-defined stream:

```
01 …
02 // Initialize and create CUDA stream cs
03 cudaStream_t cs;
04 cudaStreamCreate(&cs);
05
06 //Initialize and allocate a page-locked data-structure for our data of size
N
07 float *host_v;
08 cudaHostAlloc((void**)&host_v, N * sizeof(float), cudaHostAllocDefault);
09
10 /* Populate host_v */
11
12 /* Allocate a vector dev_v of N floats on the GPU */
13
14 cudaMemcpyAsync(dev_v, host_v, N*sizeof(float), cudaMemcpyHostToDevice, cs);
15 cuda_foo<<<1, N, sharedMemSize, cs>>>(dev_v);
16 host_foo();
17 cudaMemcpyAsync(host_v, dev_v, N*sizeof(float), cudaMemcpyDeviceToHost, cs);
18 cudaStreamDestroy(cs);
19 …
```

In addition to the stream management functions and parameters, we can notice two main differences in this example:

- the memory on the host is allocated as page-locked with the `cudaHostAlloc` function;
- the `cudaMemcpyAsync` function, after enqueing the data transfer on the *cs* stream, immediately returns the control to the CPU.

Even though the code-complexity is significantly increased, the scheduling of the operations on the GPU is exactly the same as the one in the *default-stream* example.

3.3.2 Multiple Streams and Scheduling

To analyze the behavior of the schedulers present in different generations of CUDA-enabled devices, we need to consider a computation that makes use of multiple streams and take into account the following property:

Operations queued in different CUDA streams may be interleaved.

In particular:

- devices with compute capability 2.0 or greater may execute kernels concurrently if they operate on disjoint datasets and enough resources are available on the GPU;
- devices with compute capability 1.1 are equipped with one *kernel engine* and one *copy engine*. In this case both Host-to-Device and Device-to-Host transfers are performed sequentially by the only available copy engine;
- devices with compute capability 2.0 or greater have one kernel engine and two copy engines. In this case one copy engine will take care of the Host-to-Device transfers whereas the other will be used for the Device-to-Host transfers, enabling concurrency between the two operations.

In the following two examples, we will consider four different kernels, operating on four disjoint datasets and allocated in four different CUDA streams. The examples are functionally equivalent.

Example 1 – Depth-first

```
01 …
02 /* Initialize and create four CUDA streams cs[i] */
03 /* Allocate and populate four page-locked datasets v[i] of size N */
04
05 for(uint i=1; i<=4; i++){
06    cudaMemcpyAsync(dev_v[i], h_v[i], N * sizeof(float),
                      cudaMemcpyHostToDevice, cs[i]);
07
08    /* Execute Kernel i on stream i */
09
10    cudaMemcpyAsync(h_v[i], dev_v[i], N * sizeof(float),
                      cudaMemcpyDeviceToHost, cs[i]);
11 }
12 …
```

Example 2 – Breadth-first

```
01 …
02 /* Initialize and create four CUDA streams csi */
03 /* Allocate and populate four page-locked datasets vi of size N */
04
05 cudaMemcpyAsync(dev_v1, h_v1, N*sizeof(float), cudaMemcpyHostToDevice, cs1);
06 cudaMemcpyAsync(dev_v2, h_v2, N*sizeof(float), cudaMemcpyHostToDevice, cs2);
07 cudaMemcpyAsync(dev_v3, h_v3, N*sizeof(float), cudaMemcpyHostToDevice, cs3);
08 cudaMemcpyAsync(dev_v4, h_v4, N*sizeof(float), cudaMemcpyHostToDevice, cs4);
09 cuda_foo1<<<1, N, sharedMemSize, cs1>>>(dev_v1);
```

```
cuda_foo2<<<1, N, sharedMemSize, cs2>>>(dev_v2);
cuda_foo3<<<1, N, sharedMemSize, cs3>>>(dev_v3);
cuda_foo4<<<1, N, sharedMemSize, cs4>>>(dev_v4);
cudaMemcpyAsync(h_v1, dev_v1, N*sizeof(float), cudaMemcpyDeviceToHost, cs1);
cudaMemcpyAsync(h_v2, dev_v2, N*sizeof(float), cudaMemcpyDeviceToHost, cs2);
cudaMemcpyAsync(h_v3, dev_v3, N*sizeof(float), cudaMemcpyDeviceToHost, cs3);
cudaMemcpyAsync(h_v4, dev_v4, N*sizeof(float), cudaMemcpyDeviceToHost, cs4);
…
```

As we can see, the two examples only differ in the order with which the operations are issued to the GPU.

Let us make the following assumptions:

- $Ts_i = Ts \quad \forall i$, all kernels have the same service time
- $T_{comm} = Ts$, data transfers and kernels require the same time to be completed
- $Ts = \tau$

The execution for both examples without CUDA stream usage would be sequential, with $T_{comp}^{(seq)} = 12\tau$.

Let us now analyze the scheduling behavior on devices with different compute capabilities taking into account the properties listed at the beginning of this section.

[Figure 6] shows that for devices with compute capability 1.1 the scheduling behavior of example 1 is the same of the sequential one. This is due to the presence of only one copy engine: the *i*-th+1 Host-to-Device transfer is scheduled immediately after the *i*-th Device-to-Host transfer which, in turn, to start needs the *i*-th kernel execution to be completed. The breadth-first scheduling instead, enables concurrency of all the operations resulting in $T_{comp}^{(ex1)}(1.1) = 6\tau$.

The scheduling behavior of devices with compute capability 2.0 or greater is different: the depth-first example is the optimal one given the presence of two copy engines. The breadth-first approach, instead, could be partially inefficient because the scheduler tries to execute all the kernels in parallel when there are available resources (i.e. idle streaming multiprocessors). In this case the scheduler delays the completion signal of kernels concurrently running until all of them are actually completed. This, in turn, delays all the Device-to-Host transfers. The completion time in the worst-case scenario (i.e. all the different kernels run concurrently) becomes $T_{comp}^{(ex2)}(2.0) = 9\tau$.

As a further notice, it is important to specify that devices with compute capability 3.5 or greater introduce a scheduler that doesn't make any difference between the two code snippets: the result is in both cases the optimal one.

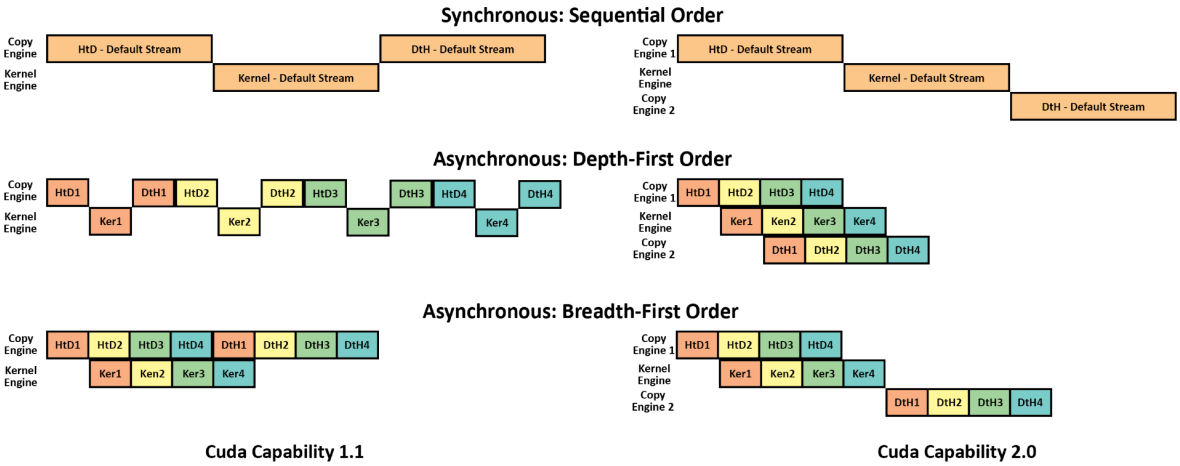

Figure 6: Scheduling Behavior and Overlap

3.4 MultiStream Design and Implementation

MultiStream is an optimization that exploits the intrinsic data-parallelism of each task and distributes it among two phases. In the first phase, before being sent to the device for the skeleton execution, the task is partitioned in as many subsets as the number of available CUDA Streams. In the second phase, SkePU executes the chosen skeleton on each of the task subsets. The distribution of the task subsets among different streams enables the overlap of communications times and service times.

3.4.1 Design

MultiStream is an implementation of the master/worker template [6: 8.4] operating on the host side where:

- the master is the host thread, which is responsible of partitioning the skeleton operands into *n* subsets and schedule a skeleton execution for each of the *n* different CUDA Streams;
- the workers are the set of *n* CUDA Streams. A sub-task is assigned to each worker which delegates the execution of the Map, MapReduce or MapArray skeleton on the device.

It is worth to notice that, even though the skeleton user function is the same for all the partitions, the kernel executions issued across different streams will still be considered separated and eligible for concurrency.

In the selection of the best scheduling strategy, we had to take into account the broad range of SkePU's supported architectures, therefore we have adopted the breadth-first schema seen in the example 2 of section 3.3.2. The breadth-first strategy should guarantee the optimal increase of performance over all the CUDA device generations,

including those with compute capability between 2.0 and 3.2. This assumption can be justified considering the fact that the SkePU run-time support will determine the best possible amount of threads per block and blocks per grid for each kernel call, therefore the usage of the device resources should be maximized and the concurrency of different kernel executions avoided.

3.4.2 Best-Case Scenario Analysis – With MultiStream

We are now ready to redefine the cost model found in section 3.1.1.

We will use $T_{comp}^{(CMn)}$ to denote the cost model completion time with *n* streams.

In that case we were analyzing a best-case scenario where only one operand was required to issue a new task and the result was collected by the host side only when all the tasks were exhausted. The cost model equation we found was:

(1a) $$T_{comp}^{(CM1)} = T_{HtD} + Ts$$

with MultiStream, the Host-to-Device transfer and the kernel execution are overlapped, thus we can rewrite it as:

(1b) $$T_{comp}^{(CMn)} = \frac{T_{HtD}}{n} + \frac{Ts}{n} + max\left[\left(T_{HtD} - \frac{T_{HtD}}{n}\right), \left(Ts - \frac{Ts}{n}\right)\right].$$

Let us, only for the sake of clarity, introduce the limit for $n \rightarrow \infty$ on (1b):

$$\lim_{n\rightarrow\infty} T_{comp}^{(GPGPU)} = \lim_{n\rightarrow\infty} \frac{T_{HtD}}{n} + \frac{Ts}{n} + max\left[\left(T_{HtD} - \frac{T_{HtD}}{n}\right), \left(Ts - \frac{Ts}{n}\right)\right] = max\left[T_{HtD}, Ts\right].$$

In other words, we move from a case in which the completion time of a task is the sum of both the time needed to transfer the task on the device and the time needed to serve it, to a case in which it just is the maximum of these two timings. In the best-case scenario, if the kernel is sufficiently complex to match the amount of time spent in communication we may achieve an ideal two times speedup.

In a real application this is obviously impossible because we don't have infinite workers at our disposal. The maximum amount of CUDA Streams varies according to the compute capabilities of the target GPU:

Compute Capability	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
# Streams	16		4	32				16

3.4.3 Implementation

In this section we discuss the *MultiStream* implementation in SkePU. We will adopt some pseudo-code and use different, explicit, variable names with respect to the ones used in the actual implementation code, to make sure the discussion remains understandable. The reason behind this choice resides in the fact that *MultiStream* is fully integrated with the other SkePU features to ensure backward compatibility, and the usage of the actual code would require an in-depth analysis of many features such as the specific implementation of the smart containers, the mechanism behind the selection of the amount of threads to be used for a given kernel call, the way smart containers are organized to keep track of which are the actually changed elements after a computation, etc. All those mechanism are not the object of this master thesis and will be left out of the discussion.

We will present the implementation of the `skepu::Map` skeleton with binary user function (i.e. two input operands, one output). The implementation of the other user functions can be easily derived from this one and, for what concerns the other skeletons, we will make the necessary distinctions when we will analyze their cost model with the *MultiStream* optimization taken into account.

The implementation of *MultiStream* can be divided in two main steps.

First we had to understand how SkePU handles its smart containers and find all the memory management functions used to transfer data in and out of the device memory. These functions don't operate on the smart containers directly, but on `device_pointer`s which are used to map a specific portion of memory on the host to its correspondent copy on the device. The *subset transfer* optimization, described in section 2.2.4, can use up to 10 `device_pointer`s to partition each smart container. To implement MultiStream, we first had to properly modify all the memory management functions such that we could define which stream to use for a specific memory update.

In the second step, we implemented the master/worker template in each skeleton. We used one `device_pointer` for each partition of the operands and output. By doing so we increased the maximum amount of tracked `device_pointer`s for each smart container from 10 to 32 (maximum supported CUDA Streams) while maintaining the possibility for *subset transfer* to further divide each *MultiStream* partition if needed. To the increase in the amount of tracked partitions corresponds an increase in the overhead due to the smart containers usage. As will be shown in the implementation code, we adopted the breadth-first model to schedule the page-locked allocation and asynchronous transfer of each `device_pointer` to the proper CUDA Stream. The kernel is then invoked *n* times.

As an additional note which didn't require much effort, we added a function to initialize a specific amount of CUDA Streams to the class that manages the GPU initialization.

skepu::Map skeleton implementation with Binary user function

Let us start by recalling the code required to initialize and perform a `skepu::Map` skeleton:

```
01 BINARY_FUNC(foo_f, float, a, b, return a op b);
02 int main(){
03    // Initialize three vectors
04    skepu::Vector<float> v1(1000);
05    skepu::Vector<float> v2(1000);
06    skepu::Vector<float> vres(1000);
07    /* Populate vectors v1 and v2 */
08    // Instantiate the Map skeleton and perform the foo_f user function
09    skepu::Map<foo_f> foo(new foo_f);
10    foo(v1, v2, vres);
11    /* Print the result */
12    return 0;
13 }
```

In the 1.2 Beta version of SkePU, to enable MultiStream, the application programmer had to define the `USE_PINNED_MEMORY` and `USE_MULTISTREAM` macros. In the v1.2 official release the former is enough.

In section 2.2.1 we have seen how the `BINARY_FUNC` macro expands, and section 3.1.3 gave a general idea on the CUDA representation of the `skepu::Vector`.

Let us now analyze what happens when the `skepu::Map` is defined:

```
01 template <typename MapFunc>
02 Map<MapFunc>::Map(MapFunc* mapFunc){
03 …
04 #ifdef SKEPU_CUDA
05    cudaDeviceID = Environment<int>::getInstance()->bestCUDADevID;
06    backEnd.maxThreads = m_environment->m_devices_CU.at(0)->getMaxThreads();
07    backEnd.maxBlocks = m_environment->m_devices_CU.at(0)->getMaxBlocks();
08 #endif
09 …
10 }
```

The SkePU run-time support uses the `Environment` class to select and initialize the best (i.e. with higher CUDA compute capabilities) CUDA device available in the system. Then stores some useful information in the `backEnd` structure, that will be used to determine how many threads per block and blocks per grid will be used for a specific kernel execution.

As far as *MultiStream* is concerned, during the device initialization we check how many CUDA streams are supported, then we initialize and start them:

```
01 …
02 cudaStream_t m_streams[MAX_POSSIBLE_CUDA_STREAMS_PER_GPU]; // 32
03 …
04 Device_CU(unsigned int id){
05    m_deviceID = id;
06    cudaSetDevice(m_deviceID);
```

```
07    initDeviceProps(id);
08 #ifdef USE_PINNED_MEMORY
09    for(unsigned int i=0; i<numConcurrKernelsSupported; i++)
10       cudaStreamCreate(&(m_streams[i]));
11 #endif
12 …
13 }
14
15 void initDeviceProps(unsigned int device){
16    cudaGetDeviceProperties(&m_deviceProp, device);
17 …
18    numConcurrKernelsSupported = getMaxConcurKernelsSupported(m_deviceProp);
19 …
20 }
```

The `getMaxConcurKernelsSupported` function checks the compute capabilities of the device and returns the value according to the table presented in section 3.4.2.

We can now analyze what happens when the skeleton is executed `foo(v1, v2, vres)`:

```
01 template <typename MapFunc>
02 template <typename T>
03 void Map<MapFunc>::CU(Vector<T>& in1, Vector<T>& in2, Vector<T>& output){
04    CU(in1.begin(), in1.end(), in2.begin(), in2.end(), output.begin());
05 }
```

As we can see, another function `CU` is called. The same call is made when the Matrix smart container is used. The `CU` function is not reported: it only acts as a selector of behavior. It checks for the `USE_PINNED_MEMORY` macro definition and calls the `mapMultiStream_CU` function. The parameters the `CU` function takes are references to the first and last element of the input operands and the start of the output smart container. Through these parameters, SkePU defines iterators that are used to access the two smart containers. The copies on the device memory can be accessed and modified using the `device_pointer` class. This class provides different memory management functions. We will describe the functions that have been modified or extended to implement *MultiStream*.

Let us start from the `updateDeviceCU` function in the Vector (Matrix) smart container class:

```
01 template <typename T>
02 typename Vector<T>::device_pointer Vector<T>::updateDevice_CU(T* start,
                                 size_type numElements, unsigned int deviceID,
                                 bool copy, unsigned int streamID){
03 …
04    /* Check for the presence of the specified range
05    * (i.e. [start, start+numElements] ) on the device
06    */
07
08    /* If the range is not found, create a device_pointer<T> dev_pointer
09     * with parameters (start, numElements) and allocate it
10    */
11
```

```
12    /* Else make sure that the data it contains are marked as invalid */
13
14    // Performed in both cases
15    if(copy){
16       dev_pointer.copyData(dev_pointer, deviceID, streamID);
17    }
18 …
19 }
```

The `device_pointer` contains the addresses of both the copy on the host and the copy on the device memory.

The call to `copyData` is the one that issues the actual data transfer:

```
01 template <typename T>
02 void device_pointer<T>::copyData(device_pointer<T> data, unsigned int
                                    deviceID, size_t streamID){
03 …
04 #ifdef USE_PINNED_MEMORY
05    cudaMemcpyAsync(data.dev_start, data.host_start, data.size,
                      cudaMemcpyHostToDevice,
                      (m_devices_CU.at(deviceID)→m_streams[streamID]) );
06 …
07 }
```

We can finally analyze the `mapMultiStream_CU` function, which is the one implementing the *master* component of *MultiStream*. All the required details are provided as comments:

```
01 /*!
02 * Applies the Map skeleton to two ranges of elements specified by iterators.
03 * Result is saved to a separate output range.
04 * The calculations are performed by one host thread using one CUDA device
05 * as backend.
06 * HtD transfers and Kernel execution are partitioned across multiple CUDA
07 * Streams to achieve overlap.
08 *
09 * The skeleton must have been created with a binary user function.
10 *
11 * \param input1Begin An iterator to the first element in the first range.
12 * \param input1End An iterator to the last element of the first range.
13 * \param input2Begin An iterator to the first element in the second range.
14 * \param input2End An iterator to the last element of the second range.
15 * \param outputBegin An iterator to the first element of the output range.
16 * \param deviceID Integer specifying the which device to use.
17 */
18 template <typename MapFunc>
19 template <typename Input1Iterator, typename Input2Iterator, typename
                 OutputIterator>
20 void Map<MapFunc>::mapMultiStream_CU(Input1Iterator input1Begin,
              Input1Iterator input1End, Input2Iterator input2Begin,
              Input2Iterator input2End, OutputIterator outputBegin,
              unsigned int deviceID){
21
22    // Set the device on which the skeleton will be performed
23    CHECK_CUDA_ERROR(cudaSetDevice(deviceID));
24
```

```
25    // Get the amount of CUDA Streams supported by the device
26    size_t numKernels = m_devices_CU.at(deviceID)→getNoConcurrentKernels();
27
28    // Determine the size of the operands partitions
29    size_t n = input1End - input1Begin;
30    if(n < numKernels)
31       numKernels = n;
32    size_t numElemPerSlice = n / numKernels;
33    size_t rest = n % numKernels;
34
35    // Declare three arrays containing the device_pointers to each operand
36    // partition
37    typename Input1Iterator::device_pointer in1_mem_p[numKernels];
38    typename Input2Iterator::device_pointer in2_mem_p[numKernels];
39    typename OutputIterator::device_pointer out_mem_p[numKernels];
40
41    // First allocate CUDA memory without making any copy
42    size_t i, numElem;
43    for(i = 0; i < numKernels; ++i){
44       if(i == numKernels-1)
45          numElem = numElemPerSlice+rest;
46       else
47          numElem = numElemPerSlice;
48
49       in1_mem_p[i] =
             updateDevice_CU((input1Begin+i*numElemPerSlice).getAddress(),
                                 numElem, deviceID, false, i);
50       in2_mem_p[i] =
             updateDevice_CU((input2Begin+i*numElemPerSlice).getAddress(),
                                 numElem, deviceID, false, i);
51       out_mem_p[i] =
             updateDevice_CU((outputBegin+i*numElemPerSlice).getAddress(),
                                 numElem, deviceID, false, i);
52    }
53
54    // Breadth-first memory transfers and kernel executions
55    //First operand memory transfer
56    for(i = 0; i < numKernels; ++i){
57       if(i == numKernels-1)
58          numElem = numElemPerSlice+rest;
59       else
60          numElem = numElemPerSlice;
61
62       in1_mem_p[i] =
             updateDevice_CU((input1Begin+i*numElemPerSlice).getAddress(),
                                 numElem, deviceID, true, i);
63    }
64    //Second operand memory transfer
65    for(i = 0; i < numKernels; ++i){
66       if(i == numKernels-1)
67          numElem = numElemPerSlice+rest;
68       else
69          numElem = numElemPerSlice;
70
71       in2_mem_p[i] =
             updateDevice_CU((input2Begin+i*numElemPerSlice).getAddress(),
                                 numElem, deviceID, true, i);
```

```
72     }
73
74     //Kernel executions
75     for(i = 0; i < numKernels; ++i){
76        if(i == numKernels-1)
77           numElem = numElemPerSlice+rest;
78        else
79           numElem = numElemPerSlice;
80
81        /* Find the best number of threads and blocks
82
83        MapKernelBinary_CU<<<numBlocks, numThreads, 0,
                     (m_devices_CU.at(deviceID)→m_streams[i])>>>
                            (*m_mapFunc, in1_mem_p[i]→getDeviceDataPointer(),
                            in2_mem_p[i]→getDeviceDataPointer(),
                            out_mem_p[i]→getDeviceDataPointer(), numElem);
84
85        // Notify to the device_pointers of the output partitions that
86        // their data has been updated such that the copy on the host
87        // memory is marked as invalid
88        out_mem_p[i]->changeDeviceData();
89     }
90  }
```

The presented code doesn't cover the Device-to-Host transfers. Their implementation is proposed as future work because requires an almost complete rewriting of the smart containers functions that trigger the update of the host copy. As of SkePU version 1.2, the data transfer is entirely issued to `m_streams[0]`, thus happens sequentially. The scheduling and overlapping behavior is presented in [Figure 7].

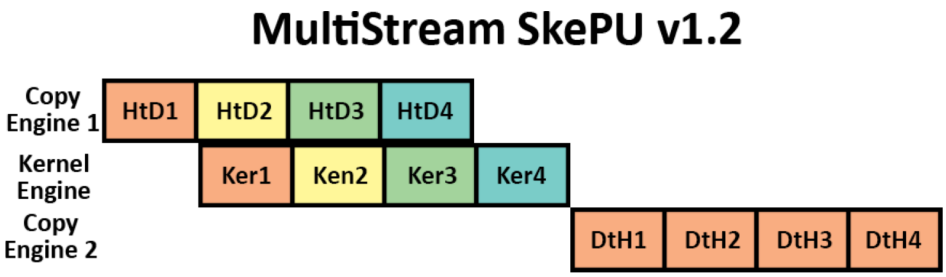

Figure 7: MultiStream Scheduling Behavior and Overlap in SkePU v1.2

In the cost model analysis we will take this limitation into account and we will provide two different equations: one, ideal, which includes a full MultiStream implementation, and one, effective, which considers the lack of overlapping of the Device-to-Host transfers operations.

3.4.4 Map Cost Model – With MultiStream

Let us recall the cost model for the `skepu::Map` skeleton we've found in section 3.1.3:

(2a) $T_{comp}^{(CM1)} = (k+1)T_{comm}(N) + Ts$ with $1 \leq k \leq 3$ input operands and *N* the number of elements of each operand.

Fixed *n* the number of available CUDA Streams on the device we get:

(2b) $$T_{comp}^{(id)} = min[(k+1)T_{comm}(\frac{N}{n}), \frac{Ts}{n}] + max[kT_{comm}(N), Ts]$$

(2c) $$T_{comp}^{(CMn)} = min[kT_{comm}(\frac{N}{n}), \frac{Ts}{n}] + max[kT_{comm}(N), Ts] + T_{comm}(N)$$

3.4.5 MapReduce Cost Model – With MultiStream

Before getting into the cost model analysis, let us describe the main differences in the implementation of `skepu::MapReduce` with respect to `skepu::Map`.

The map phase is exactly the same with the sole exception that a temporary `device_pointer` is used to store its result.

The reduce kernel is then executed for each stream: it applies the reduce function to produce one element per block, then issues a Device-to-Host transfer to complete the reduce on the host side. In this transfer, *MultiStream* loses some performance: the number of blocks and threads used to initialize the kernel execution is constant in the SkePU Reduce implementation. This means that, calling *Z* the number of blocks per thread, we transfer nZ elements with *MultiStream* instead of Z only.

The cost model equation without MultiStream was:

(3a) $T_{comp}^{(CM1)} = kT_{comm}(N) + Ts_{map} + Ts_{reduce} + T_{comm}(Z)$.

With *MultiStream* we get $T_{comp}^{(id)} = T_{comp}^{(eff)}$ since the final part of the reduce is computed on the host side, thus:

(3b) $$T_{comp}^{(CMn)} = min[kT_{comm}(\frac{N}{n}), \frac{Ts_{map}}{n}] + max[kT_{comm}(N), Ts_{map}] + Ts_{reduce} + nT_{comm}(Z)$$

3.4.6 MapArray Cost Model – With MultiStream

The implementation of the skepu::MapArray is similar to the one presented in section 3.4.3, but we need to consider that the first operand can't be partitioned because it is entirely needed for the computation of each sub-task.

The cost model equation found without MultiStream was:

(4a) $T_{comp}^{(CM1)} = T_{comm}(V) + 2T_{comm}(N) + Ts$, with *V* size of the first input.

With the MultiStream optimization it becomes:

(4b) $T_{comp}^{(id)} = T_{comm}(V) + min[2T_{comm}(\frac{N}{n}), \frac{Ts}{n}] + max[T_{comm}(N), Ts]$

(4c) $T_{comp}^{(CM)} = T_{comm}(V) + min[T_{comm}(\frac{N}{n}), \frac{Ts}{n}] + max[T_{comm}(N), Ts] + T_{comm}(N)$

3.5 Final Remarks

We have shown that *MultiStream* can potentially provide a significant speedup in a task computation by ideally reducing the sum of the time needed to transfer the task on the GPU and the time needed to execute the kernel, to the max between the two of them. *MultiStream* doesn't currently support the overlap of the Device-to-Host transfers which would further improve the performance, but with some changes to the smart containers is definitely possible to implement the complete optimization.

In the next chapter, we will test MultiStream and the calculated cost models against various benchmarks to prove their correctness.

4. MultiStream Testing

In this chapter, we discuss some experimental results aimed at assuring the functionality and performance of the MultiStream implementation. We will start by benchmarking the SkePU skeletons with and without *MultiStream* to check the correctness of the computed cost models. We will then introduce *NevMap*, a standalone CUDA implementation of the Map skeleton, built to show the full potential of *MultiStream* by enabling also the overlap of the Device-to-Host transfers. Finally, we will make a brief introduction to FastFlow, a stream-parallel programming framework, and we will use it to issue a stream of tasks to SkePU to show how *MultiStream* would perform in a real use-case scenario.

4.1 Testing Platforms and Measuring Methods

For the sake of completeness, let us recall the specifics of the two heterogeneous GPU-based systems we have used for testing.

	System A	*System B*
Name	**Titanic**	**c7**
CPU	Dual AMD Opteron 6176 2.3 GHz 12 Cores	Intel Core i5-3210M 2.5 Ghz 2 Cores + HyperThreading
RAM	32 GB DDR2	6 GB DDR3
GPU	NVIDIA Tesla C2050 56 SM – 3GB GDDR5	NVIDIA GeForce GT630M 12 SM – 1GB GDDR5
Comp.Capability	2.0 (16 CUDA Streams)	2.0 (16 CUDA Streams)

The vast majority of the presented tests returns the same performance figures on both systems, therefore we will show the results from *Titanic*. The second system, *c7*, has been taken into account in section 3.2.2 when we measured the performance of page-locked memory with respect to the pageable one, and will be considered again in those tests in which a lower number of streaming multiprocessors is required to gather further details.

All tests are performed on a modified SkePU version that gives us the possibility to enable the page-locked optimization without necessarily using *MultiStream*. This guarantees that the $T_{HtD} \approx T_{DtH} \approx T_{comm}$ relation introduced in section 3.1.3 is valid, and that any measured speedup is due to the *MultiStream* optimization only.
The measured performances make use of two different probes:

- We use the `sys/time` class (`gettimeofday`) in the skeleton implementations to measure the overhead of setting up a computation with *MultiStream*.
- CUDA Events are, instead, adopted to benchmark the operations that run asynchronously on the device, as suggested by NVIDIA.

We will also adopt the following conventions:

$T_X^{(1S)}$ experienced time of performance index *X* without *MultiStream*

$T_X^{(MS)}$ experienced time of performance index *X* with *MultiStream*

4.2 MultiStream Overhead

To evaluate the overhead of MultiStream, we have inserted probes in SkePU and taken measures with different values of adopted CUDA Streams, operand size, and number of operands. Given the complexity of the SkePU library, we weren't able to identify every source of overhead, but a large portion seems to be linearly dependent on the number and size of partitions involved in a skeleton computation. We will use $T_{OH}^{(MS)}$ to denote the overhead we were able to actually measure and identify.
On Titanic $T_{OH}^{(MS)} \approx 0.125\,ms$ for each smart container of 16 million float elements and each stream adopted in the computation of one task.
Another portion of the overhead seems to be directly proportional to the complexity of the function used for the skeleton computation. Since the user function entirely depends on the application logic, we couldn't define a generalized function to model it.

4.3 SkePU Skeletons Benchmarks

In this section, we test the implementation of *MultiStream* for each of the supported skeletons. The benchmarking applications apply complex (synthetic) user functions such that $Ts^{(1S)} \approx T_{comm}^{(1S)}$ (slightly lower, actually).
The performance measurements used to test the correctness of the cost models are taken using input operands of length $N = 16\,M$ with elements of float type.

4.3.1 Map Performance Measurement

Let us recall the cost model of the skepu::Map with *MultiStream* found in section 3.4.4 and include the overhead evaluated in section 4.2:

(2c) $$T_{comp}^{(CMn)} = min[k\,T_{comm}^{(1S)}(\frac{N}{n}), \frac{Ts^{(1S)}}{n}] + max[k\,T_{comm}^{(1S)}(N), Ts^{(1S)}] + T_{comm}^{(1S)}(N) + T_{OH}^{(MS)}$$

The measured performance indexes without *MultiStream* are:

$T_{comp}^{(1S)} = 55.4\,ms$, $Ts^{(1S)} = 17.2\,ms$, $T_{comm}^{(1S)}(N) = 19.1\,ms$

With *MultiStream* instead: $T_{OH}^{(MS)} = 4\,ms$, $T_{comp}^{(MS)} = 45.5\,ms$, resulting in a 21.7% speedup.

Let us now substitute the retrieved data in (*2c*) knowing that $k=1$:

$$T_{comp}^{(CMn)} = min[\frac{19.1}{16} + \frac{17.2}{16}] + max[19.1, 17.2] + 19.1 + 4 = 43.3\,ms$$

The cost model seems to reliably approximate (95.2%) the effective performance.

4.3.2 MapReduce Performance Measurement

The measured performance indexes of the `skepu::MapReduce` skeleton execution without *MultiStream* are:

$T_{comp}^{(1S)} = 55.6\,ms$, $Ts_{map}^{(1S)} = 13\,ms$, $Ts_{reduce}^{(1S)} = 4.4\,ms$, $T_{comm}^{(1S)}(N) = 19.1\,ms$

With *MultiStream* instead: $T_{comm}^{(1S)}(nZ) = 1.2\,ms$, $T_{OH}^{(MS)} = 6\,ms$, $T_{comp}^{(MS)} = 52.3\,ms$

The speedup in this case is the 6.3%.

Substituting in the cost model equation with $k=2$ we have:

(3b) $$T_{comp}^{(CMn)} = min[k\,T_{comm}^{(1S)}(\frac{N}{n}), \frac{Ts_{map}^{(1S)}}{n}] + max[k\,T_{comm}^{(1S)}(N), Ts_{map}^{(1S)}] + Ts_{reduce}^{(1S)} + nT_{comm}^{(1S)}(Z) + T_{OH}^{(MS)}$$

$$T_{comp}^{(CMn)} = min[2\frac{19.1}{16}, \frac{13}{16}] + max[2*19.1, 13] + 4.4 + 1.2 + 6 = 50.6\,ms$$

The worse result with respect to the Map skeleton is caused by two main factors:

- The overhead is the one expected from the usage of (*k*+1) smart containers. This is actually the case since we have to consider the two input operands (*k*) and the temporary data-structure that contains the output of the map phase.
- In section 3.4.5, we have seen that the parameters used by SkePU to initialize the Reduce kernel are constant. This implies that for each one of the *n* Reduce executions, we setup a new communication to perform the Device-to-Host transfer of the *Z* elements that will be used to complete the reduction phase on the host side, resulting in an additional source of overhead.

Also in this case, the cost model seems to correctly approximate (96.7%) the effectively measured performance.

4.3.3 MapArray Performance Measurement

In this test we have used as first operand (i.e. the array fully accessible by each device thread) a vector of length $V=192$. This value ensures that the kernel service time is close to the communication time of the second operand. Even though *V* is small, the time required to transfer the vector is particularly high because we are actually paying the setup of the transfer instead of the transfer itself.

The measured performance indexes without *MultiStream* are:

$T_{comp}^{(1S)}=56.3\,ms$, $Ts^{(1S)}=18.1\,ms$, $T_{comm}^{(1S)}(V)=0.1\,ms$, $T_{comm}^{(1S)}(N)=19.1\,ms$

With *MultiStream* instead: $T_{comp}^{(MS)}=46.3\,ms$, $T_{OH}^{(MS)}=4\,ms$

The achieved speedup is similar to the one experienced with the Map skeleton: 21.6%.

Substituting in the cost model equation found in section 3.4.6, we get:

(4c)

$$T_{comp}^{(CMn)}=T_{comm}^{(1S)}(V)+min[T_{comm}^{(1S)}(\frac{N}{n}),\frac{Ts^{(1S)}}{n}]+max[T_{comm}^{(1S)}(N),Ts^{(1S)}]+T_{comm}^{(1S)}(N)+T_{OH}^{(MS)}$$

$$T_{comp}^{(CMn)}=0.1+min[\frac{19.1}{16},\frac{18.1}{16}]+max[19.1,18.1]+19.1+4=43.4\,ms$$

In this case the cost model approximation seems to be slightly different with respect to the experienced completion time (93.7%). We didn't have time to identify and include any additional source of overhead, but we still think that the cost model is sufficiently reliable for our purposes.

4.3.4 Synthetic Benchmarks Conclusions

The performed synthetic benchmarks have shown that the SkePU implementation of *MultiStream* behaves as predicted by our cost models and is able to provide a significant speedup (>20%) in both the Map and MapArray cases.

The result obtained for the MapReduce case is less important but remains positive. We didn't have time to code a better implementation but we believe that further optimizations are possible. In particular, it should be investigated the possibility of performing a single Reduce, using only one stream, after all the Map kernels executions have been completed. This way only one Device-to-Host communication with a small payload would be setup and performed.

4.4 Standalone MultiStream Performance

In this section we compare SkePU performance against *NevMap*, a standalone implementation of the Map skeleton which provides the complete MultiStream functionality (i.e. also Device-to-Host transfers may be overlapped). The purpose of this

test is to evaluate the potential performance improvement that SkePU may achieve provided a full *MultiStream* implementation.

4.4.1 NevMap Benchmark Overview

The programmable interface of *NevMap* acts as an easy-to-use wrapper for CUDA. It is inspired by SkePU, although it trades off many of the facilities that make SkePU programming model so high-level and portable, for a higher control over the behavior of the CUDA architecture. This obviously isn't in line with the principles of the skeleton based parallel programming approach, but it will serve our testing purposes.

A detailed description of *NevMap*, along with the full source code can be found in Appendix A.

Like in section 4.3, the benchmarking application applies a complex user functions such that $Ts^{(1S)} \approx T_{comm}^{(1S)}$. In particular, it is a two-input one-output function used to instantiate a Map skeleton that takes two operands of length $N = 16M$. The computation is expressed by the following code:

```
01 FUNC_2i1o(foo, float, a, b, return /* complex function f(a,b) */;)
02 int main(void) {
03    float *in1, *in2, *out;
04    size_t numElements = 16000000;
05    cudaHostAlloc((void**) &in1, numElements * sizeof(float),
                  cudaHostAllocDefault);
06    cudaHostAlloc((void**) &in2, numElements * sizeof(float),
                  cudaHostAllocDefault);
07    cudaHostAlloc((void**) &out, numElements * sizeof(float),
                  cudaHostAllocDefault);
08
09    /* Populate the two input vectors */
10
11    // Initialize the Device, CUDA Streams and Map skeleton
12    NevMap<float, foo> nevmap(in1, in2, out, new foo, numElements);
13
14    // Transfer operands and execute the map using a Depth-first schedule
15    // The output is asynchronously sent to the host as soon as available
16    nevmap.transferExecuteDF();
17    /*
18    * Any operation on different data can be performed by the CPU here
19    */
20    // Explicitly make sure that the output is in the host memory
21    nevmap.synch();
22    return 0;
23 }
```

It can be observed that this code resembles the one of a typical SkePU application, although `cudaHostAlloc` has to be explicitly used to allocate page-locked memory since no smart containers are available, and we need to call the `synch` function to make sure that the device has finished computing the Map skeleton and the result has been fully transferred to the host memory.

4.4.2 Complete MultiStream Performance

Let us list the measured performance indexes. We will also compare them with our cost model to enforce its correctness with a different function and one more operand.

Without *MultiStream* we get:

$T_{comp}^{(1S)}=77.5\,ms$, $Ts^{(1S)}=20.2\,ms$, $T_{comm}^{(1S)}(N)=19.1\,ms$

With the partial *MultiStream* SkePU implementation: $T_{comp}^{(MS)}=67.7\,ms$, $T_{OH}^{(MS)}=6\,ms$

Taking into account its stand-alone implementation, *NevMap* doesn't suffer any of the overheads caused by the management of smart containers or by the other SkePU features. Therefore, to be as precise as possible in our evaluation, we will need to use a derived completion time $T_{comp}^{(NevD)}=T_{comp}^{(Nev)}+T_{OH}^{(MS)}+D$ where $T_{comp}^{(Nev)}$ is the actual *NevMap* completion time and $D=T_{comp}^{(MS)}-T_{comp}^{(CMn)}$.

Solving the (*2c*) cost model we get $T_{comp}^{(CMn)}=64.6$ (95.4%), thus $D=3.1\,ms$.

NevMap effective completion time is $T_{comp}^{(Nev)}=47.4$ from which we derive:

$T_{comp}^{(NevD)}=56.5\,ms$

[Figure 8] shows the benefit a full implementation of MultiStream would give to SkePU. The Device-to-Host transfers cut out a large portion of the potentially achievable speedup (up to 37%). We also plotted the actual *NevMap* speedup which reaches the 63%. This value testifies how an increase in programmability and portability affects the potential performance achievable with a lower-level solution, but it also has to be seen as a target for further optimizations of the SkePU backend.

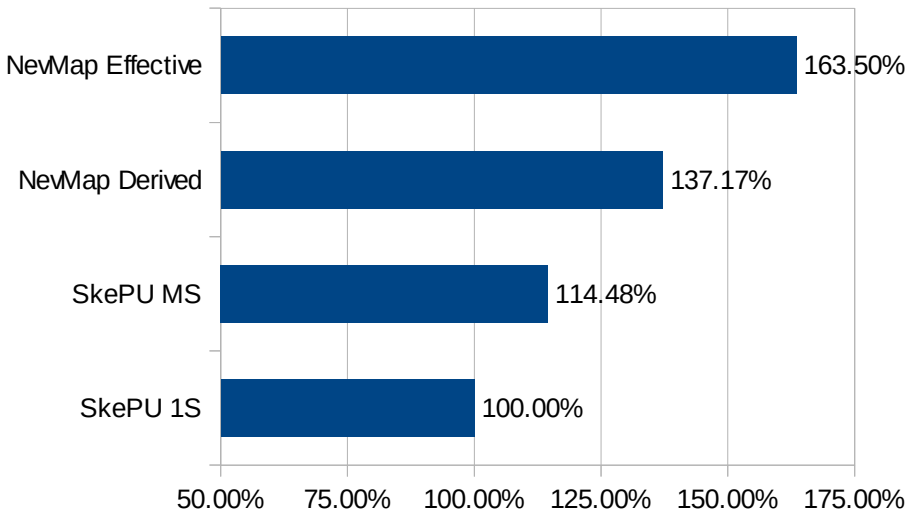

Figure 8: Potential MultiStream Speedup

4.5 Real Stream-Parallel Use-case Benchmark

In this section we will briefly introduce FastFlow, a stream-parallel programming framework, which we use to issue a stream of tasks to SkePU. We will then show how SkePU performs with and without MultiStream using an application that computes the Luminance Greyscale algorithm of a stream of images.

4.5.1 FastFlow

FastFlow is a C++ stream-parallel programming framework developed and maintained by professor Massimo Torquati (Pisa) and professor Marco Aldinucci (Torino) at the Department of Computer Science of the University of Pisa and Torino, Italy.
FastFlow is designed to promote skeleton based parallel programming, providing the application programmer with a stack of layers that progressively abstracts out the development of parallel applications [7]. Its run-time support is implemented through nonblocking lock and fence-free algorithms to guarantee an efficient exploitation of high frequency streaming and fine-grain parallelism. Sequential code can be reused to program the logic of high-level patterns (e.g. parallel-for, stencil-reduce, farm-with-feedback) and core patterns (farm, pipeline, loopback) designed to easily exploit parallelism out of sequential applications. Moreover, application programmers are given the possibility to design their own patterns through the definition of a graph made of `ff_nodes` (i.e. processes) and `channels` (i.e. units devoted to the synchronization and communication between `ff_nodes`).

4.5.2 Greyscale

In our case we use FastFlow to implement a four-stage pipeline that produces a stream of sRGB images on the host side and sends them to the device. The GPGPU portion of the application is implemented using SkePU which applies a pixel-by-pixel Map skeleton to convert the image from color to grey. The result is then transferred to the host for further computation.
The workflow of the application can be represented by [Figure 9]:

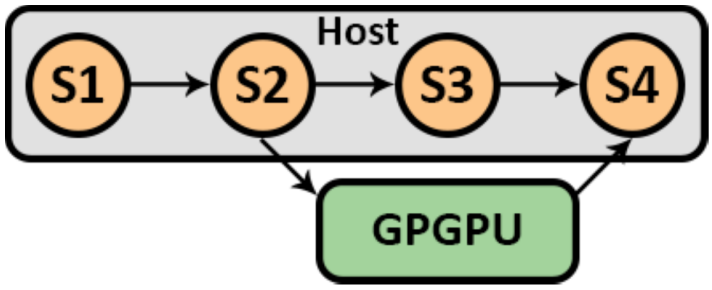

Figure 9: Greyscale Workflow

- *S1* is the first stage of the FastFlow pipeline, responsible of loading/generating the sRGB image, create a task out of it (it can be the image plus additional data, for example) and send a pointer of the task out to the next stage.
- S2 receives the task and demands the computation of Greyscale to SkePU. Since the copy on the host side is not accessed, the task can be sent to the next stage as soon as the skepu::Map function has returned (asynchronously).
- S3 is a dummy stage. It has been included to explicitly show that the CPU can concurrently process any other function on the task data as long as they are independent of the image sent to the device. Once the S3 computation is done, it sends the task on the next and final stage.
- GPGPU represents the skepu::Map computation on the device, which can happen in parallel with S3 on the host side.
- S4 is the final stage which collects the result of the GPGPU computation, any other result that S3 may have computed, and concludes the computation (e.g. by showing or storing the greyscale image).

The Greyscale algorithm applies three functions pixel-by-pixel, to each of the RGB channels of the image. We are assuming the images are in the sRGB colorspace [11], which means they are gamma compressed [12].

The first function applies a gamma expansion. The second one computes the new pixel value according to the BT.709 ITU-R recommendation which weights each color channel according to the human eye perception. The third function, finally, applies a gamma compression. This is the algorithm used, with little variations, by many image processors such as GIMP or Adobe Photoshop to automatically convert color images into greyscale ones. The complete source code of the application can be found in Appendix B.

4.5.3 Measured Performance Vs Cost Model

In this section we compare the performance measurements taken on both *Titanic* and *C7* systems against out cost model. With the former system, the Greyscale algorithm takes less time to complete with respect to the time needed to transfer the image to and from the device. The latter system, instead, shows the opposite behavior giving us the possibility to test our cost model changing the predominant factor.

The measurements are taken for an image of 1920x1080px (i.e. vector length = 2M)

	$T_{comp}^{(1S)}$	$Ts^{(1S)}$	$T_{comm}^{(1S)}$	$T_{OH}^{(MS)}$	$T_{comp}^{(MS)}$	$T_{comp}^{(CM)}$	*Approx%*
Titanic	18.86ms	3.90ms	7.48ms	0.5ms	16.93ms	15.7ms	92.7%
C7	41.91ms	27.03ms	7.45ms	3.0ms	40.23ms	37.95ms	94.3%

In both cases, the *MultiStream* cost model seems to reliably approximate the measured completion time, even though a better knowledge of the overhead's causes could make it more precise.

4.5.4 Speedup Vs Operand Size

The last test we present, compares the speedup obtained by running the Greyscale application with different image sizes. FastFlow is used to generate a stream of 60 images. The following table represents the collected measurements for common image resolutions, while [Figure 10] plots them (logarithmic X-axis):

Resolution	**640x480**	**800x600**	**1024x768**	**1280x720**	**1600x900**	**1920x1080**	**2560x1440**	**3840x2160**
Input Length	307200	480000	786432	921600	1440000	2073600	3686400	8294400
SkePU 1S	180	276	441.6	514.8	790.8	1131.6	1992	4470
SkePU MS	177.6	264	412.8	478.8	724.8	1024.8	1794	3996
Speedup	*101.35%*	*104.55%*	*106.98%*	*107.52%*	*109.11%*	*110.42%*	*111.04%*	*111.86%*

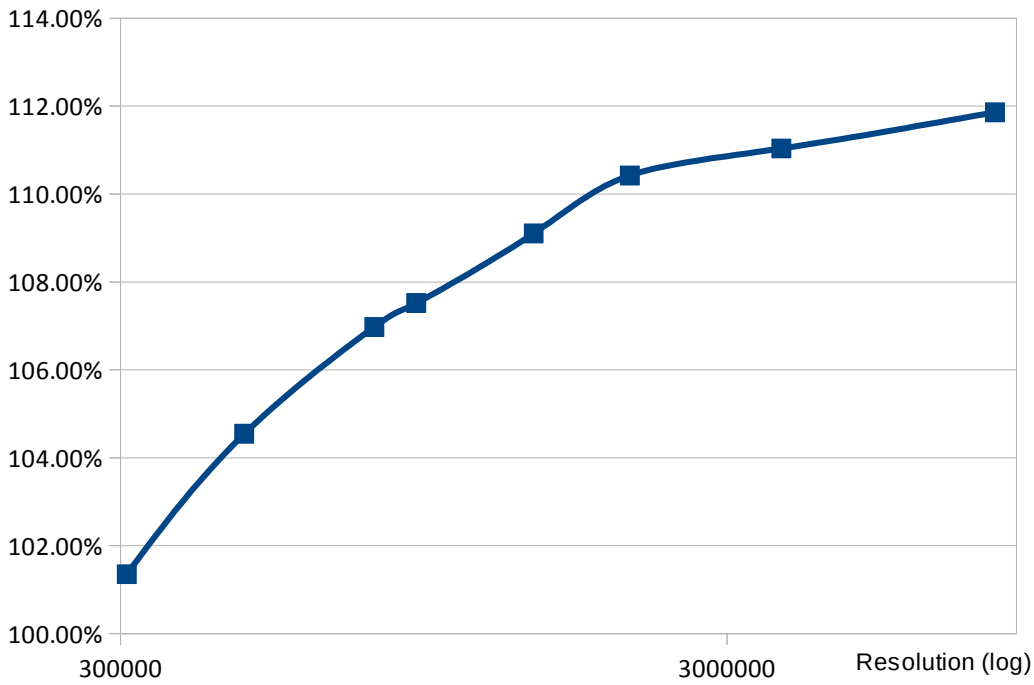

Figure 10: Speedup Vs Operand Size

The test shows that the achievable speedup with *MultiStream* is input-size dependent. This is an expected result: with MultiStream we pay *n* times the cost of setting up a computation on the device, moreover this cost is distributed among smaller tasks that are only partially overlapped (i.e. no Device-to-Host transfer overlap). As the size of a task decreases the effects of this cost becomes more evident.

4.6 Final Remarks

In this chapter we have tested *MultiStream* and we have shown that it can provide a significant speedup to the execution of all the SkePU skeletons it supports. We want to emphasize that all the experiments we run, take into account the worst-case scenario, namely: the task is produced on the host, transferred to the device for the given skeleton computation and its result is immediately gathered by the host. This means that *MultiStream* has to be seen both as an optimization targeting the GPGPU computation per-se, and as a step forward in the usage of GPGPU to offload the processing of a stream of tasks from the CPU.

In the next two chapters, we will share some thoughts about possible improvements for both *MultiStream* and SkePU and we will conclude this master thesis discussion.

5. Improvements and Future Work

In this chapter we will describe what are, in our opinion, the possible ways to improve and extend our work.

Complete MultiStream's SkePU Implementation

Probably the most obvious improvement would be a complete implementation of the *MultiStream* optimization mechanism in SkePU. As we have seen in section 4.4.2, the lack of Device-to-Host transfer overlapping has a significant impact in the potentially achievable speedup. The implementation needs to extend all the functions that access or modify elements of the smart containers from the host side. These functions trigger the Device-to-Host data copy and are currently unable to distinguish different streams.

SkePU MapOverlap Support

MultiStream may provide a speedup also in the execution of the MapOverlap/stencil skeleton. In this case, however, some data will need to be replicated during the partitioning of the task to be computed. This operation will increase both the overhead and the memory usage, so a proper cost model should be evaluated to analyze the potential benefits.

MultiStream's SkePU Backend Optimization

We have shown that the smart container partitioning is one of the sources of overhead. This is probably due to the increase of `device_pointer`s that need to be tracked for modification. Since *MultiStream* overlaps a big portion of the transfer times for sufficiently complex kernels, it could be possible to extend the auto-tuning SkePU feature to disable the *subset transfer* optimization (which is limited to 10 subsets per operand) and perform the transfer of the whole smart container. This wouldn't require the usage of multiple `device_pointer`s and may improve the overall performance.

On the same line of thinking, the auto-tuning SkePU feature could be extended to determine the best amount of CUDA Streams to use with respect to the size of each task. In this way, for smaller task, the overhead of setting up multiple skeleton computations would be lower and MultiStream could be able to provide a more consistent speedup.

It is worth to observe that our first two suggestions are the most natural extension to our work: they require a little bit of effort to determine the best possible implementation, but make use of all the notions and methods shown in this master thesis. If we had more time at our disposal, our work would have surely proceeded in those two directions.
As far as the last suggestion is concerned, it surely requires more designing effort and the precise identification and quantification of each source of overhead to build a model able to distinguish which optimization is worth to be enabled and to which degree.

6. Conclusions

In this master thesis, we have presented *MultiStream*, an optimization method that targets parallel computations on heterogeneous GPU-based systems.
We have shown how, it is possible to partition the inherent data-parallelism of a given task among two layers. The first, coarse-grained, is *MultiStream*: an implementation of the master/worker template on the host side, that asynchronously delegates the execution of sub-tasks to the GPU exploiting page-locked memory and CUDA Streams. This enables the overlapping of GPU kernel executions with data transfers between host and device memories which, in turn, significantly decreases the completion time of a given task execution.
We have implemented *MultiStream* first on SkePU, a data-parallel skeleton programming library that provides the application programmer with a high-level, platform-independent, interface for the expression of data-parallel computations. We have seen how, even with a partial implementation that omits the overlap of Device-to-Host data transfers, *MultiStream* is able to easily achieve a notable 10% speedup on all the supported skeletons (Map, MapReduce, MapArray). Moreover, we have built a cost model for each skeleton that predicts with good approximation (>92%) the expected speedup that derives from the usage of our solution.
To evaluate the potential performance improvement achievable with *MultiStream*, we have implemented a stand-alone Map skeleton that exploits the full *MultiStream* optimization without adding any noticeable overhead. We have shown how the results can be extremely promising (>60% speedup) provided an optimal implementation of the backend supporting *MultiStream*.
In conclusion, this master thesis can be intended as an additional promoter of the skeleton based parallel programming approach: with *MultiStream*, we have proven that low-level optimizations that require the knowledge of complex and platform-dependent mechanisms, such as CUDA Streams, are not limited to the usage in ad-hoc hand-written CUDA code, but can be efficiently adopted in the implementation of algorithmic skeletons and transparently offered to the application programmer.

Bibliography

[1] *NVIDIA CUDA C programming guide* v7.0, March 2015. NVIDIA Corporation http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htm

[2] Johan Enmyren, Christoph Kessler. *SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems.* Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, USA, Sep. 2010. ACM. http://www.ida.liu.se/~chrke55/skepu/SkePU-HLPP-2010.pdf

[3] Usman Dastgeer. *Performance-Aware Component Composition for GPU-based Systems*, Chapter 3. PhD thesis, Linköping Studies in Science and Technology, Dissertation No. 1581, Linköping University, May 2014. http://www.diva-portal.org/smash/record.jsf?pid=diva2:712422&dswid=-8407

[4] Usman Dastgeer, Christoph Kessler. *Smart containers and skeleton programming for GPU-based systems.* International Journal of Parallel Programming, March 2015. DOI: 10.1007/s10766-015-0357-6.

[5] *SkePU*. Department for Computer and Information Science (IDA), Linköping University, Sweden. http://www.ida.liu.se/labs/pelab/skepu/index.html

[6] Marco Danelutto. *Distributed Systems: Paradigms and Models*. Department of Computer Science, University of Pisa, September 2014. http://backus.di.unipi.it/~marcod/SPM1415/spmSept14.pdf

[7] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. *FastFlow: high-level and efficient streaming on multi-core* in Programming Multi-core and Many-core Computing Systems. S. Pllana and F. Xhafa, Wiley, 2014. http://www.di.unipi.it/~aldinuc/paper_files/2011_FF_tutorial-draft.pdf

[8] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013.

[9] Murray Cole. *Algorithmic Skeletons: structured management of parallel computation*. MIT Press, Cambdridge, MA, USA, 1989.

[10] Horacio González-Vélez and Mario Leyton. *A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers* in Software: Practice and Experience volume 40:1135–1160, 2010.

[11] M. Stokes, M. Anderson, S. Chandrasekar and R. Motta. *A Standard Default Color Space for the Internet – sRGB*, November 1996. http://www.w3.org/Graphics/Color/sRGB

[12] *Understanding Gamma Correction*. Cambridge in Colour, 2015. http://www.cambridgeincolour.com/tutorials/gamma-correction.htm

APPENDIX A

This appendix contains the source code of NevMap, a stand-alone CUDA implementation of the Map skeleton with two input operands and one output which support the full *MultiStream* optimization.

The level of abstraction isn't as high as the SkePU's one: page-locked memory allocation and synchronization between host and device memories has to be performed explicitly using the provided functions. The source code is abundantly commented.

nevmap.h:

```
001 /*
002  * nevmap.h
003  *
004  * The current implementation supports asychronous HtD and DtH
005  * data transfers and overlapping between data transfers and
006  * kernel executions making use of CUDA Streams (if data are Pinned).
007  *
008  * There are two different possible usage:
009  *
010  * *** Breadth-First Scheduling (Compute Capability 1.1) ***
011  *
012  * 1. Definition of the Map function (SkePU-like)
013  * 2. Allocation of the (in1, in2, out) datasets with
014  *    cudaHostAlloc() function
015  * 3. Work on the datasets (e.g. fill them with actual data)
016  * 4. Instantiation of a new NevMap passing the datasets,
017  *    the function and the size of the operands (all have to
018  *    be of the same size) as parameters
019  * 5. updateInX() to transfer both input to the device (non-blocking)
020  * 6. execute() of the Map Skeleton function (non-blocking)
021  * 7. updateOut() to retrieve the results on the Host side (non-blocking)
022  * 8. synch() to make sure output data are consistent before accessing
023  *    them on the Host side (blocking)
024  *
025  * *** Depth-First Scheduling (Compute Capability 2.0 or greater) ***
026  * Substitute points 5-6-7 of Breadth-First with a single call to the
027  * transferExecuteDF() function (non-blocking)
028  *
029  * Between non-blocking operations other code can be concurrently
030  *  executed on the Host side provided it has no data dependencies with
031  * the one off-loaded for GPU execution (no check is made).
032  *
033  */
034
035 #include "nevmap_kernel.h"
036
037 #ifndef NEVMAP_H_
038 #define NEVMAP_H_
039
```

```
040 #define NSTREAMS 16
041
042 /*!
043  * SkePU-like BinaryFunction macro with two input and one output
044  */
045 #define FUNC_2i1o(name, T, i1, i2, func)\
046 struct name {\
047    __device__ T CU(T i1, T i2){\
048       func\
049    }\
050 };
051
052 /*!
053  * Map Skeleton implementation
054  */
055 template<typename T, typename MapFunc>
056 class NevMap {
057    size_t m_numElements;
058    size_t numElementsPerStream;
059    size_t numElementsPerStreamSize;
060    size_t rest;
061    size_t restSize;
062    cudaStream_t streams[NSTREAMS];
063    MapFunc *m_mapFunc;
064    T *h_in1, *h_in2, *h_out;
065    T *d_in1[NSTREAMS], *d_in2[NSTREAMS], *d_out[NSTREAMS];
066
067 public:
068    NevMap(T *in1, T *in2, T *out, MapFunc *mapFunc, size_t numElements);
069    ~NevMap();
070    void setMapFunc(MapFunc *mapFunc);
071    void setIn1(T *in1);
072    void setIn2(T *in2);
073    void setOut(T *out);
074    void updateIn1();
075    void updateIn2();
076    void updateOut();
077    void updateOutAndSynch();
078    void synch();
079    void execute();
080    void execute(MapFunc *mapFunc);
081    void transferExecuteDF();
082
083 private:
084    void deviceInit();
085 };
086
087 /*!
088  * Map constructor
089  */
090 template<typename T, typename MapFunc>
091 NevMap<T, MapFunc>::NevMap(T *in1, T *in2, T *out, MapFunc *mapFunc,
092       size_t numElements) {
093    m_numElements = numElements;
094    setIn1(in1);
095    setIn2(in2);
096    setOut(out);
```

```
097    setMapFunc(mapFunc);
098    deviceInit();
099 }
100
101 /*!
102  * Map destructor
103  */
104 template<typename T, typename MapFunc>
105 NevMap<T, MapFunc>::~NevMap() {
106    //Free all Device allocated memory and destroy streams
107    for (int i = 0; i < NSTREAMS; i++) {
108       cudaFree(d_in1[i]);
109       cudaFree(d_in2[i]);
110       cudaFree(d_out[i]);
111       cudaStreamDestroy(streams[i]);
112    }
113 }
114
115 /*!
116  * Set the map function to be computed
117  */
118 template<typename T, typename MapFunc>
119 void NevMap<T, MapFunc>::setMapFunc(MapFunc *mapFunc) {
120    m_mapFunc = mapFunc;
121 }
122
123 /*!
124  * Set the first Input data.
125  * To actually transfer it to the Device
126  * a call to updateIn1() has to be made
127  */
128 template<typename T, typename MapFunc>
129 void NevMap<T, MapFunc>::setIn1(T *in1) {
130    h_in1 = in1;
131 }
132
133 /*!
134  * Set the second Input data.
135  * To actually transfer it to the Device
136  * a call to updateIn2() has to be made
137  */
138 template<typename T, typename MapFunc>
139 void NevMap<T, MapFunc>::setIn2(T *in2) {
140    h_in2 = in2;
141 }
142
143 /*!
144  * Set the Output data.
145  * After the function execution, call updateOut() to transfer
146  * results back to Host memory (non-blocking).
147  * A call to synch() is needed to ensure data consistency on
148  * the Host side before using it.
149  */
150 template<typename T, typename MapFunc>
151 void NevMap<T, MapFunc>::setOut(T *out) {
152    h_out = out;
153 }
```

```
154
155 /*!
156  * Transfer Input1 data to Device Memory (HdT)
157  */
158 template<typename T, typename MapFunc>
159 void NevMap<T, MapFunc>::updateIn1() {
160    for (int i = 0; i < NSTREAMS - 1; i++) {
161       cudaMemcpyAsync(d_in1[i], &h_in1[i * numElementsPerStream],
162             numElementsPerStreamSize, cudaMemcpyHostToDevice, streams[i]);
163    }
164    cudaMemcpyAsync(d_in1[NSTREAMS - 1],
165          &h_in1[(NSTREAMS - 1) * numElementsPerStream],
166          numElementsPerStreamSize + restSize, cudaMemcpyHostToDevice,
167          streams[NSTREAMS - 1]);
168 }
169
170 /*!
171  * Transfer Input2 data to Device Memory (HdT)
172  */
173 template<typename T, typename MapFunc>
174 void NevMap<T, MapFunc>::updateIn2() {
175    for (int i = 0; i < NSTREAMS - 1; i++) {
176       cudaMemcpyAsync(d_in2[i], &h_in2[i * numElementsPerStream],
177             numElementsPerStreamSize, cudaMemcpyHostToDevice, streams[i]);
178    }
179    cudaMemcpyAsync(d_in2[NSTREAMS - 1],
180          &h_in2[(NSTREAMS - 1) * numElementsPerStream],
181          numElementsPerStreamSize + restSize, cudaMemcpyHostToDevice,
182          streams[NSTREAMS - 1]);
183 }
184
185 /*!
186  * Transfer Output data to Host Memory (DtH).
187  * Immediately return control to CPU.
188  * It should be followed by a call to synch() as soon as
189  * the Output data is needed on the Host side.
190  */
191 template<typename T, typename MapFunc>
192 void NevMap<T, MapFunc>::updateOut() {
193    for (int i = 0; i < NSTREAMS - 1; i++) {
194       cudaMemcpyAsync(&h_out[i * numElementsPerStream], d_out[i],
195             numElementsPerStreamSize, cudaMemcpyDeviceToHost, streams[i]);
196    }
197    cudaMemcpyAsync(&h_out[(NSTREAMS - 1) * numElementsPerStream],
198          d_out[NSTREAMS - 1], numElementsPerStreamSize + restSize,
199          cudaMemcpyHostToDevice, streams[NSTREAMS - 1]);
200 }
201
202 /*!
203  * Transfer Output data to Host Memory (DtH) and Synch with CPU
204  */
205 template<typename T, typename MapFunc>
206 void NevMap<T, MapFunc>::updateOutAndSynch() {
207    updateOut();
208    synch();
209 }
210
```

```
211 /*!
212  * Execute the Map function
213  */
214 template<typename T, typename MapFunc>
215 void NevMap<T, MapFunc>::execute() {
216    for (int i = 0; i < NSTREAMS - 1; i++) {
217       kernel2i1o<<<24, 256, 0, streams[i]>>>(d_in1[i], d_in2[i], d_out[i],
218             numElementsPerStream, m_mapFunc);
219    }
220    kernel2i1o<<<24, 256, 0, streams[NSTREAMS - 1]>>>(d_in1[NSTREAMS - 1],
221          d_in2[NSTREAMS - 1], d_out[NSTREAMS - 1],
222          numElementsPerStream + rest, m_mapFunc);
223 }
224
225 /*!
226  * Transfer and Execute Depth-First
227  */
228 template<typename T, typename MapFunc>
229 void NevMap<T, MapFunc>::transferExecuteDF() {
230    for (int i = 0; i < NSTREAMS - 1; i++) {
231       cudaMemcpyAsync(d_in1[i], &h_in1[i * numElementsPerStream],
232             numElementsPerStreamSize, cudaMemcpyHostToDevice, streams[i]);
233       cudaMemcpyAsync(d_in2[i], &h_in2[i * numElementsPerStream],
234             numElementsPerStreamSize, cudaMemcpyHostToDevice, streams[i]);
235       kernel2i1o<<<24, 256, 0, streams[i]>>>(d_in1[i], d_in2[i], d_out[i],
236             numElementsPerStream, m_mapFunc);
237       cudaMemcpyAsync(&h_out[i * numElementsPerStream], d_out[i],
238             numElementsPerStreamSize, cudaMemcpyDeviceToHost, streams[i]);
239    }
240    cudaMemcpyAsync(d_in1[NSTREAMS - 1],
241          &h_in1[(NSTREAMS - 1) * numElementsPerStream],
242          numElementsPerStreamSize + restSize, cudaMemcpyHostToDevice,
243          streams[NSTREAMS - 1]);
244    cudaMemcpyAsync(d_in2[NSTREAMS - 1],
245          &h_in2[(NSTREAMS - 1) * numElementsPerStream],
246          numElementsPerStreamSize + restSize, cudaMemcpyHostToDevice,
247          streams[NSTREAMS - 1]);
248    kernel2i1o<<<24, 256, 0, streams[NSTREAMS - 1]>>>(d_in1[NSTREAMS - 1],
249          d_in2[NSTREAMS - 1], d_out[NSTREAMS - 1],
250          numElementsPerStream + rest, m_mapFunc);
251    cudaMemcpyAsync(&h_out[(NSTREAMS - 1) * numElementsPerStream],
252          d_out[NSTREAMS - 1], numElementsPerStreamSize + restSize,
253          cudaMemcpyHostToDevice, streams[NSTREAMS - 1]);
254 }
255
256 /*!
257  * Synchronizes CUDA Streams with the CPU execution
258  */
259 template<typename T, typename MapFunc>
260 void NevMap<T, MapFunc>::synch() {
261    for (int i = 0; i < NSTREAMS; i++) {
262       cudaStreamSynchronize(streams[i]);
263    }
264 }
265
266 /*!
267  * Change the Map function and execute it
```

```
268  */
269 template<typename T, typename MapFunc>
270 void NevMap<T, MapFunc>::execute(MapFunc *mapFunc) {
271    setMapFunc(mapFunc);
272    execute();
273 }
274
275 /*!
276  * CUDA Device initialization
277  */
278 template<typename T, typename MapFunc>
279 void NevMap<T, MapFunc>::deviceInit() {
280    numElementsPerStream = m_numElements / NSTREAMS;
281    numElementsPerStreamSize = numElementsPerStream * sizeof(T);
282    rest = m_numElements % NSTREAMS;
283    restSize = rest * sizeof(T);
284
285    /*
286     * Create CUDA Streams and allocate Device memory
287     */
288    for (int i = 0; i < NSTREAMS - 1; i++) {
289       cudaStreamCreate(&streams[i]);
290       cudaMalloc((void**) &d_in1[i], numElementsPerStreamSize);
291       cudaMalloc((void**) &d_in2[i], numElementsPerStreamSize);
292       cudaMalloc((void**) &d_out[i], numElementsPerStreamSize);
293    }
294    cudaStreamCreate(&streams[NSTREAMS - 1]);
295    cudaMalloc((void**) &d_in1[NSTREAMS - 1],
296          numElementsPerStreamSize + restSize);
297    cudaMalloc((void**) &d_in2[NSTREAMS - 1],
298          numElementsPerStreamSize + restSize);
299    cudaMalloc((void**) &d_out[NSTREAMS - 1],
300          numElementsPerStreamSize + restSize);
301 }
302
303 #endif /* NEVMAP_H_ */
```

nevmap_kernel.h:

```
01 #ifndef NEVMAP_KERNEL_H_
02 #define NEVMAP_KERNEL_H_
03
04 template<typename T, typename MapFunc>
05 __global__ void kernel2i1o(T *i1, T *i2, T *o, size_t numElements,
                             MapFunc mapFunc) {
06
07    size_t index = blockIdx.x * blockDim.x + threadIdx.x;
08    size_t gridSize = blockDim.x * gridDim.x;
09
10    while (index < numElements) {
11       o[index] = (*mapFunc).CU(i1[index], i2[index]);
12       index += gridSize;
13    }
14 }
15
16 #endif /* NEVMAP_KERNEL_H_ */
```

Appendix B

In this appendix the source code of the Greyscale application is presented. The functions to load or create each image, and to perform a computation on the skeleton output are left for implementation.

```
001 #include <iostream>
002 #include <ff/pipeline.hpp>
003 #include "math.h"
004 #include "skepu/vector.h"
005 #include "skepu/map.h"
006
007 struct Pixel {
008    float r;
009    float g;
010    float b;
011 };
012
013 // Map function: Ycom(Luma(Yexp(sRGB)))
014 UNARY_FUNC_CONSTANT(grey_f, Pixel, Pixel, px, lum,
015
016    //Express color components as [0,1] values
017    px.r = px.r/255;
018    px.g = px.g/255;
019    px.b = px.b/255;
020
021    //Apply gamma expansion to color components
022    if(px.r > 0.04045){
023       px.r = powf(((px.r + 0.055)/1.055), 2.4);
024    } else {
025       px.r = px.r/12.92;
026    }
027    if(px.g > 0.04045){
028       px.g = powf(((px.g + 0.055)/1.055), 2.4);
029    } else {
030       px.g = px.g/12.92;
031    }
032    if(px.b > 0.04045){
033       px.b = powf(((px.b + 0.055)/1.055), 2.4);
034    } else {
035       px.b = px.b/12.92;
036    }
037
038    //Compute luminance
039    float y = px.r * lum.r + px.g * lum.g + px.b * lum.b;
040
041    //Apply gamma compression to luminance and get back to range [0, 255]
042    if(y > 0.0031308){
043       y = powf((y * 1.055), (1/2.4)) - 0.055;
044    } else {
045       y = y * 12.92;
```

```
046    }
047    y = rintf(y * 255);
048
049    //Store luminance value to each component
050    px.r = y;
051    px.g = y;
052    px.b = y;
053    return px;
054 )
055
056 using namespace ff;
057 typedef skepu::Vector<Pixel> sv_t;
058 const Pixel LUMA = {0.2126, 0.7152, 0.0722};
059 const uint NUM_IMAGES = 60;
060 const size_t SIZE = 1920*1080;
061 const uint SHADES = 255;
062 static skepu::Map<grey_f> RGBtoGrey (new grey_f);
063
064 /*
065 * StageOne: Create the images to be processed
066 */
067 struct StageOne: ff_node {
068    void *svc(void *){
069
070       //Set the LUMA (Pixel) constant for the skepu::Map
071       RGBtoGrey.setConstant(LUMA);
072
073       /* Create tasks */
074       for(uint i = 0; i < NUM_IMAGES; i++){
075          void *t = new sv_t(SIZE);
076          sv_t *task = reinterpret_cast<sv_t *>(t);
077
078          /* Create/Load the single image as a skepu::Vector of Pixel
079
080          //Send task to the next stage
081          ff_send_out(task);
082       }
083       return EOS;
084    }
085 };
086
087 /*
088 * StageTwo: Use SkePU to convert images from RGB to Greyscale
089 * No access to the image data is made on this stage
090 */
091 struct StageTwo: ff_node {
092    void *svc(void *t){
093       sv_t *task = reinterpret_cast<sv_t *>(t);
094
095       //Compute the Map
096       RGBtoGrey(*task);
097
098       //Send task to the next stage
099       ff_send_out(task);
100       return GO_ON;
101    }
102 };
```

```
103
104 /*
105 * StageThree: CPU is free to work on different data w.r.t.
106 * what is being computed by the GPU.
107 *
108 * In this case, the task is just sent to the next stage
109 */
110 struct StageThree: ff_node {
111    void *svc(void *t){
112       sv_t *task = reinterpret_cast<sv_t *>(t);
113
114       //Send task to the next stage
115       ff_send_out(task);
116       return GO_ON;
117    }
118 };
119
120 /*
121 * StageFour: Access or modify the output of the GPGPU computation.
122 * It triggers the DtH data transfer, thus the time spent on this stage is
123 * T_DtH fully paid both with or without MultiStream
124 * (no DtH overlap support on SkePU)
125 */
126 struct StageFour: ff_node {
127    void *svc(void *t){
128
129       sv_t *task = reinterpret_cast<sv_t *>(t);
130
131       /* Access/Modify the output from the Host side */
132
133       delete task;
134
135       return GO_ON;
136    }
137 };
138
139 int main(){
140
141    //Initialize & Run FF_Pipeline
142    ff_pipeline pipe;
143    pipe.add_stage(new StageOne);
144    pipe.add_stage(new StageTwo);
145    pipe.add_stage(new StageThree);
146    pipe.add_stage(new StageFour);
147    if(pipe.run_and_wait_end()<0){
148       error("Error with FastFlow Pipeline");
149    }
150
151    return 0;
152 }
```