SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems

Johan Enmyren Christoph W. Kessler

PELAB, Department of Computer and Information Science
Linköping University, S-58183 Linköping, Sweden
{x10johen, chrke}@ida.liu.se

Abstract

We present SkePU, a C++ template library which provides a simple and unified interface for specifying data-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL. The interface is also general enough to support other architectures, and SkePU implements both a sequential CPU and a parallel OpenMP backend. It also supports multi-GPU systems.

Copying data between the host and the GPU device memory can be a performance bottleneck. A key technique in SkePU is the implementation of lazy memory copying in the container type used to represent skeleton operands, which allows to avoid unnecessary memory transfers.

We evaluate SkePU with small benchmarks and a larger application, a Runge-Kutta ODE solver. The results show that a skeleton approach to GPU programming is viable, especially when the computation burden is large compared to memory I/O (the lazy memory copying can help to achieve this). It also shows that utilizing several GPUs have a potential for performance gains. We see that SkePU offers good performance with a more complex and realistic task such as ODE solving, with up to 10 times faster run times when using SkePU with a GPU backend compared to a sequential solver running on a fast CPU.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Algorithms, Languages, Performance

Keywords Skeleton Programming, GPU, CUDA, OpenCL, Data Parallelism

1. Introduction

The general trend towards multi- and many-core based systems constitutes a disruptive change in the fundamental programming model in mainstream computing and requires rewriting of sequential application programs into parallel form to turn the steadily increasing number of cores into performance. Worse yet, there is a number of very different architectural paradigms such as homogeneous SMP-like multicores, heterogeneous multicores like Cell Broadband Engine, or hybrid CPU/GPU systems, sometimes with a very high complexity of programming such systems efficiently. Moreover, we observe a quick evolution process on the hardware side, pushing new architecture generations and variations on the market with short time intervals. The lack of a universal parallel programming model immediately leads to a portability problem.

Skeleton programming [3, 15, 16] is an approach that could solve the portability problem to a large degree. It requires the programmer to rewrite a program using so-called skeletons, predefined generic components derived from higher-order functions that can be parameterized in sequential problem-specific code, and for which efficient implementation for a given target platform may exist. Skeleton programming constrains the programmer to using only the given set of skeletons for the code that is to be parallelized or ported automatically—computations that do not fit any predefined skeleton (combination) still have to be rewritten manually. In turn, parallelism and leveraging other architectural features comes almost for free for skeleton-expressed computations, as skeleton instances can easily be expanded or bound to equivalent expert-written efficient target code that encapsulates all low-level platform-specific details such as managing parallelism, load balancing, communication, utilization of SIMD instructions etc.

In existing imperative skeleton programming environments, the parametrization of skeletons in problem-specific code can be based on one of four different techniques: On passing function variables (i.e., function pointers), on subclassing skeleton classes and defining abstract member functions in OO-based skeleton systems, on macro expansion including (C++) template programming, and on language constructs requiring a separate compiler. Today, it seems that the most promising approach is the one based on C++ templates, as it is very powerful, does not incur run-time overhead, and requires no additional tools.

In this work, we make the following contributions:

- We describe the design and implementation of SkePU, a new C++ based skeleton programming library for single- and multi-GPU systems that supports multiple back-ends, namely CUDA and OpenCL for GPUs and OpenMP for multi-core CPUs.
- We show how to optimize memory transfers for skeleton operand data by a lazy copying technique implemented in the vector data container that is used in SkePU to represent array operands.
- We show with an experimental evaluation that code ported to SkePU leads to significant speedup if run on a GPU compared to a fast CPU core.

The remainder of this paper is organized as follows: We give a short introduction to the architecture and programming of GPUs in Section 2. The design and implementation of SkePU is described

HLPP'10, September 25, 2010, Baltimore, Maryland, USA.

in Section 3. In Section 4 we report on the experimental evaluation. Section 5 reviews related work, and Section 6 concludes. Section 7 gives a short discussion of the current limitations of SkePU and proposes future work on the library.

2. NVIDIA GPU Architecture

For the past decades, the demand for faster and faster graphic acceleration has lead the way for a quick development in the area of GPU (Graphics Processing Unit) architecture. At first they were mainly used for pure graphic calculations, but when programmable shaders started to arrive, people began experimenting with utilizing the GPU's processing power for other purposes. Its potential in non-graphic related areas was recognized and in 2006 NVIDIA released the first version of CUDA, a general purpose parallel computing architecture, whose purpose was to simplify the development of applications that utilize the power of the GPU [14].

One of the big differences between a CPU and a GPU is how they are composed, or what the chip-area is used for. Since the CPU is not specialized on a specific task, it is instead aimed at being fairly general, with a lot of transistors being used for caching and flow control. The GPU, on the other hand, devotes much more space on the chip for pure floating-point calculations since having a high throughput of massively parallel computations is what graphics processing is all about. This makes the GPU very powerful on certain kinds of problems, especially those that have a data-parallel nature, preferably with much computation compared to memory transfers [14].

The CUDA architecture consists of several compute units called *SMs* (Streaming Multiprocessors). They do all the thread management and are able to switch threads with no scheduling overhead. This zero-overhead allows one to use, for example, one thread per data element in data parallel problems. The multiprocessor executes threads in groups of 32, called *warps*, but each thread executes with its own instruction address and register state, which allows for separate branching. It is, however, most efficient if all threads in one warp take the same execution path, otherwise the execution in the warp is sequentialized. Another important thing to think about is to allow simultaneous memory accesses to be *coalesced* into only one memory transaction; how this is done is described in [14].

2.1 Programming with CUDA and OpenCL

Programming for the NVIDIA CUDA architecture can be done via two platforms: NVIDIAs own CUDA platform, which is divided into CUDA runtime and CUDA driver APIs, and OpenCL. For SkePU, both CUDA runtime and OpenCL are available as backends even though they differ quite a bit in design.

When the CUDA runtime is used, device and host code are mixed in the same source. To separate them, the source code must be compiled with NVIDIAs own compiler *NVCC*. With the help of a conventional C/C++ compiler like *GCC*, it generates an executable with an embedded binary form of the device code. Uploading of code to the device and its execution is handled automatically.

OpenCL is an open standard by the Khronos group [13]. It does not specifically target NVIDIA GPUs but instead aims at being a more general standard for programming heterogeneous processing platforms. It takes a different approach to device code generation. Instead of using a separate compiler, device code is stored as strings in the program which can be used as arguments to API calls that compile and upload the code to the device at runtime.

SkePU implements both of these as backends, with the same interface and no need to change the code. The only difference is that when compiling for CUDA, the source file must have the extension .cu for the NVCC compiler to recognize it.

```
skepu::Vector<double> input(100,10);
.
.
input.flush();
```

Listing 1. Vector creation.

3. SkePU

SkePU is a C++ template library designed to make parallel programming easier with the use of higher-order functions, skeletons. It is modeled after *BlockLib* [1], a similar C library for the *IBM Cell BE*, and implements the same set of data parallel skeletons. SkePU was originally constructed with the aim of providing the same functionality as BlockLib, but instead of using the Cell processor, it was geared towards GPUs, using CUDA and/or OpenCL as backend. A large portion of the library therefore consists of GPU memory management, kernels and, in the case of OpenCL, code generation and compilation. The interface is however fairly general and does not make the library bound to only GPUs. This can also be seen in SkePU as there is a sequential CPU and an OpenMP based implementation of all the skeletons. Which variant to use can be controlled by defining either `SKEPU_CUDA`, `SKEPU_OPENCL` or `SKEPU_OPENMP` in the source code. With no defines, the single-thread CPU variant is used. Other modifications of the source code are not necessary since the interface is the same for all implementation variants.

In addition to the skeletal functions, SkePU also includes one container which must be used when doing computations with the skeletons. It is a vector/array type, designed after the STL container `vector`. Actually, its implementation uses the STL `vector` internally and therefore also has an interface mostly compatible with it. The SkePU `vector` hides GPU memory management and also uses lazy memory copying to avoid unnecessary memory operations.

3.1 Container

As mentioned earlier, the only container in the current version of SkePU is an array or `vector`. It is modeled after the STL `vector` and is largely compatible with it. However, since it uses proxy elements for some operations to be able to distinguish between reads and writes, some behavior might differ. See [12] for more information.

The SkePU `vector` keeps track of which parts of it are currently allocated and uploaded to the GPU. If a computation is done, changing the vector in the GPU memory, it is not directly transferred back to the host memory. Instead, the vector waits until an element is accessed on the host side before any copying is done (for example through the [] operator); this lazy memory copying is of great use if several skeletons are called one after the other, with no modifications of the vector by the host in between. In that case, the vectors are kept on the device (GPU) through all the computations, which greatly improves performance. Most of the memory copying is done implicitly but the vector also contains a `flush` operation which updates the vector from the device and deallocates its memory. Listing 1 shows how a vector of size 100 is created with all elements initialized to 10 and also how `flush` is called.

3.2 User Functions

To provide a simple way of defining functions that can be used with the skeletons, several preprocessor macros have been implemented that expand to the right kind of structure that constitutes the function. In SkePU, user functions are basically a `struct` with member functions for CUDA and CPU, and strings for OpenCL. Listing 2 shows one of the macros and its expansion.

```
BINARY_FUNC(plus, double, a, b,
    return a+b;
)

// expands to:

struct plus
{
    skepu::FuncType funcType;
    std::string func_CL;
    std::string funcName_CL;
    std::string datatype_CL;
    plus()
    {
        funcType = skepu::BINARY;
        funcName_CL.append("plus");
        datatype_CL.append("double");
        func_CL.append(
        "double plus(double a, double b)\n"
        "{\n"
        "    return a+b;\n"
        "}\n");
    }
    double CPU(double a, double b)
    {
        return a+b;
    }
    __device__ double CU(double a, double b)
    {
        return a+b;
    }
};
```

Listing 2. User function, macro expansion.

```
UNARY_FUNC(name, type1, param1, func)
UNARY_FUNC_CONSTANT(name, type1, param1, const1, func)
BINARY_FUNC(name, type1, param1, param2, func)
BINARY_FUNC_CONSTANT(name, type1, param1, param2, \
const1, func)
TERNARY_FUNC(name, type1, param1, param2, param3, func)
TERNARY_FUNC_CONSTANT(name, type1, param1, param2, \
param3, const1, func)
OVERLAP_FUNC(name, type1, over, param1, func)
ARRAY_FUNC(name, type1, param1, param2, func)
```

Listing 3. Available macros.

The macros available in the current version of SkePU are shown in Listing 3.

3.3 Skeleton Functions

In the object-oriented spirit of C++, the skeleton functions in SkePU are represented by objects. By overloading `operator()` they can be made to behave in a way similar to regular functions. All of the skeletons contain member functions representing each of the different implementations, CUDA, OpenCL, OpenMP and CPU. The member functions are called CU, CL, OMP and CPU respectively. If the skeleton is called with `operator()`, the library decides which one to use depending on what is available. In the OpenCL case, the skeleton objects also contain the necessary code generation and compilation procedures. When a skeleton is instantiated, it creates an environment to execute in, containing all available OpenCL or CUDA devices in the system. This environment is created as a singleton so that it is shared among all skeletons in the program.

The skeletons can be called with whole vectors as arguments, doing the operation on all elements of the vector. Another way to call them is with iterators. In that case, a start iterator and an end iterator are instead provided which makes it possible to only apply

```
BINARY_FUNC(plus, double, a, b,
    return a+b;
)

// Creates a reduction skeleton from the plus operator
skepu::Reduce<plus> globalSum(new plus);

skepu::Vector<double> input(100,10);

// Applies the skeleton to the vector input.
// Backend depends on what #defines are made.
double sum = globalSum(input);
double halfsum = globalSum(input.begin(),
                           input.begin()+50);

// Call CPU backend explicitly
double sum = globalSum.CPU(input);
```

Listing 4. Skeleton creation.

the skeleton on parts of the vector. Listing 4 shows how a skeleton is created from a user-defined function and how it is used. First a user function called `plus` is created as described in Section 3.2. Then a skeleton object is instantiated with that function as a parameter. In the current version of SkePU it needs to be provided both as a template parameter and as a pointer to an instantiated version of the user function (remember that the user functions are in fact `structs`).

As mentioned earlier, SkePU implements the same set of skeletons as BlockLib: The two common data parallel patterns *Map* and *Reduce*, a combination of those called *MapReduce*, and a variant of Map called *MapOverlap*. Apart from these, SkePU also implements another variant of Map called *MapArray*. Below is a short description of each of the skeletons.

In the Map skeleton, every element in the result vector r is a function f of the corresponding elements in one or more input vectors $v_0 \dots v_k$. The vectors have length N. A more formal way to describe this operation is: $r[i] = f(v_0[i], \dots, v_k[i])\ \forall i \in \{0, \dots, N-1\}$ In SkePU, the number of input vectors k is limited to a maximum of three ($k \leq 3$). An example of Map, which calculates the result vector as the sum of two input vectors is shown in listing 5. The output is shown as a comment at the end. A Map skeleton with the name `sum` is instantiated in the same way as described earlier and is then applied to vector `v0` and `v1` with result in `r`.

Reduction is another common data parallel pattern. The scalar result is computed by applying a commutative associative binary operator $\oplus$ between each element in the vector. With the same notation as before, reduction can be described like this: $r = v[0] \oplus v[1] \oplus \dots \oplus v[N-1]$. A reduction using + as operator would, for example, yield the global sum of the input vector. This is shown in listing 6. The syntax of skeleton instantiation is the same as before but note when calling the reduce skeleton in the line `double r = globalSum(v0)` the scalar result is returned by the function rather than returned in a parameter.

MapReduce is basically just a combination of the two above: It produces the same result as if one would first Map one or more vectors to a result vector, then do a reduction on that result. It is provided since it combines the mapping and reduction in the same computation kernel and therefore avoids some synchronization, which speeds up the calculation. Formally: $r = f(v_0[0], \dots, v_k[0]) \oplus \dots \oplus f(v_0[N-1], \dots, v_k[N-1])$. An example of MapReduce is shown in listing 7 which computes the dot product. The MapReduce skeleton is instantiated in a similar way as the Map and Reduce skeletons, but since it needs two user functions, one for the mapping part and one for reduction, two parameters are needed at instantiation time. First is the map-

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/map.h"

BINARY_FUNC(plus, double, a, b,
    return a+b;
)

int main()
{
    skepu::Map<plus> sum(new plus);

    skepu::Vector<double> v0(10,10);
    skepu::Vector<double> v1(10,5);
    skepu::Vector<double> r;

    sum(v0,v1,r);

    std::cout<<"Result: "<<r<<"\n";

    return 0;
}

// Output
// Result: 15 15 15 15 15 15 15 15 15 15
```

Listing 5. A Map example.

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/reduce.h"

BINARY_FUNC(plus, double, a, b,
    return a+b;
)

int main()
{
    skepu::Reduce<plus> globalSum(new plus);

    skepu::Vector<double> v0(1000,2);

    double r = globalSum(v0);

    std::cout<<"Result: "<<r<<"\n";

    return 0;
}

// Output
// Result: 2000
```

Listing 6. An example of a reduction with + as operator.

ping function then the reduce function. In listing 7 a MapReduce skeleton is created which will map two vectors with `mult` and then reduce the result with `plus` producing the dot product between the two vectors.

The higher order function MapOverlap is similar to a Map, but each element $r[i]$ of the result vector is a function of several adjacent elements of one input vector that reside at a certain constant maximum distance from i in the input vector. The number of these elements is controlled by the parameter `overlap`. With the notation used above: $r[i] = f(v[i-d], v[i-d+1], \ldots, v[i+d])\, \forall i \in \{0, \ldots, N-1\}$. Convolution is an example of a calculation that fits into this pattern. The edge policy, how MapOverlap behaves when a read outside the array bounds is performed, can be either cyclic or constant. When cyclic, the value is taken from the other side of the array and when constant, a user-defined constant is used. When nothing is specified, the default behavior in SkePU is constant with

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus, double, a, b,
    return a+b;
)

BINARY_FUNC(mult, double, a, b,
    return a*b;
)

int main()
{
    skepu::MapReduce<mult, plus> dotProduct(new mult,
                                            new plus);

    skepu::Vector<double> v0(1000,2);
    skepu::Vector<double> v1(1000,2);

    double r = dotProduct(v0,v1);

    std::cout<<"Result: "<<r<<"\n";

    return 0;
}

// Output
// Result: 4000
```

Listing 7. A MapReduce example that computes the dot product.

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapoverlap.h"

OVERLAP_FUNC(over, double, 2, a,
    return a[-2]*0.4f + a[-1]*0.2f + a[0]*0.1f +
           a[1]*0.2f + a[2]*0.4f;
)

int main()
{
    skepu::MapOverlap<over> conv(new over);

    skepu::Vector<double> v0(10,10);
    skepu::Vector<double> r;

    conv(v0, r, skepu::CONSTANT, (double)0);

    std::cout<<"Result: "<<r<<"\n";

    return 0;
}

// Output
// Result: 7 9 13 13 13 13 13 13 9 7
```

Listing 8. A MapOverlap example.

0 as value. In the current implementation of SkePU, when using any of the GPU variants of MapOverlap, the maximum overlap that can be used is limited by the shared memory available to the GPU, and also the maximum number of threads per block. These two factors typically limit the overlap to < 256. An example program that does a convolution with the help of MapOverlap is shown in listing 8. Note that the indexing is relative to the element calculated, $0 \pm overlap$. A MapOverlap skeleton is instantiated with `over` as user function and is then called with vector `v0` as input and vector `r` as result. It is also using the constant `0` at the edges, decided by the parameters `skepu::CONSTANT` and `(double)0`.

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/maparray.h"

ARRAY_FUNC(arr, double, a, b,
    int index = (int)b;
    return a[index];
)

int main()
{
    skepu::MapArray<arr> reverse(new arr);

    skepu::Vector<double> v0(10);
    skepu::Vector<double> v1(10);
    skepu::Vector<double> r;

    //Sets v0 = 1 2 3 4...
    //     v1 = 9 8 7 6...
    for(int i = 0; i < 10; ++i)
    {
        v0[i] = i+1;
        v1[i] = 10-i-1;
    }

    reverse(v0, v1, r);

    std::cout<<"Result: " <<r <<"\n";

    return 0;
}

// Output
// Result: 10 9 8 7 6 5 4 3 2 1
```

Listing 9. A MapArray example that reverses a vector

MapArray is yet another variant of Map. It produces a result vector from two input vectors where each element of the result, $r[i]$, is a function of the corresponding element of one of the input vectors, $v_1[i]$ and any number of elements from the other input vector v_0. This means that at each call to the user defined function f, which is done for each element in v_1, all elements from v_0 can be accessed. The notation for accessing an element in v_0 is the same as arrays in C. $v_0[i]$ where i is a number from 0 to the length of v_0. Formally: $r[i] = f(v_0, v_1[i]) \forall i \in \{0, \ldots, N-1\}$. MapArray was first devised to avoid the overlap constraints of MapOverlap and to provide a simple yet powerful way of mapping several elements of an input vector. Listing 9 shows how MapArray can be used to reverse a vector by using $v_1[i]$ as index to v_0. A MapArray skeleton is instantiated and called with `v0` and `v1` as inputs and `r` as output. `v0` will be corresponding to parameter `a` in the user function `arr` and `v1` to `b`. Therefore, when the skeleton is applied, each element in `v1` can be mapped to any number of elements in `v0`. In listing 9, `v1` contains indexes to `v0` of the form 9, 8, 7..., therefore, as the user function `arr` specifies, the first element in `r` will be `v0[9]` the next `v0[8]` etc, resulting in a reverse of `v0`.

3.4 Dependencies

SkePU does not use any third party libraries except for CUDA and OpenCL. It does however make some use of the C++ standard library, especially STL which must be available for the library to compile. If either CUDA or OpenCL is to be used, of course they must also be installed. CUDA programs need to be compiled with NVCC since CUDA support is provided with the CUDA runtime API. SkePU is a template library, which means that there is no need to link against a precompiled library, only include the header files.

3.5 Multi-GPU support

SkePU has support for carrying out computations with the help of several GPUs on a data-parallel level. It utilizes the different GPUs by dividing the input vectors equally amongst them and doing the calculations in parallel on the devices. Here CUDA and OpenCL differ a lot. In OpenCL, one CPU thread can control several GPUs, by switching queues. In CUDA, or to be more precise, in the CUDA runtime system which SkePU is based on, this is not possible. Instead, each CPU thread is bound to one device. To make multi-GPU computation possible, several host threads must then be created. This is done in SkePU, but in the current version new threads are started for each skeleton call and bound to devices; this binding can take a lot of time, and hence the multi-GPU support in SkePU with CUDA is not very efficient. With OpenCL however, it works much better.

By default, SkePU will utilize as many GPUs as it can find in the system; however, this can be controlled by defining `SKEPU_NUMGPU`. Setting it to 0 makes it use its default behavior. Any other number represents the number of GPUs it should try to use.

4. Evaluation

All of the following evaluations were performed on a server equipped with 2 quad-core Intel(R) Xeon(R) CPU E5520 clocked at 2.27GHz, making a total of 8 cores available on the CPU side. Further it had two NVIDIA Corporation GT200 [Tesla C1060] GPUs installed. It ran Linux with kernel version 2.6.33 and GCC 4.5 (for compilation with NVCC, GCC 4.3 was used due to problems with using STL together with CUDA code with newer versions of GCC). CUDA version 3.0 was used with NVIDIA drivers 195.36.15 and their included OpenCL implementation.

Since there are problems with running multi-GPU using CUDA as backend in the current version of SkePU (see Section 3.5 for an explanation), no evaluation was done with this combination. Instead, the multi-GPU support was evaluated using OpenCL.

4.1 Time Distribution

To see how the execution time is distributed when calling a skeleton with a GPU backend, some measurements have been made on small test cases and the result can be seen in figure 1. Time was measured with OpenCL as backend. The test functions are shown in Listing 10. The dominating time for all skeletons is the memory transfer time; this is of course dependent on how computation-heavy the user-defined functions are, and since the test functions are quite small, this result is expected. It is however important to note this difference in time since eliminating unnecessary memory transfers is a key factor in reaching good performance.

4.2 Gaussian Blur

The MapOverlap skeleton was tested with a common operation in computer graphics, Gaussian blur. It is performed by convolving the image with a Gaussian function producing a new smoother och blurred image. The method basically calculates the new value of the pixel based on its own and its surrounding pixel values. It can be done either in two dimensions, for each pixel access a square around it, or in one dimension by running two passes over the image, one row-wise and one column-wise. Since SkePU at the moment only works with a one dimensional data structure, the second approach was used. When calculating a pixel value, the surrounding pixels are needed but only in a limited neighbourhood. This fits well into the calculation pattern of the MapOverlap skeleton. MapArray was also used to restructure the array from being saved row-wise, to column-wise. The blurring calculation then becomes: MapOverlap to blur horizontally, MapArray to restructure image

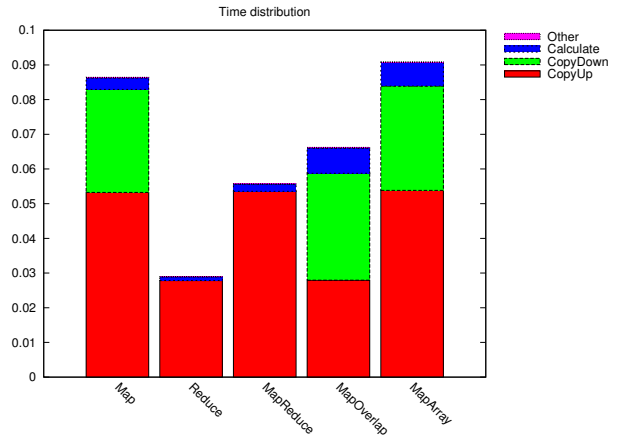


Figure 1. Time distribution for the implemented skeletons. OpenCL backend.

```
BINARY_FUNC(map_test, double, a, b,
    return ( fmax(a,b)*(a-b) );
)

BINARY_FUNC(reduce_test, double, a, b,
    return ( a + b );
)

OVERLAP_FUNC(mapoverlap_test, double, 5, a,
    return a[-5]*0.5f + a[-4]*0.6f +
           a[-3]*0.7f + a[-2]*0.8f +
           a[-1]*0.9f + a[0] + a[1]*0.9f +
           a[2]*0.8f + a[3]*0.7f + a[4]*0.6f + a[5]*0.5f;
)

ARRAY_FUNC(maparray_test, double, a, b,
    int index;
    index = (int)b;
    return ( a[index] );
)
```

Listing 10. Functions used in time distribution evaluation.

and MapOverlap to blur vertically. The image was first loaded into a vector with padding between rows.

Timing was only done on the actual blur computation, not including the loading of images and creation of vectors. For CUDA and OpenCL, the time for transferring the image to the GPU and copying the result back is included. The filtering was done with two passes of a 19 value filter kernel which can be seen in Listing 11. For simplicity only grayscale images of quadratic sizes were used in the benchmark. The result can be seen in figure 2 where 2(a) shows time when applying the filter kernel *once* to the image and 2(b) when applying it nine times in a row, resulting in heavier blur. We see that, while faster than the CPU variant, CUDA and OpenCL versions are slower than the one using OpenMP on 8 CPU cores for one filtering. This is due to the memory transfer time being much larger than the actual calculation. In figure 2(b) however filtering is done nine times which means more computations and less memory I/O due to the lazy memory copying of the vector. Then the two single GPU variants outperform even the OpenMP version. Since there is a data dependency in the MapOverlap skeleton when running on multiple-GPUs, we also see that running this configuration loses a lot of performance when applying MapOverlap several times in a row because it needs to transfer data between the GPUs, via the host.

```
OVERLAP_FUNC(blur_kernel, int, 19, a,
    return (a[-9] + 18*a[-8] + 153*a[-7] + 816*a[-6] +
            3060*a[-5] + 8568*a[-4] + 18564*a[-3] +
            31824*a[-2] + 43758*a[-1] + 48620*a[0] +
            43758*a[1] + 31824*a[2] + 18564*a[3] +
            8568*a[4] + 3060*a[5] + 816*a[6] + 153*a[7] +
            18*a[8] + a[9])>>18;
)
```

Listing 11. User function used by MapOverlap when blurring an image.

4.3 Dot Product

The performance of the MapReduce skeleton was measured by calculating the dot product of two vectors. It was then compared to the performance of the `cublasDdot()` function in the CUBLAS library provided by NVIDIA. SkePU was compiled both with CUDA and OpenCL backends and all the calculations were done with double precision. The time of only the dot product calculation was measured, *excluding* the time of copying the vector. The results can be seen in Figure 3 where the time is a total time of 1000 calls to the dot product routine. We see that the performance of the SkePU CUDA version is very similar to `cublasDdot()` although CUBLAS performs slightly better in the course of 1000 runs. The SkePU variant with OpenCL as backend using only one GPU is a bit slower while the multi-GPU variant shows its potential, especially at larger vector sizes. Using two GPUs it shows a speedup of just under 2.

4.4 A Runge-Kutta ODE Solver

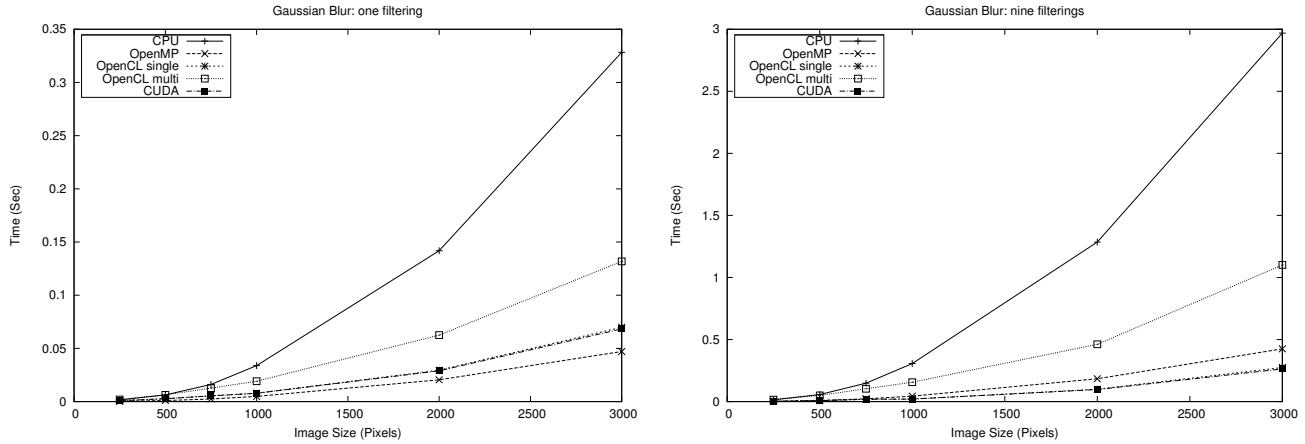
To see how well SkePU performed in a more realistic setting with a more complex task, a sequential Runge-Kutta ODE solver was ported to GPU using the SkePU library. The original code used for the porting is part of *LibSolve*, a library of various Runge-Kutta solvers for ODEs by Korch and Rauber [9]. This is the same library used by BlockLib in [1] for evaluation.

LibSolve contains several Runge-Kutta implementations, iterated and embedded ones, as well as implementations for parallel machines using shared or distributed memory. Similarly to BlockLib, the simplest default sequential implementation was used for the port to SkePU, however other solver variants were used unmodified for comparison.

The LibSolve package contains two ODE test sets, one called BRUSS2D which is based on the two-dimensional brusselator equation. The other one is called MEDAKZO, the medical Akzo Nobel problem [9]. BRUSS2D consists of two variants depending on the ordering of grid points, BRUSS2D-MIX and BRUSS2D-ROW. For evaluation of SkePU only BRUSS2D-MIX was considered. Four different grid sizes (problem size) were evaluated, 250, 500, 750 and 1000.

The porting was fairly straight forward since the default sequential solver in LibSolve is a conventional Runge-Kutta solver consisting of several loops over arrays sized according to the problem size. These loops could instead be replaced by calls to the Map, Reduce and MapReduce skeletons. The right hand side evaluation function was implemented with the MapArray skeleton.

As mentioned earlier, the benchmarking was done using the BRUSS2D-MIX problem with four different problem sizes (N=250, N=500, N=750 and N=1000). In all tests the integration interval was 0-4 (H=4) and time was measured with LibSolves internal timer functions, which on UNIX systems uses `gettimeofday()`. The different solver variants used in the testing were:



(a) Average time of blurring quadratic greyscale images of different sizes. Gaussian kernel is applied *one* time to the image.

(b) Average time of blurring quadratic greyscale images of different sizes. Gaussian kernel is applied *nine* times to the image.

Figure 2. Average time of blurring images of different sizes. Average of 100 runs.

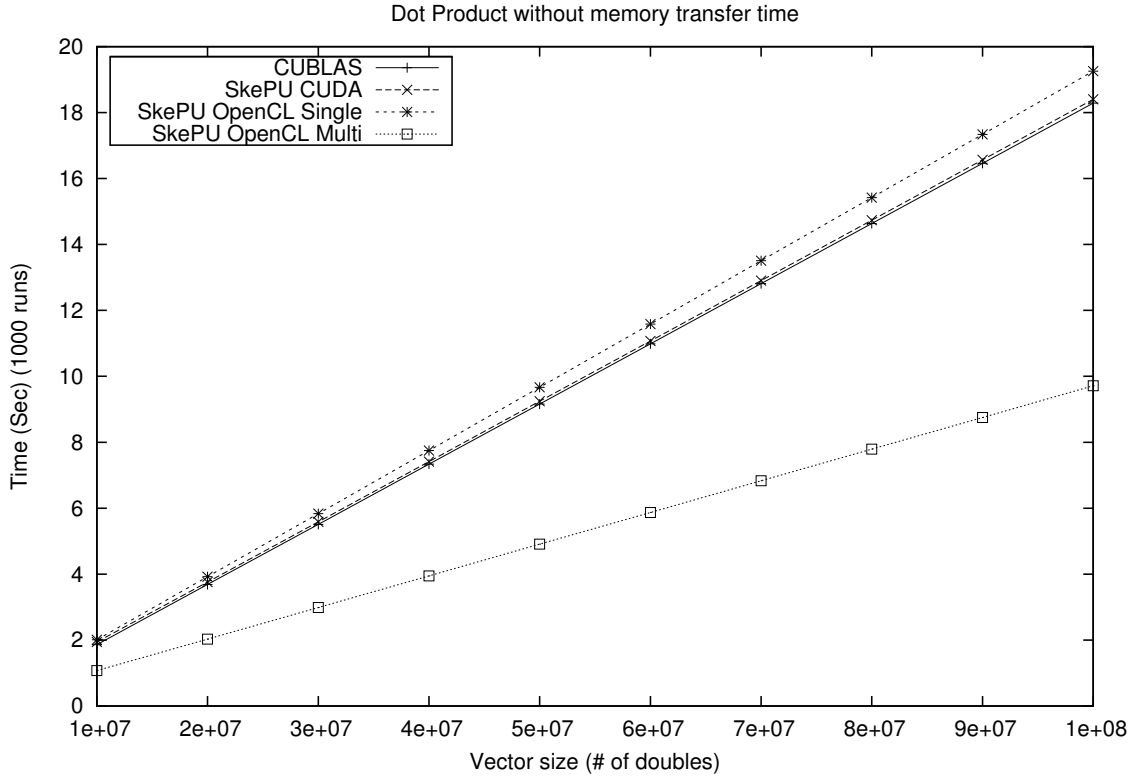


Figure 3. Time for 1000 runs of dot product calculation without host-to-device-memory transfer time.

ls-seq-def: The default sequential implementation in LibSolve.

ls-seq-A: A slightly optimized variant of ls-seq-def.

ls-shm-def: The default shared memory implementation in LibSolve. It uses pthreads and were run with 8 threads, one for each core of the benchmarking computer.

ls-shm-A: A slightly optimized variant of ls-shm-def. It also uses pthreads and were run with 8 threads.

skepu-CL: SkePU port of ls-seq-def using OpenCL as backend and running on *one* Tesla C1060 GPU.

skepu-CL-multi: SkePU port of ls-seq-def using OpenCL as backend and running on *two* Tesla C1060 GPU.

skepu-CU: SkePU port of ls-seq-def using CUDA as backend and running on *one* Tesla C1060 GPU.

skepu-OMP: SkePU port of ls-seq-def using OpenMP as backend and utilizing 8 threads.

skepu-CPU: SkePU port of ls-seq-def using the default CPU backend.

CU-hand: A "hand" implemented CUDA variant. It is similar to the SkePU ports however no SkePU code was utilized. Instead CUBLAS functions were used where applicable and some hand made kernels.

The result can be seen in Figure 4 and Table 1. The two slowest are the sequential variants (ls-seq-def and ls-seq-A), with ls-seq-A of course performing slightly better due to the optimizations. LibSolves shared memory solvers (ls-shm-def and ls-shm-A) shows a great performance increase compared to the sequential variants with almost five times faster running time for the largest problem size (N=1000).

We also see that the SkePU CPU solver is comparable to the default LibSolve sequential implementation and the OpenMP variant is similar to the shared memory solvers. The SkePU OpenCL and CUDA ported solvers are however almost 10 times faster than the sequential solvers for the largest problem size. The reason for this is that all the calculations of the core loop in the ODE solver can be run on the GPU, without any memory transfers except once in the beginning and once at the end. This is done implicitly in SkePU since it is using lazy memory copying. However, the SkePU multi-GPU solver does not perform as well, the reason here also lies in the memory copying. Since the evaluation function needs access to more of one vector than what it has stored in GPU memory (in multi-GPU mode, SkePU divides the vectors evenly among the GPUs), some memory transfers are needed: First from one GPU to host, then from host to the other GPU; this slows down the calculations considerably.

Comparing the "hand" implemented CUDA variant, we see that it is similar in performance to skepu-CU with CU-hand being slightly faster (approximately 10%). This is both due to the extra overhead when using SkePU functions and some implementation differences.

There is also a start-up time for the OpenCL implementations during which they compile and create the skeleton kernels. This time (≈5-10 seconds) is not included in the times presented here since it is considered an initialization which only needs to be done once when the application starts executing.

5. Related Work

A lot of work and research have been made in the area of skeletal parallel programming. With the arrival of CUDA and OpenCL, which has provided an easy way of utilizing the parallel processing power of graphics hardware, the skeleton approach has also been tried in this fairly new area of parallel computing. The development of SkePU, which was presented in this paper, has been inspired by several other similar projects that exist today.

Thrust [7] is an open source project whose goal is to provide STL like functionality in parallel using CUDA on NVIDIA graphics cards. It is in the form of a C++ template library and implements functionality like transform (map), reduction, prefix-sum (scan), sorting etc. It also uses a vector container that algorithms operate on. There is also a possibility to define your own functions to be used in several operations.

CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum ("scan"), parallel sort and parallel reduction [6]. It does not however provide higher-order functions which can take any user defined function as an input.

One approach to creating a more generic skeleton framework for GPGPU programming is made by Sato and Iwasaki [17]. Instead of making a pure library, they introduce the skeletons as functions to the C programming language; however, these skeletons are transformed to CUDA code by their own compiler and so, in a way, constitute a new programming language. The compiler can also generate equivalent C code using macros and it is therefore entirely C compatible. One advantage of using a separate compiler is that optimizations can be built in. This is also done by the framework proposed in [17] which optimizes the use of the skeleton functions.

So far SkePU implements one container, a vector type which is built around the STL vector and is largely inspired by *CuPP* [2]. CuPP is however not a skeleton library but rather an abstraction from CUDA which integrates better with the C++ programming language and hides things like memory management.

In [8] an implementation of the `Parallel_for` skeleton is described. It is a multi-target implementation designed to work on both CPU and GPU.

SkelCL [18] is a work in progress at the University of Münster. It is a skeleton library implemented using OpenCL and is similar to SkePU in functionality. It can also be used with multiple GPUs.

One of the main differences between SkePU and the libraries described above is that SkePU can be compiled with several backends. Most other works has all used either NVIDIAs CUDA framework or OpenCL for their implementations. SkelCL and SkePU are to our knowledge also the only skeleton libraries that exists for OpenCL today. SkePU also tries to seamlessly integrate multi-GPU functionality in its implemented skeletons.

6. Conclusions

Parallel programming with skeletons has been successfully applied to various architectures and with different libraries as backends [4, 5, 10, 11]. We have shown in this paper that it is also possible to use this approach for doing calculations in parallel on GPUs with good results.

We have presented SkePU, a C++ template library which provides a simple and unified interface for doing data-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL. The interface is also general enough to support other architectures and SkePU implements both a sequential CPU and a parallel OpenMP backend.

We have seen that copying data between the host and the GPU can be a bottleneck and therefore a container which uses lazy memory copying has been implemented to avoid unnecessary memory transfers.

In the Gaussian Blur benchmark, this lazy memory copying was utilized and we saw that using the GPU backends offers good performance, especially when the amount of computation is large (filtering the image several times in a row). The dot product benchmark compared the SkePU OpenCL and CUDA backends to a specialized library in the form of CUBLAS. It showed that the over-

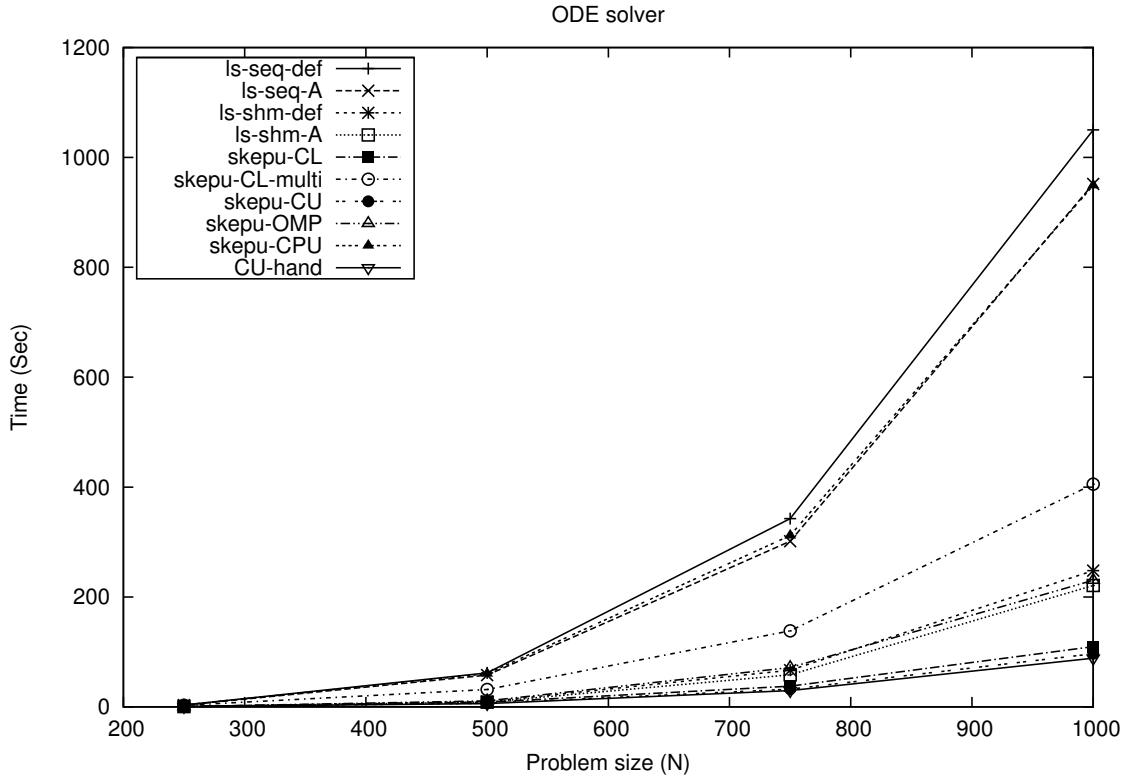


Figure 4. Times for running different LibSolve solvers for N =250,500,750 and 1000 with the BRUSS2D-MIX problem.

N	ls-seq-def	ls-seq-A	ls-shm-def	ls-shm-A	skepu-CL	skepu-CL-multi	skepu-CU	skepu-OMP	skepu-CPU	CU-hand
250	3.42	3.05	0.51	0.71	0.77	3.00	0.60	0.51	3.32	0.53
500	61.81	57.88	9.47	8.71	8.68	31.72	6.70	11.08	60.31	6.10
750	342.54	301.46	66.89	58.44	37.82	138.37	32.32	71.70	311.82	29.60
1000	1050.14	951.60	248.12	221.00	109.50	405.24	96.65	230.54	949.19	88.50

Table 1. Times for running different LibSolve solvers for increasing N with the BRUSS2D-MIX problem. Time in seconds.

head of using SkePU is quite small, and that a multi-GPU backend have a potential for performance gains.

From the LibSolve benchmark, we can see that SkePU offers good performance with a more complex and realistic task such as ODE solving with up to 10 times faster run times when using SkePU with a GPU backend compared to a sequential solver.

7. Limitations and Future Work

The work on SkePU has only just started and so there are a few limitations with the library. First it only supports a one-dimensional data structure (`skepu::vector`) and the skeletons can only operate on that kind of data. This makes it harder and less effective to use SkePU for image applications, matrix operations and other problems that have a two-dimensional structure.

The updates of a vector, from device to host, is currently limited to updating the entire vector, not parts of it. This have some performance implications, especially when there are data dependencies in the multi-GPU implementation. One example is if MapOverlap is applied several times in a row on the same vector. With a single-GPU this is done without transferring the vector back to the host between calls. In the multi-GPU case however, since the overlap creates a data dependecy between the GPUs, the entire vector needs to be transferred back and forth between the MapOverlap calls.

One skeleton call can not utilize several backends at once, only one is used at a time. You can however define both `SKEPU_OPENCL` and `SKEPU_OPENMP` for example and manually specify to use either of them for each skeleton call.

There is no task parallelism supported. Especially with multi-GPU support, there is an opportunity for this kind of parallelism; it has however not been implemented yet.

SkePU is currently work in progress and several additions to the library are planned for. More skeletons will be added, especially the data-parallel operation *Scan*, also known as *prefix-sum*. A two-dimensional data structure, *Matrix*, is to be added and some of the skeletons updated to use this new data-type. Hybrid parallelization to allow mixing of several backends and some kind of optimization to utilize them in a good way is also on the list of future functionality.

Acknowledgments

This work was funded by EU FP7, project PEPPHER, grant #248481 (www.peppher.eu), and SSF project ePUMA. We would

like to thank Mattias Korch and Thomas Rauber for sharing their ODE solver library [9] with us.

References

[1] M. Ålind, M. V. Eriksson, and C. W. Kessler. BlockLib: A skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-031-9. doi: http://doi.acm.org/10.1145/1370082.1370088.

[2] J. Breitbart. CuPP - A framework for easy CUDA integration. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. doi: http://dx.doi.org/10.1109/IPDPS.2009.5160937.

[3] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman and MIT Press, 1989.

[4] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2003.12.002.

[5] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. QUAFF: Efficient C++ design for parallel skeletons. *Parallel Comput.*, 32(7):604–615, 2006. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2006.06.001.

[6] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDPP: CUDA Data Parallel Primitives Library. `http://gpgpu.org/developer/cudpp`, 2009.

[7] J. Hoberock and N. Bell. Thrust: C++ Template Library for CUDA . `http://code.google.com/p/thrust/`, 2009.

[8] W. Kirschenmann, L. Plagne, and S. Vialle. Multi-target C++ implementation of parallel skeletons. In *POOSC '09: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 1–10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-547-5. doi: http://doi.acm.org/10.1145/1595655.1595662.

[9] M. Korch and T. Rauber. Optimizing locality and scalability of embedded runge–kutta solvers using block-based pipelining. *Journal of Parallel and Distributed Computing*, 66(3):444–468, 2006. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2005.09.003.

[10] H. Kuchen. A skeleton library. *Euro-Par 2002 Parallel Processing*, pages 85–124.

[11] M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:289–296, 2010. ISSN 1066-6192. doi: http://doi.ieeecomputersociety.org/10.1109/PDP.2010.26.

[12] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 020163371X.

[13] A. Munshi. The OpenCL specification version 1.0. *Khronos OpenCL Working Group*, 2009.

[14] Nvidia. CUDA Programming Guide Version 2.3.1. *NVIDIA Corporation*, 2009.

[15] S. Pelagatti. *Structured development of parallel programs*. Taylor & Francis, Inc., Bristol, PA, USA, 1998. ISBN 0-7484-0759-6.

[16] F. A. Rabhi and S. Gorlatch, editors. *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, London, UK, 2003. ISBN 1-85233-506-8.

[17] S. Sato and H. Iwasaki. A skeletal parallel framework with fusion optimizer for gpgpu programming. In *APLAS '09: Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, pages 79–94, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10671-2. doi: http://dx.doi.org/10.1007/978-3-642-10672-9_8.

[18] M. Steuwer, P. Kegel, and S. Gorlatch. Towards a portable multi-gpu skeleton library. Private Communication, June 2010.