Department of Physics and Astronomy
University of Heidelberg

Master thesis in Physics

Automated partitioning of data-parallel programs

With a static code analysis detecting memory access patterns

Christoph Julian Klein
born in Menden (Sauerland)

November 30, 2016

With the supervision of

Prof. Dr.-Ing. Ulrich Brüning
&
JProf. Dr. Holger Fröning

at the Institute of Computer Engineering (ZITI)

Automated partitioning of data-parallel programs

This thesis presents a workflow, which deploys polyhedral compilation techniques to analyze memory access patterns in OpenCL kernels, and automatically transforms them & the belonging host code, to enable a partitioned execution of the grid on multiple devices within one node. The used formalism deals with regular quasi-affine memory accesses within the kernels, which can be obtained with the Polly and managed with the Integer Set Library (ISL). The workflow makes use of the LLVM toolstack and includes a self-written library for calculations, which must be done at application runtime. With consideration of mandatory result merging processes, and inter kernel dependency resolutions, three test cases have been performed with virtual zero user input to evaluate the scalability and the overhead of the runtime library. With 16 GPUs the application speedup was measured to be 14.9 for the matrix multiplication, 4.2 for the stencil code, and 9.9 for the N body code in comparison to the original single GPU execution. This thesis belongs to the Mekong project, which aims to unburden multi GPU programming by simulating a one device environment to the programmer.

Automated partitioning of data-parallel programs

Diese Thesis präsentiert eine Software, welche polyhedrale Compiliertechniken verwendet um Speicherzugriffsmuster in OpenCL GPU Codes zu ermitteln, um diese, sowie die zugehörigen instruierenden CPU Codes, automatisch für eine Ausführung auf mehreren GPUs, innerhalb eines Computers, zu transformieren. Der eingeführte Formalismus berücksichtigt, innerhalb des GPU Codes, reguläre quasi-affine Speicherzugriffe, welche unter Verwendung der Polly Bibliothek und Integer Set Library extrahiert und gehandhabt werden. Die Software verwendet außerdem die LLVM Compilerinfrastruktur und stellt eine Bibliothek zur Verfügung, die Berechnungen, die notwendigerweise zur Programmlaufzeit durchgeführt werden müssen, ausführt. Unter Berücksichtigung der korrekten Rückführung von Ergebnissen & Auflösung von inter Kernelabhängigkeiten wurden drei Testfälle, quasi ohne zusätzliche Benutzereingaben, durchgeführt, um die Skalierbarkeit und den ‚durch die Laufzeitbibliothek eingeführten, zusätzlichen Rechenaufwand zu evaluieren. Mit 16 verfügbaren Grafikkarten wurde eine Beschleunigung um Faktor $14,9$ für die Matrixmultiplikation, $4,2$ für den Stencil-Code und $9,9$ für den N-Body-Code im Vergleich zur ursprünglichen Ausführungszeit gemessen. Die präsentierte Arbeit gehört zu dem Mekong Projekt, dessen Ziel die Vereinfachung von Multi-GPU-Programmierung, durch simulieren einer singulären GPU Umgebung, darstellt.

Contents

List of Figures

List of Tables

0.1. Acknowledgements

I want to reveal my gratefully acknowledgement regarding Prof. Dr.-Ing. Ulrich Brüning as my first supervisor, who made my thesis and my collaboration at project Mekong possible. Moreover I express my deep gratitude towards JProf. Dr. Holger Fröning as my second supervisor and head of the project Mekong. Furthermore I give thanks to Alexander Matz as my advisor, who was my every day helper on practical issues. Lastly, I want to mention Giada Covini, who supports me mentally at all times.

1. Introduction

This chapter presents a short way along the milestones of parallel programming to state of the art GPU programming. The first section shall explain why GPU programming is inevitable in order to keep further performance improvement; and what Mekong's key points contributing to that development look like. The successive sections position the work of this thesis within the overall project and state its contributions.

1.1. Motivation

1.1.1. For Parallel Programming

Commonly known programming languages are based on a model, which is translated to hardware by assuming a von Neumann architecture[28]. Programming such a system with one CPU, one bus, and one memory unit, is popular for its deterministic sequential behaviour. Thus understanding each program step is relatively facile.

In times of Moore's Law[17] economy and science proceed towards hardware performance improvement. Besides other limiting factors there is a physical power wall preventing a linear relation between power consumption and processor frequency. To continue enhancement developing multi core architectures is inevitable. This creates the necessity for multi core programming models like Pthreads[9]. In a reasonably written Pthreads program there exist more parallel executable tasks than there are threads. Thus one thread calculates multiple output elements. If you increase the number of threads until it is much larger than the number of physical processors, you will end up with a performance decrease due to expensive context switching on a central processing unit (CPU). The most nearby solution would be to increase the number of processors, which is limited by the fact that multiple CPU caches sharing one global memory must be kept coherent. Therefore a more cache-relaxed model like the bulk-synchronous parallel (BSP) one[25] is needed. It introduces the necessity for parallel slackness and cheap context switching in order to deploy a larger number of threads than there are processing units. As a consequence memory latency can be hidden efficiently. These features are appropriately implemented on a graphical processing unit (GPU), which matches a many core architecture with one global memory and relaxed cache coherence. Moreover GPUs are more efficient in terms of consumed energy per floating point operation (FLOP).

1.1.2. For Mekong

As you can see in section 2.1 programming a GPU requires additional effort and a parallel way of thinking in comparison to sequentially programming a von Neumann

architecture[28]. Besides latterly mentioned point novices struggle with the manually[1] instructed memory copy operations (due to the separated host and device memory) and the asynchronous kernel execution. The complexity of the stated arguments grows even larger if you want to program *multiple* devices in a system, because each GPU has to be instructed individually. This can be achieved e.g. in CUDA by calling the `cudaSetDevice` function explicitly every time before subsequent function calls should be executed on the chosen device. Most programmers experience this procedure as a lavish and cumbersome one.

Unburdening multi GPU programming by simulating a one GPU environment to the user represents a core task of the Mekong project. Likewise the Mekong River is split into multiple smaller parts at the Mekong Delta, one long term goal of Mekong is the automatic parallelization of a single GPU program to a multiple GPU deploying one. This should be achieved by analyzing and modifying the input program at compile time with virtual zero user interaction. The main benefit of using Mekong's workflow will be the increase in the programmer's productivity, rather than a better performance in comparison to human-written multi GPU program.

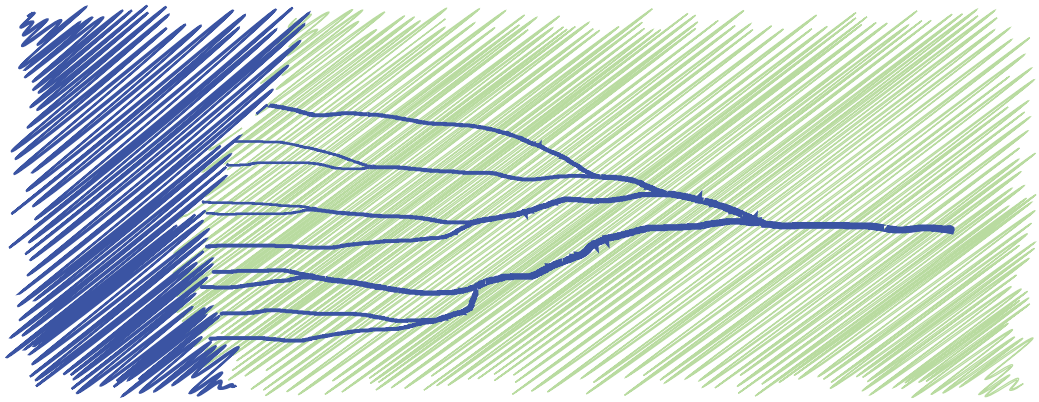


Figure 1.: Sketch of the Mekong Delta.

1.2. Concept & Methodology

The work done in this thesis includes the static partitioning of kernels to utilize multiple GPUs, which includes index recalculations in the device code; and function transformations in the host code. Problems which could only be solved at application runtime have been located into a runtime library, which interacts with the results of Mekong's static parts. All implemented functions operating on a device use the CUDA Driver API, e.g. to instruct data movements[2] or kernel calculations. The order of the sections in this thesis are aligned with the chronological processing of

[1] Universal Address Space (UVA)[10] is an CUDA feature, which avoids manual memory copy operations. UVA is supposed to increase productivity but not performance.

[2] If the used devices have a DMA engine, all load and store instructions will be executed deploying that engine.

an input application in Mekong's workflow. Thus the thesis starts with techniques and tasks regarding the compilation of a program, and continues with explaining & evaluating the application runtime behaviour. Lastly, I mention future work and make a conclusion.

Input applications are considered to use only regular memory accesses in their kernels. A memory access is defined as regular if the accessed index can be represented as a quasi affine function of the thread ID, block ID, block size, grid size, constants, and *scalar* kernel arguments. Below you can see an example of a regular and a non-regular memory access inside a kernel.

```
///// REGULAR MEMORY ACCESS //////
kernel(float* devicePtr, int N) {

  ...
  devicePtr[2 * id + N + 4];
  ...
}
```

```
//// IRREGULAR MEMORY ACCESS /////
kernel(float* devicePtr, int*
    map, int N) {
  ...
  devicePtr[map[id]];
  ...
}
```

1.3. Contributions

Overall this thesis explains the functionality of the implemented software, which consists of two parts: firstly, analysis and transformation passes, which operate at the compile time of the input code. And secondly, a library addressing problems at runtime. Writing, testing, and developing the software results in the following contributions:

- The integration of the Polly library into a custom LLVM pass for analysis of quasi-affin regular memory access patterns of OpenCL kernels at compile time.
- LLVM custom passes, which modify the host and device code automatically to enable a grid partitioned execution on multiple devices, which is based on a manually given splitting dimension.
- A library written in C++, which addresses tasks at runtime of the transformed input application. This includes a grid partition algorithm, data merging processes, and inter kernel dependency resolutions deploying the Integer Set Library (ISL), and making use of the results gained by the analysis passes.
- A proof of functionality and performance analysis using multiple test cases.
- The introduction of concepts & formalisms regarding performance improvement & cancelation of constraints in the future.

2. Background

This chapter will provide information about how parallel models (mentioned in section 1.1) can be implemented and executed on hardware using programming models. Moreover you will get an impression of compilation in general and in specific how LLVM relates to that.

2.1. Programming a GPU

Popular GPU programming models are CUDA[18] and OpenCL[8], which require a CPU code to instruct the device, and a device code, which will actually be executed by the GPU. Both models are very similar and follow the same concept:

firstly, the programmer sends the data needed for calculation to the GPU, as the device has a separate memory unit[1]. Secondly, the GPU is instructed to execute a written device function (kernel) with elected arguments. After scheduling the kernel the CPU process immediately returns, although the kernel calculation on the GPU is not finished yet. Thus a synchronization call can be inserted to pause the CPU process until the calculation has been finished. And lastly, the programmer copies the result data calculated by the GPU back to the host memory.

The latter three steps explained an exemplary behaviour of the host code. The device code is written in single instruction multiple data (SIMD)[7] fashion. This can be met by an execution of the code by a huge amount of threads[2]. The threads are aligned on a three dimensional grid in both programming models. To identify the the thread, which executes the code, it is common to save the ID of a thread in the beginning of a device code.

```
/////////////// CUDA ///////////////     ////////////// OpenCL ///////////////
int id = threadIdx.x
       + blockIdx.x * blockDim.x;      int id = get_global_id(0);
```

Each statement saves the thread ID of the first grid dimension. `threadIdx.x`, `blockIdx.x`, and `blockDim.x` can be seen as compiler macros, which will be translated into a function call by NVIDIA's compiler `nvcc`[19]. The threads are grouped in thread blocks. The number of threads per block and thread blocks is set in the host code kernel launch statement. Threads within the **same block** can synchronize with each other to ensure the validity of calculated results.

[1] In this thesis *host memory* will refer to the RAM of the CPU, whereas *device memory* will refer to the memory used by **one** GPU.

[2] If you want to know how the hardware is able to process a massive number of threads I refer to `https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture`

```
/////////////// CUDA ///////////////
doWork(devicePtr, id);
__syncthreads();
useWork(devicePtr, id);
```

```
////////////// OpenCL ///////////////
doWork(devicePtr, id);
barrier(CLK_LOCAL_MEM_FENCE);
useWork(devicePtr, id);
```

The example above shows that the synchronization between both function calls ensures that the data produced by `doWork` within a thread block is ready when `useWork` is called. If there was a synchronization between different thread blocks, the BSP model[25] would be broken, because it would not be possible to launch much more threads than there are processors due to possible dead locks.

Threads can be aligned on a linear grid, but also on a two or three dimensional grid, which is set in the host code when the kernel launch is scheduled. It is handy to have a multi dimensional grid when concerning logical multi dimensional problems.

To complete the example of a device code the following one firstly writes the array index to each thread position. Secondly, it executes a scan *inside* each block. For further information I refer to the CUDA programming guide [3] and the OpenCL reference pages[4]. The next section mentions shortly the additional required effort to program multiple GPUs within one node.

```
/////////////// CUDA ///////////////
scan_by_block(int* devicePtr) {

int id = threadIdx.x
       + blockIdx.x * blockDim.x;
int gid = blockIdx.x;
int gs = blockDim.x;
devicePtr[id] = id;
__syncthreads();
int acc = 0;
int i;
for (i = gs * gid; i <= id; ++i) {
  acc += devicePtr[i];
}
__syncthreads();
devicePtr[id] = acc;
}
```

```
////////////// OpenCL ///////////////
scan_by_block(__global int*
    devicePtr) {
int id = get_global_id(0);

int gid = get_group_id(0);
int gs = get_local_size(0);
devicePtr[id] = id;
barrier(CLK_LOCAL_MEM_FENCE);
int acc = 0;
int i;
for (i = gs * gid; i <= id; ++i) {
  acc += devicePtr[i];
}
barrier(CLK_LOCAL_MEM_FENCE);
devicePtr[id] = acc;
}
```

2.2. Programming Multiple GPUs

Whereas the last section exemplified the programming of one single GPU, this section gives an introduction to the key points, which necessary to programm multiple GPUs within one computing node.

Like the memory units of CPU and GPU are separated from each other, as it

[3] https://docs.nvidia.com/cuda/cuda-c-programming-guide
[4] https://www.khronos.org/registry/cl/sdk/2.1/docs/man/xhtml

was mentioned in the latter section, the memory units of multiple GPUs are also separated from each other, although they are located within the boundaries of one node. If you want to copy data from one GPU to the other, the data will be copied to the host first, and subsequently to the target GPU; or you must enable a peer to peer access between both participating GPUs first. But in general the functions mentioned above are still valid in a multi GPU context, though it has to be specified which GPU is the current one. This can be achieved with the `cudaSetDevice` function in CUDA Runtime API. All subsequent CUDA function calls will affect the device which has been specified in the last `cudaSetDevice` call.

```
cudaSetDevice(0);
... // function calls here affect the first device
cudaSetDevice(1);
... // function calls here affect the second device
cudaSetDevice(3);
... // function calls here affect the third device
```

Changing the device in CUDA Driver API is even more cumbersome, and not mentioned here. If you allocate a device pointer with a CUDA function, you do not have to save the corresponding device ID, because this is implicitly saved in the pointer value due to Unified Virtual Addressing (UVA), which is a CUDA feature, and available for devices with compute capability of 2.0 or greater.

Many applications, which require multiple kernel launches in a single GPU implementation, need to be manually parallelized to multiple GPUs with caution, as it is often the case, that the GPUs must communicate with each other between the kernel launches, to get the last recent data. In a manual parallelization errors can be found likely in the synchronization and communication between the host and devices.

Making the additional effort mentioned in this section superfluous by using the proposed workflow is the aim of the Mekong project. The following section will shortly mention other GPU programming models which does not affect the work presented in this thesis directly.

2.3. Other GPU Programming Models

With minor relevance for the work presented in this thesis there are other GPU programming models like OpenACC[2], OpenMP 4.0[1]. OpenACC & OpenMP 4.0 are basically libraries, which are used deploying compile pragmas within the host code. Thus the user can mark parallel environments inside the host code, wich will be translated into a parallel execution on the device at compile time of the program. OpenACC targets per default the execution on the device, whereas with OpenMP 4.0 this must be declared explicitly.

2.4. Modifying a Code

An important step to achieve an automatic parallelization at compile time is the modification of the original single GPU code (host & device side). For that purpose we use the LLVM compiler framework[4]. LLVM is well known for its modular architecture, and its analysis & optimization features.

In an abstract presentation the compilation of a code consists of three steps.

1. The front end, which checks the input code for correctness and translates it to an intermediate representation (LLVM IR).

2. The code optimization, which improves performance and reduces code size, e.g. with dead code elimination.

3. And the back end, which transforms the intermediate representation to the target language. In many cases this is an architecture dependent machine language.

The advantage of using LLVM is that we have not to care about step one and three, and can easily focus on analyzing and modifying IR in front of step two.

Thus we must know how to access and modify LLVM IR, and which rules are observed by IR. The essential model LLVM IR obeys is called *Static Single Assignment*[24] (SSA). An informal and short definition of SSA is represented by the following sentence:

> "A program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text."[24]

Basically this means that a regular written C code in LLVM IR looks like

```
/////////////// C Code /////////////     ;;;;;;;;;;;;; LLVM IR ;;;;;;;;;;;;;
int x;                                  %x = alloca i32
int y;                                  %y = alloca i32
int z;                                  %z = alloca i32
x = 1;                                  store i32 1, i32* %x
                                        %0 = load i32, i32* %x
                                        %add = add i32 %0
y = x + 1;                              store i32 %add, i32* %y
                                        %1 = load i32, i32* %z
y = z;                                  store i32 %1, i32* %y
                                        %2 = load i32, i32* %x
                                        %add1 = add i32 %2
z = x + 1;                              store i32 %add1, i32* %z
```

The code on the right side was automatically generated by the clang compiler. The assembly affinity of LLVM IR is highly visible. Variable names generally start with a per cent sign. The SSA form of LLVM IR is an advantage when analyzing the code, as you can be certain that a value of a variable is not changed due to any side effects.

Another example below presents a simple if-clause.

```
/////////////// C Code //////////////     ;;;;;;;;;;;;; LLVM IR ;;;;;;;;;;;;;
int i;                                   %i = alloca i32
i = 3;                                   store i32 3, i32* %i
                                         %0 = load i32, i32* %i
if (i > 4)                               %cmp = icmp sgt i32 %0, 4
                                         br i1 %cmp, label %then, label
                                             %else

{                                        then:
  i = 5;                                   store i32 5, i32* %i
}                                          br label %end

else {                                   else:
  i = 7;                                   store i32 7, i32* %i
                                           br label %end

}                                        end:
```

The sub grouping of instructions represents another element of LLVM IR, which is called a basic block. A basic block is a set of instructions, which are always executed sequentially. LLVM's branch instruction `br` decides based on boolean `%cmp` which basic block shall be executed next.

A function call in LLVM IR looks like:

```
/////////////// C Code //////////////     ;;;;;;;;;;;;; LLVM IR ;;;;;;;;;;;;;
foo();                                   %call = call i32 @foo()
bar();                                   %call1 = invoke i32 @bar()
```

Which we might have not expected. The `invoke` instruction declares a call of a function, which can throw an error, thus returning the control flow either to the normal or to the exception label. The example above shows that working with LLVM IR is rather technical and is not always predictable. Moreover the IR can have various shapes depending on the Clang/LLVM version.

Writing an LLVM pass enables us to analyze and modify the LVVM IR. We can access the IR objects with LLVM's IR API, which has been written in C++, and organized in a hierarchical class structure. Theoretically, you are able to change everything in IR, e.g. you can delete or change function calls; modify the order of instruction execution; or add additional arguments to a function declaration. But after all IR modifications have been done the IR is checked for a consistent state. An inconsistent state is e.g. a value usage before its definition, or anything which contradicts to the SSA model. If the IR fails the consistency test, the target code can not be generated.

In figure 2 you can see a code snippet of a pass, which replaces all calls of function `foo()` to calls of function `bar()`. Another detail which affects directly the work

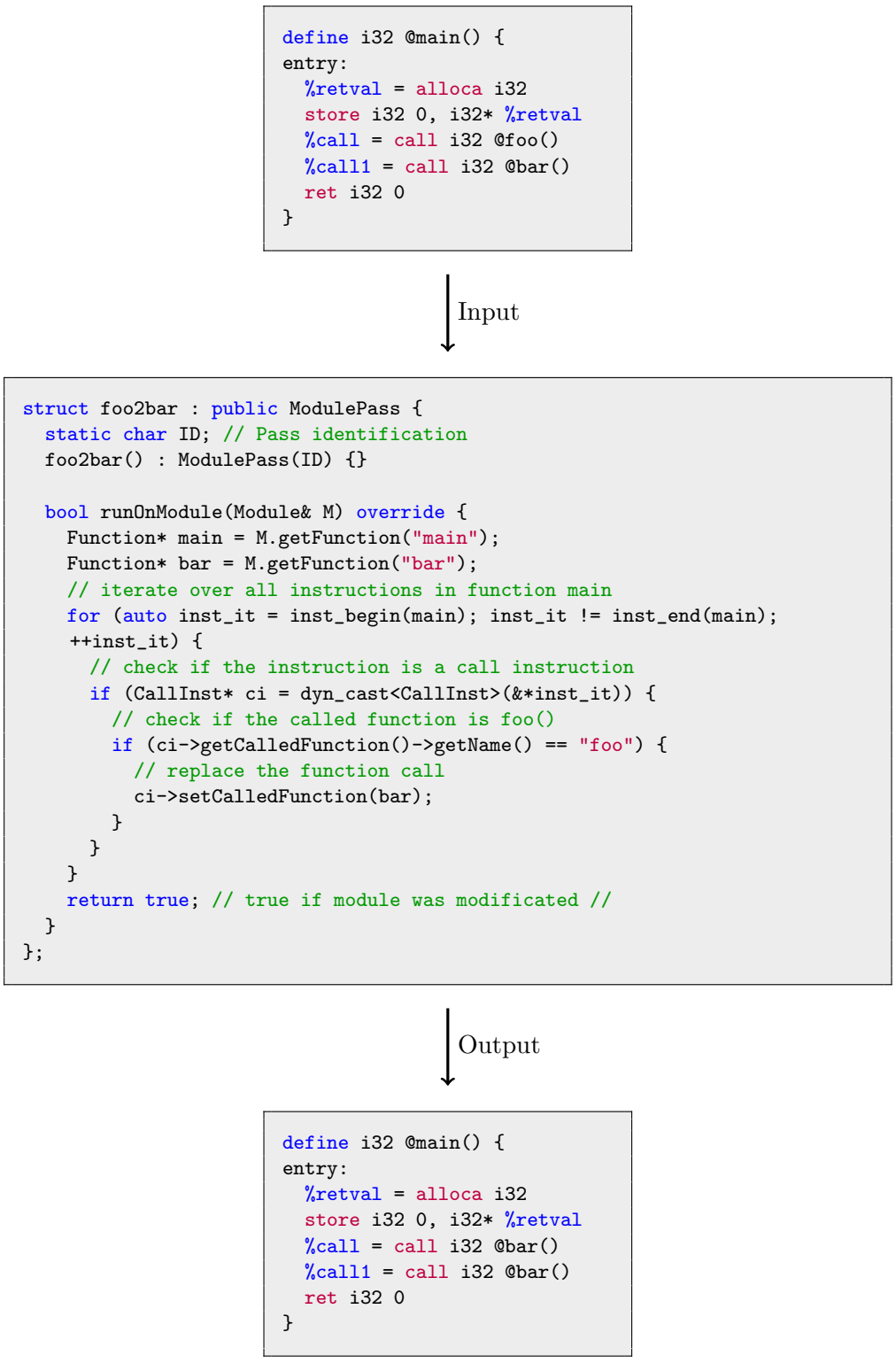


Figure 2.: Small code snippet to demonstrate how to change a called function in LLVM IR using a pass. The pass changes all calls of `foo()` to calls of `bar()`.

presented in this thesis, is the different transformation of OpenCL & C code into LLVM IR, which as an assembly like language must resemble assembly code generated for a specific target architecture. As OpenCL generally targets parallel architectures, its LLVM IR representation should embody that.

In pure SSA form there is one way to insert control flow into the code, which is also present in LLVM IR. With a phi-node it is possible to assign a value dependent on the previously executed block. Imagine we have two blocks of instructions called A and B, which both point to instruction block C. In the beginning of block C a phi-node instruction could be located: $x = \Phi(A : 1, B : 2)$, which assigns the value of one to x if block A has been executed previously, or sets x to a value of two if B was the preceding block.

Moreover in LLVM IR we can insert control flow deploying registers. E.g. there is a pointer p, which points to a certain memory location. That memory location can be written with different values in different blocks, without contradicting to the SSA model. Block A can assign value 1, whereas block B can assign a value of 2, with the usage of a store instruction. The same memory location can be read in successive block C, and is either 1 or 2, which depends on the previous executed block.

2.5. Related Work

In the recent years a couple of publications regarding the partitioning of single device applications written in SIMD fashion were released. The most relevant publications, which propose similar techniques like they are presented in this thesis, were written by Lee et al., who put out a runtime framework mapping kernels to multiple devices (MKMD)[16]. MKMD targets GPUs and CPUs for its kernel distribution, considers inter kernel dependencies, and is compatible with OpenCL APIs. Another approach by Lee et al. which also includes the kernel modifications at compile time, introduces the single-kernel multiple device (SKMD)[15] system, which simulates one large virtual device to the programmer, but orchestrates launched kernels to heterogeneous compute units. The striking difference to Mekong's approach is how both projects mentioned above deal with inter kernel dependencies, and how they get the distributed output elements back to the host. Lee et al. states the usage of a merge kernel, which is basically a modified version of the original result producing kernel. The merge kernel is run on one compute unit and collects the results made by the other units, which must cause communication via an universal address space between the computing units. Mekong's runtime library addresses this problem with cached calculations based on the memory access patterns, which are gathered in Mekong's analysis part.

Pandit & Govindarajan also deploys a merge kernel in their proposed intermediate layer Fluidic[21], which is located between the programmer and the OpenCL API. Fluidic has been written in C, makes usage of Pthreads and OpenCL, and is appropriate for a work partitioning between CPU and one GPU within one node.

Schaetz & Uecker propose a multi GPU C++ programming library[23] for deploying multiple GPUs in a object oriented fashion. The library style resembles the well known MPI communication model, thus it is mandatory for the programmer to use an API. The library was implemented onto a CUDA backend, which requires the

CUDA Driver and CUDA capable GPUs for execution.

Kim et al. presents an OpenCL framework[13], which simulates a one device environment to the programmer, but deploys underneath various available GPUs. When the partitioned results of an applcation has been calculated, the framework can copy them correctly back to the host, because a sample run was scheduled previously to determine the written elements of a kernel. This requires the kernel memory accesses to be affine. The difference to Mekong's approach is the sample run, which determines the written indices at runtime, whereas Mekong's runtime library knows them in advance, due to the static kernel analysis.

Ravinshankar et al. proposes a formalism[22] regarding the generation of code for distributed memory systems, which is capable of a mixture between regualar and irregular memory accesses within different parts of a program.

Kaestle et al. implemented a runtime library Shoal[12] for parallel programs on NUMA machines. The library provides an array data type, which contains additional logic and information about memory access patterns, which can be treated particularly by compilation techniques.

Cilardo et al. outline an automated memory partitioning scheme[6] at compile time, which partitions while minimizing structural conflicts on memory ports. The approach relies on the Z-polyhedral model for static program analysis and is able to reorder and reindex affin memory accesses within nested for-loops. In fact their work represents a compiler optimization technique at its roots.

3. Project Mekong

This chapter provides information about the the high level overview of the project; how Mekong utilizes LLVM passes for its analysis and transformation part; the usage of the analysis results in the runtime part, and lastly, how you can use the software yourself.

3.1. High Level Overview

In figure 3 you can see Mekong's subsequent processing steps while parallelizing a code. We start with the transformation to LLVM IR by utilizing the `clang`[3] compiler. Clang is well known as a compiler frontend for C, C++, and Objective-C, but is also able to compile OpenCL code to LLVM IR. As the compilation of CUDA code with Clang is still in an active development state and NVIDIA does not grant access to the internal representation of their CUDA compiler `nvcc`, we choose to have the device code in OpenCL.

The next step is the code analysis, which is done with usage of custom and LVVM passes; and the Polly library[5] to recognize memory access patterns. The analysis results are written into a database. In consequence we are able to make the partitioning decision, which is written into the database too.

Afterwards we modify the code to enable the execution of the device code on multiple GPUs. This step includes e.g. adding kernel arguments and replacing function calls in the host code.

In the last step amongst other things we link the code with our runtime library, which contains the functionality of the substituted functions. The result is a device code in PTX[20] format and a host code in machine format, which instructs the GPU to load the PTX code at application runtime with a just-in-time compilation, thus the input host code must be written in CUDA driver API. The results produced by the partitioned executable should be correct in the sense that it delivers one GPU results within floating point rounding errors.

When we partition a kernel to multiple GPUs, we can cause a communication between the GPUs, because one GPU might produce results the other one must read. Thus an update of the invalid data is necessary. This results in memory copy operations between GPUs, which we call *communication*. Mekong should partition an application while minimizing the communication between GPUs and maximize productivity, thus minimizing user interaction.

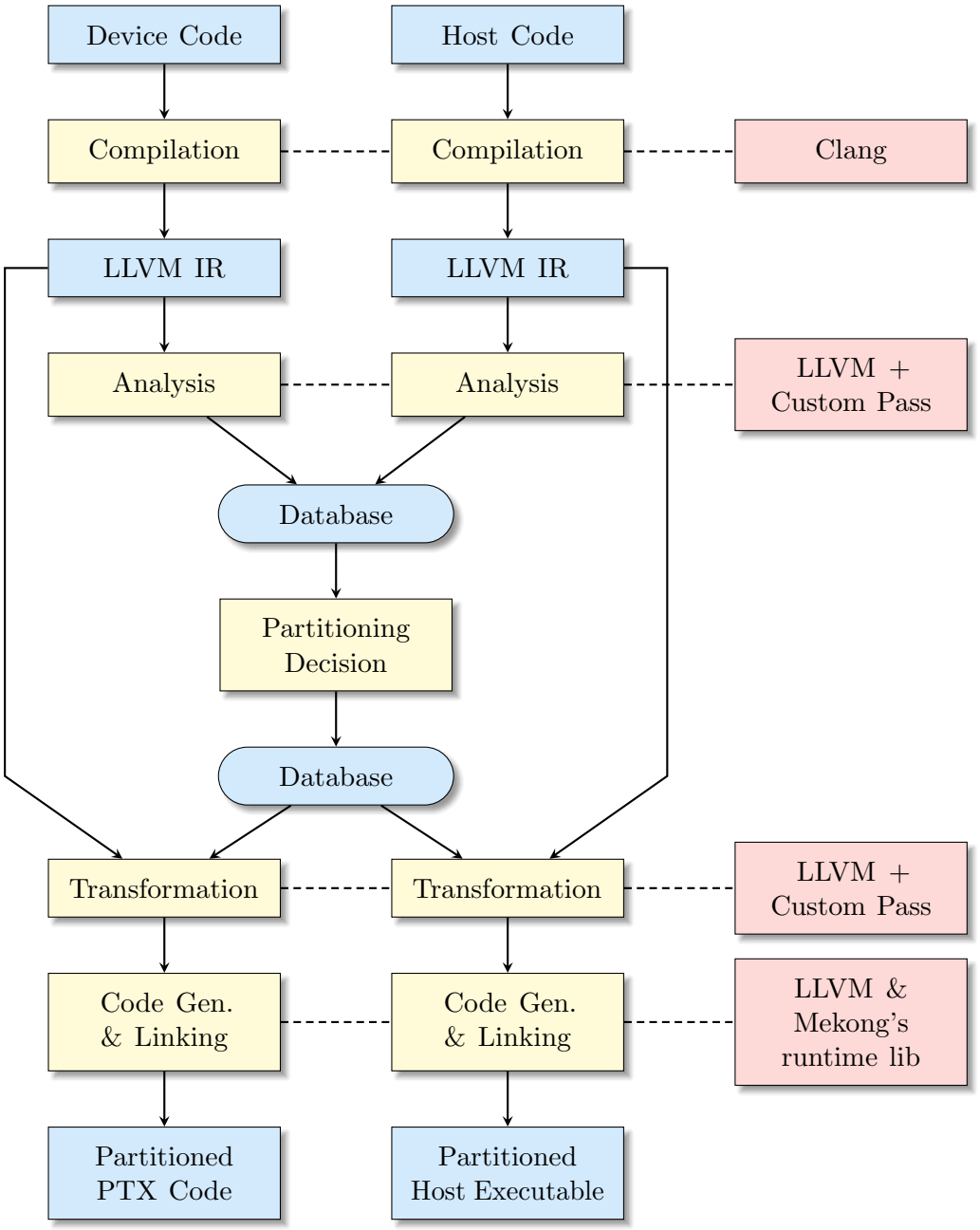


Figure 3.: Mekong's processing toolchain.

3.2. Mekong's Analysis Part

As you can see in figure 3 the first step of Mekong's toolchain is the code analysis part, which collects static available information and writes it into a database. For subsequent processing steps, e.g. the available kernels and their argument types, iterativity of kernel launches in the host code, kernel regularity, and memory access patterns are a useful information. Gathering this information is achieved by deploying custom and LLVM passes; and the Polly library to get the memory access patterns (see section 3.2.1 for memory access pattern representation). You can see a complete presentation of the information which is gathered in table 1.

Generally the host code analysis is independent of the device code analyis, but as no reasonable information of the host code is used in subsequent processing steps up to now, the host code analysis pass is not implemented yet.

As the GPU programming model contains implicitly the device code execution on a three dimensional grid, we have to embody the grid by wrapping the kernel body with three for-loops in order to make the analysis of the memory access patterns possible. This modification is temporary and will be discarded after the analysis is done. This is achieved using a custom pass wrapping three for-loops in LLVM IR representation of the device code around every kernel body. You can think of the changes the loop wrapping pass induces to be similar to the code snippet below:

```
// Before the loop wrapping pass
void kernel (...) {



  ... // kernel body



}
```

```
// After the loop wrapping pass
void kernel (..., int size_x,
                  int size_y,
                  int size_z) {
for (int x=0; x<size_x; ++x) {
for (int y=0; y<size_y; ++y) {
for (int z=0; z<size_z; ++z) {
  ... // kernel body
} // x loop
} // y loop
} // z loop
} // kernel end
```

You saw in section 2.4 that there are two different methods how control flow can be represented in LLVM IR: (1) using phi-nodes, (2) using registers. Up to now the loop wrapping pass supports the latter mentioned point only.

If Polly succeeds in its analysis the user *only* has to decide which grid dimension(s) shall be split. For execution Mekong's runtime part will divide the thread blocks along that dimension(s) to different GPUs (see section 3.4). In future versions of Mekong that dimension should be determined by searching for the highest stride causing dimension for all memory accesses inside a kernel. E.g. for `devicePtr[idx + N * idy]` the y-dimension causes a higher stride than the x-dimension. To decrease communication between the GPUs the kernel should be split along the y-dimension.

Once all information is collected, it is written into a simple self written database called `dashdb`. As the database format is similar to the JSON[11] format, I refer to

Key	**Explanation**
kernel names	names of the available kernels in the device code
kernel partitioning	possible values: 'x', 'y', 'z', 'xy', 'yz', ...
argument name	kernel argument name
arg. type name	LLVM IR argument type names, e.g. 'i32', 'float*'
arg. size	size of argument in bits; zero for pointer types
arg. element size	size in bits of pointed type; zero for non-pointer types
arg. fundamental type	pointed type or own type, e.g. 'f', 'd', 'i' for float, double, or integer
arg. dim size	number of dimensions of the `kernel_array` represented by a pointer type; zero for non-pointer types
arg. pointer level	`int*` has ptr lvl 1; `float` has ptr lvl 0; `double**` has ptr lvl 2;
arg. read map	ISL map which describes the read behaviour on that argument (empty for non-pointer types).
arg. write map	ISL map which describes the write behaviour on that argument (empty for non-pointer types).
arg. read parameter	pattern to describe the dependency of the read map on scalar kernel arguments.
arg. write parameter	pattern to describe the dependency of the write map on scalar kernel arguments.

Table 1.: List of gathered information in Mekong's analysis part up to now.

the repository[1] visible on my github account for further reading about the database.

3.2.1. Memory Access Patterns with ISL & Polly

An elementary part of Mekong's analysis is the extraction of memory access patterns located in the device code. As we deploy the Polly library for that purpose and Polly uses the Integer Set Library (ISL)[26] for its internal representation of memory access patterns, *our* usage of Polly is tightly coupled to the ISL.

The ISL is a "C library for manipulating sets and relations of integer points bounded by affine constraints"[27]. The library provides common mathematical operations on sets and mappings living in $\mathbb{Z}^n$. The "affine constraints" mentioned in the citation above mean amongst others that it is *not* allowed to define a set like

```
{ [x, y] : y <= x * x},
```

but all sets describing an n-dimensional polyhedron are valid. Polly in general is an optimization library using polyhedral techniques to restructure memory accesses in nested for-loops. Moreover it works on LLVM IR, thus can be invoked deploying the Clang compiler. Before optimizing Polly must analyze the code, which is used my Mekong's analysis pass, whereas Polly's optimization pass is not used at all.

For example if a code looks like

```
void foo(int* in, int* out, int N) {
  for (int x = 0; x < 1024; ++x) {
    for (int y = N; y < 2048; ++y) {
      out[x + y * 1024] = in[x + N];
    }
  }
}
```

Polly can extract the domain of both nested for-loops, which looks like

```
[N] -> { [x, y] : 0 <= x < 1024 and N <= y < 2048 },
```

where `N` is a space parameter in ISL representation. To our advantage Polly is able to extract memory access patterns in dependency of unknown variable values at compile time, as far as the constraints are affine. It is *not* possible to define a set like

```
[N] -> { [x, y] : y <= N * N },
```

because the constraint is not affine. Moreover the extracted memory read accesses of the code examples above looks like

```
[N] -> { [x, y] -> in[x + N] : 0 <= x < 1024 and N <= y < 2048 },
```

whereas the write access has a shape of

```
[N] -> { [x, y] -> out[x, y] : 0 <= x < 1024 and N <= y < 2048 },
```

[1] `https://github.com/codecircuit/dashdb`

which embodies the logical two dimensionality of the array `out`. The assumption of the two dimensional access is true as long as variable `x` is smaller than the factor variable `y` is multiplied with. In our example this constraint is met by the limits imposed by the `x`-for-loop. The delinearization of memory accesses provides additional information about the programmer's logical intention, rather than about his technical realization only. Up to now the delinearization of accesses is not used for any improvement, but there is the auspicious potential, which can be exploited in future versions of Mekong.

Mekong's analysis part wraps the kernel body in three for-loops, which simulate the execution grid, and uses Polly to extract the memory access patterns of *every* domain inside the kernel. Afterwards for every pointer kernel argument[2] all read and write accesses are united into an `isl_union_map` object, which is then written into the database. You can imagine the resulting map as a function of scalar kernel arguments and kernel launch parameters (e.g. super global thread ID). This function is able to return a set of accessed indices on a `kernel_array`.

3.2.2. Discussion on Polly Dependency

Without the knowledge of the memory write accesses of a kernel our workflow would not know which array indices are written by a certain thread. Thus by partitioning the super grid to multiple GPUs, it would not be possible to know which GPU writes a certain output element. In the end it could be impossible to copy the correct data back to the host.

As the analysis of memory access patterns in LLVM IR is a rather complex subject, we decided to rely on an external library covering that scope. Polly as a well known library in their field of activity is in a state of active development, and was therefore chosen. Moreover Polly's modular architecture provides us an easy access to our wanted information only.

That means if Polly fails[3] in its analysis, Mekong's whole workflow will fail too. But as Polly is rather good in detecting memory access patterns we decided to rely on that dependency.

3.3. Mekong's Transformation Part

Mekong's host code transformation basically consists of CUDA relevant function call replacements. Thus we firstly insert the definitions of our wrapping functions, which are defined in our runtime library, and secondly, substitute the appropriate CUDA functions. The signature of a wrapping function is equal to its belonging CUDA function. Otherwise we had to transform LLVM IR types, which is error-prone and not possible if a value of changed type is used in an external library, which we can not modify.

Moreover to provide a report of Mekong's runtime statistics, a call instruction is inserted in front of every return statement. Up to now only the main function of the

[2] Write or read accesses to shared or local memory are not relevant for our workflow, because the block borders are kept while partitioning an execution grid.

[3] In that case it might be possible to make Polly's analysis successful by simplifying the device code in LLVM IR (see section 5.1

CUDA Function	Mekong's Wrapping Function
cuInit	wrapInit
cuDeviceGetCount	wrapDeviceGetCount
cuDeviceGet	wrapDeviceGet
cuDeviceComputeCapability	wrapDeviceComputeCapability
cuCtxCreate_v2	wrapCtxCreate
cuModuleLoad	wrapModuleLoad
cuModuleGetFunction	wrapModuleGetFunction
cuMemAlloc_v2	wrapMemAlloc
cuMemcpyHtoD_v2	wrapMemcpyHtoD
cuLaunchKernel	wrapLaunchKernel
cuCtxSynchronize	wrapCtxSynchronize
cuMemcpyDtoH_v2	wrapMemcpyDtoH
cuMemFree_v2	wrapMemFree
cuCtxDestroy_v2	wrapCtxDestroy

Table 2.: List of replaced functions in the host code.

input code is affected by host code transformations. You can see a complete list of the functions, which are substituted by our workflow in table 2.

The main reason for transforming the device code is the partitioning of the original grid to different GPUs. Although a block is executed on a smaller grid on one GPU, it has to know where it belongs on the original grid, as it is common to access memory locations in dependency of the thread or block ID (see section 2.1). From now on the original grid, which is scheduled in the unmodified input code, will be called super grid. And a piece of the partitioned super gridwill be called sub grid.

The device code transformation is sub divided into three steps:

1. **Promoting:** every kernel and device function, which calls any OpenCL global ID or size function (`get_global_id`, `get_global_size`), will be cloned with six additional arguments: `offset_[x|y|z]` and `superGridSize_[x|y|z]`. The offset points to the first thread of a GPU in coordinates of the super grid, whereas the additional size contains the size of the super grid in every dimension counted in threads. If we had a one dimensional grid and two GPUs for partitioning with two blocks each & three threads per block, the offset would be:

	GPU0						GPU1					
super global thread ID	0	1	2	3	4	5	6	7	8	9	10	11
sub global thread ID	0	1	2	3	4	5	0	1	2	3	4	5
offset	0			0			6			6		
super block ID	0			1			2			3		

You can think of the changes happening in LLVM IR after this step to be like the code snippet below:

```
// BEFORE PROMOTING          // AFTER PROMOTING
void kernel(...) {           void kernel (..., int64_t offset_x,
                                               int64_t offset_y,
                                               int64_t offset_z,
                                               int64_t superGridSize_x,
                                               int64_t superGridSize_y,
                                               int64_t superGridSize_z)
                             {
... // kernel body             ... // kernel body
}                            }
```

2. **Propagating:** a promoted kernel or device function possibly contains call instructions of other device functions. If there is a promoted version of the called function, we must change the call to the promoted version in order to ensure a correct partitioned kernel execution.

3. **Offsetting:** in the last step, we must modify the values of the `get_global_id` or `get_global_size` calls, which are inside a promoted kernel or device function. Up to now we change calls only if they affect the dimension, which shall be split. Thus changing the splitting dimension at program runtime is not possible[4]. For modification we use the additional kernel arguments described in step one. You can think of the modifications happening in LLVM IR after this step to be similar to the code snippet below:

```
// BEFORE OFFSETTING          // AFTER OFFSETTING
int id = get_global_id(0);    int id = get_global_id(0) +
int s =                           offset_x;
    get_global_size(0);       int s = superGridSize_x;
```

3.4. Mekong's Runtime Part

Mekong's runtime library handles all tasks, which *must* be processed at application runtime. This is mandatory, because some tasks require data, which is not available at compile time (e.g. command line parameters). Typically the first tasks of the runtime are the various CUDA Driver API context, module, kernel, device, and device memory pointer handling. As you can read in section 3.3, our host code transformation pass does not modify types in the input application. Nevertheless from the programmer's sight we simulate a one GPU environment, thus he has to define one CUDA Driver API context, module and one device. Moreover if he loads a

[4]But that is intended, because Mekong's primary design is the maximization of tasks at compile time.

kernel function or allocates a device memory pointer, he does it for a single device. In case of a device memory allocation it seems to him that the `cuMemAlloc` function returns one single pointer.

What actually happens is that our runtime allocates a device memory buffer on every available GPU, writes them into a lookup map withe the device buffer allocated on the first GPU as a key, and returns the key as the result of the wrapping function. This mapping behaviour is analog for the CUDA Driver API context, module, device and kernel management.

A static alternative to this dynamic mapping method is a type casting at compile time, which we have tried in earlier versions of Mekong. The approach included the transformation of every used device memory pointer (`CUdeviceptr`) into a list of device pointers with one entry for each available GPU (`CUdeviceptr*`). One insuperable problem that arises with a static type casting approach is the usage of a device pointer object in an external library. E.g. the usage of a device pointer object in a function call. When we changed the type of the device pointer the function call was not valid anymore, because we could not change the function definitions, as it was contained in an external library. A single example representing the latter situation arises with the usage of the `swap` contained in the C++ standard library to swap two device memory pointers.

(1) Buffer allocation, (2) data initialization, (3) memory copy operation to device, (4) kernel launch, and (5) memory copy operation to host represents a common execution order for a host code, which instructs a GPU. As the data initialization generally happens without using any CUDA functions, the next active task for our runtime is the handling of a *memory copy operation to device*. Up to now every allocated device buffer in the host code will be allocated on *every* device. Thus it is possible to handle every CUDA host to device copy operation as a broadcast from host to *all* devices. This results in a data replication, which is strongly noticeable when investigating the performance of host to device memory copy sensitive applications[5] (see section 4.2 for performance results of such an application).

When compiling the runtime library it is possible to choose between a trivial implemented host to device broadcast or using a library called Sofire for that purpose written by Dominik Sterk, who is also a member of our working group. Sofire is still in an active development state, thus any substantial performance gain could not be recognized yet.

The next wrapped function belongs to CUDA's kernel launch function and contains the highest complexity of Mekong's runtime library. A call possibly causes the actual grid partitioning, a dependency resolution between kernel launches, and the kernel execution. Subsequently after the calculations are done on the device, an application usually schedules a device to host memory copy operation. The runtime library must know which kernel launch has written to the required device buffer last and must calculate with knowledge of the memory write access patterns which GPU holds valid output elements to copy them back to the host. Thus the amount of data copied back is equal to a one GPU environment.

The following sections explain the functionality behind the kernel launch wrapping

[5] See section 5.2 to read about how a broadcast can be avoided in some special cases.

function and present an overview of the implementation.

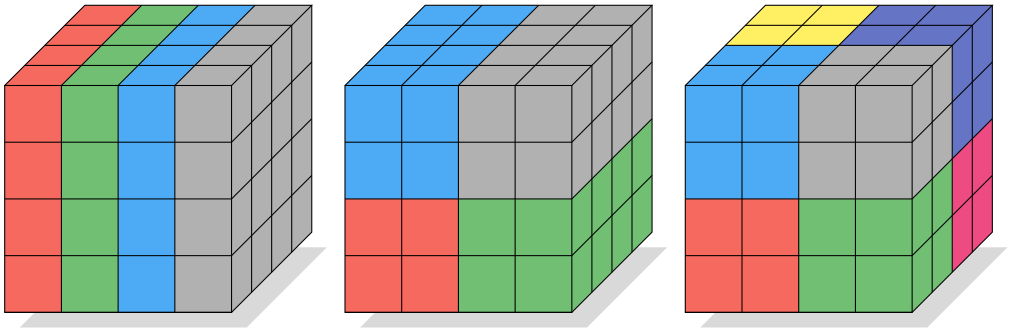


Figure 4.: Partitioning examples for various dimensions. A small cube represents one thread block, whereas a large block illustrates the whole execution grid. A color indicates the execution of a block by a specific GPU.

3.4.1. Partitioning Strategy

When a kernel launch is scheduled the runtime has to decide how the super grid is partitioned to the available GPUs. This *must* be done at runtime, as we do not know about the kernel launch configuration (grid size, block size, etc.) at compile time, because it is common practice to code the kernel launch configuration to be dependent on command line parameters. Mekong's analysis determined already which dimension(s) of the super grid should be split, thus at the moment of kernel launch we have to distribute the blocks to different GPUs. You can see example partitionings of various dimensionality in figure 4. Up to now our code supports one and two dimensional partitioning schemes. The written algorithm distributes as fair as possible, and splits the super grid into at least `numGPUs` partitions. For each partition actually one kernel launch will be scheduled. Moreover a partition must know where it belongs on the super grid, thus it needs a size and offset in three dimensional space. We choose to have full flexible partitions, as it might be useful to have multiple partitions per GPU if we want to synchronize only with a certain part of the super grid or to prioritize specific partitions. This is beneficial for overlapping communication and calculation in general. For example it might be possible to finish the calculation of blocks, which produce data needed by subsequent kernel launches, first. Then while calculating the other blocks mandatory memory copy operations might already be executed. As soon as all mandatory memory copies are finished the subsequent kernel can already be scheduled. Our partitioning algorithm for one dimensional partitioning schemes, distributes the work as fair as possible to all available GPUs. You can see examples for its behaviour in figure 5. Basically the main task of the 1D splitting algorithm is the calculation of the partition sizes and their offsets on the super grid. These values are needed when a kernel is launched, which executes the partition. The device code transformation (see section 3.3) added additional kernel arguments, thus the offset is known inside the kernel body as a kernel function argument and the size

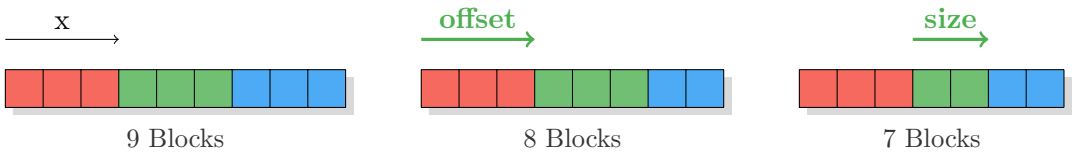


Figure 5.: Example 1D partitionings for three GPUs and a different number of blocks. For the green partition you can see the offset on the super grid and its size.

of a partition is known by the kernel launch configuration.

You can see a code snippet for the two dimensional partitioning in figure 7, which works as it follows:

1. E.g. if the analysis fixes the x- and y-dimension as the partitioning dimensions for the super grid, the algorithm checks which dimension is smaller in number of blocks. That dimension size is assigned to variable `smallDimSize`, and the other one appropriately to `largeDimSize`.

2. Then it tries to partition like a human would intuitively do. If one saw a rectangle, which he had to divide into `numGPUs` parts, one would search for a factor breakdown, e.g. `i * ii = numGPUs`. Subsequently the `small_factor` is taken, and one would try to cut the rectangle along its smaller side into `small_factor` parts. This can only be done if the constraints of the super grid do not contradict to that, as we can only cut the super grid along block borders. Consecutively, the same procedure is applied to the large grid dimension.

3. Lastly, we have to calculate the offset vector, which indicates the position of a partition on the super grid.

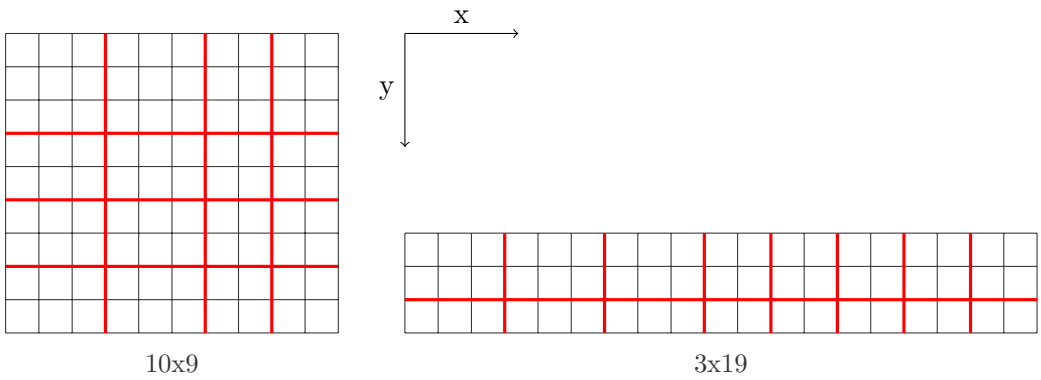


Figure 6.: Exemplary 2D partitionings for 16 GPUs. A small square represents one thread block. The red lines indicate where the super grid is cut into partitions. The invisible grid dimension has at least a size of one block.

Moreover you can see in figure 6 two exemplary cases of how our splitting algorithm works with two dimensional partitioning schemes.

A three dimensional partitioning scheme is not supported yet. I have not implemented an algorithm for that case, because the necessity due to a test case which requires or benefits from a three dimensional partitioning scheme was not given yet. Moreover Polly's support for codes dealing with logical three dimensional problems seems also not to be existent. Various attempts to analyze an as simple as possible written three dimensional stencil kernel failed. Moreover a benchmark should be done to estimate if an application can benefit from a two dimensional partitioning in comparison to a one dimensional one. The problem which might arise with a multi dimensional partitioning scheme could be that it possibly creates non continuous memory copy operations if there are inter kernel dependencies causing GPU communication.

3.4.2. islpy Example

This chapter will exemplary go through all ISL calculations, which are done in our workflow. The starting point for that is the ISL output of Polly's memory access pattern analysis. For reasons of readability and comprehensibility the examples are written in python using a module called islpy[14], which is simply a wrapper for the originally in C written ISL. You can skip to the next section if you want to proceed with an abstract view on the inter kernel dependency resolution.

Imagine we have a kernel code like

```
__kernel void stencil(__global float* in, __global float* out, int N) {
  idx = get_global_id(0);
  idy = get_global_id(1);
  if (idx > 0 && idy > 0 && idx < N - 1 && idy < N - 1) {
    int id = idx + idy * N;
    float acc = 0;
    // DOMAIN 0
    for (int i = 0; i < 3; ++i) {
      acc += in[id - 1 + i];          // access A
    }
    // DOMAIN 1
    acc += in[id - N];                // access B
    acc += in[id + N];                // access C
    out[id] = acc;                    // access D
  }
}
```

which is a 2D stencil kernel reading the top, bottom, left and right neighbour cells and adding their cell value to the own one. Now one can deploy the ISL to represent the memory access patterns of the kernel. We have to keep in mind that access A is in a different domain than access B, C & D. To start with, all accesses are in the domain of the implicitly executed grid, but moreover access A is located in another for-loop, which represents another iteration variable. Thus with the three dimensional execution grid and one for-loop access A lives in a four dimensional iteration space, whereas the other accesses reside only in a three dimensinal one. A possible output

```
// Get super grid sizes in number of blocks
smallDimSize = getSmallDimSize(partitioningDims);
largeDimSize = getLargeDimSize(partitioningDims);
if (smallDimSize < 2) { throw; } // no 2D partitioning possible
for (int i = numDevice/2; i > 1; --i) {
  if (numDevice % i != 0) { continue; }
  int ii = numDevice / i; // now ii * i = numDevice
  int small_factor = min(ii, i);
  int large_factor = max(ii, i);
  // If the small factor is too large, search for a smaller one
  if (small_factor > smallGridSize) { continue; }
  // Calculate how many work is on small split dimension
  vector<Array3> work(numDevice, superGridSizes);
  for (unsigned gpu = 0; gpu < numDevice; ++gpu) {
    work[gpu][smallDim] = superGridSizes[smallDim] / small_factor;
  }
  // distribute the rest along the gpus
  for (unsigned rest = 0; rest < (superGridSizes[smallDim] % small_factor); ++rest)
    {
    ++work[rest % numDevice][smallDim];
  }
  // Calculate how many work is on split dimension big;
  // analog to 1D splitting for big_i gpus
  for (unsigned gpu = 0; gpu < numDevice; ++gpu) {
    work[gpu][largeDim] = superGridSizes[largeDim] / large_factor;
  }
  // distribute the rest along the gpus
  for (unsigned rest = 0; rest < superGridSizes[largeDim] % large_factor; ++rest) {
    ++work[rest % numDevice][largeDim];
  }
  // Calculate the offsets
  vector<Array3> offset(numDevice, {0, 0, 0});
  for (int i = 0; i < 2; ++i) {
    int currSplitDim = splitDims[i];
    unsigned count = work[0][currSplitDim];
    for (unsigned gpu = 1; gpu < numDevice; ++gpu) {
      offset[gpu][currSplitDim] = count * blockSizes[currSplitDim];
      count += work[gpu][currSplitDim];
    }
  }
  for (unsigned gpu = 0; gpu < numDevice; ++gpu) {
    Partition p(work[gpu], offset[gpu], gpu));
    res.push_back(p);
  }
  return res;
}
```

Figure 7.: Code snippet creating the two dimensional partitioning of a super grid to enable execution on multiple GPUs.

of Polly's analysis could look in ISL representation like

```
// domain0
[size_x, size_y, size_z, N] -> {
  domain0[idx, idy, idz, i] : 0 < idx < N - 1 and 0 < idy < N - 1 and
                              idx < size_x and idy < size_y and
                              0 <= idz < size_z and 0 <= i < 3
};
// acc_A
[size_x, size_y, size_z, N] -> {
  domain0[idx, idy, idz, i] -> in[idy, idx - 1 + i]
};
// domain1
[size_x, size_y, size_z, N] -> {
  domain1[idx, idy, idz] : 0 < idx < N - 1 and 0 < idy < N - 1 and
                           idx < size_x and idy < size_y and
                           0 <= idz < size_z
};
// acc_B
[size_x, size_y, size_z, N] -> {
  domain1[idx, idy, idz] -> in[idy + 1, idx]
};
// acc_C
[size_x, size_y, size_z, N] -> {
  domain1[idx, idy, idz] -> in[idy - 1, idx]
};
```

It is noticeable that there are separate objects for accesses and the domain space, where the accesses live in. The domains are limited by the three for-loops, which had been wrapped around the kernel to simulate the execution grid, before Polly's analysis was started. The range of the three for-loops starts with zero and is limited by `size_[x|y|z]`, thus the domains are limited by these variables too. As their values (grid and block size) are not known at compile time they are space parameters, which can be set to a specific value in later calculation stages. Furthermore `domain0` is limited by the limits of the for-loop over `i`. The access maps indirectly contain the limits of the domains by referring to their name `domain0` and `domain1`.

Polly assigns the first dimension of a two dimensional mapping to the unbounded value. That means in our case `idy`, because if `idx >= N`, the access `in[idx+idy*N]` would not be a logical consistent two dimensional access anymore. Thus the value `idx` is bounded and assigned to the second dimension.

In the following will be explained what Mekong's analysis part must do to appropriately prepare the access patterns for Mekong's runtime part. Subsequently the usage in the runtime will be outlined.

ISL Operations in Mekong's Analysis

The analysis has to handle Polly's output and must compress it before writing it into the database. This should be done, as we want to minimize the work, which must

be done at runtime. You saw in the ISL access patterns above that the accesses and domains are split. Thus the first step is the union of all accesses for each domain. In the python snippet below `acc_[A|B|C]` is of type `islpy.Map`, and `domain[0|1]` is of type `islpy.Set`.

```
# domain0 contains only one access; no union necessary
acc_B_C = acc_B.union(acc_C)
```

The second step is the intersection of these accesses with their belonging domain to reduce the necessity of an additional object.

```
acc_domain0 =   acc_A.intersect_domain(domain0)
acc_domain1 = acc_B_C.intersect_domain(domain1)
```

And lastly, all accesses on the same array can be united in one `isl_union_map` object.

```
## type cast
acc_domain0 = islpy.UnionMap.from_map(acc_domain0)
acc_domain1 = islpy.UnionMap.from_map(acc_domain1)
## union
all_acc = acc_domain1.union(acc_domain0)
```

The print of `all_acc` is rather long and is not shown here. Essential is that we have compressed the information about all read accesses on array `in` into one object. You can imagine `all_acc` to be a function of kernel launch arguments (like `N`) and the configuration (like grid size, block size, etc.). This function is able to return the accessed points of a specific kernel launch on the array `in`. With knowledge of the sizes of array `in` the function is also able to return the accessed indices by delinearizing the two dimensional points.

ISL Operations in Mekong's Runtime

The runtime library provides the creation of partitions, which represents the execution of a part of the super grid. Each partition is executed on one device, thus with knowledge of the written indices by a partition it is known which GPU wrote that indices and has the data in its memory. It is important to know about that, because in case of a memory copy operation from device to host scheduled in the original application code, we must know which GPU holds valid output elements to get them back to the host correctly. Therefore we will exemplary show how to calculate the accessed indices of a specific partition using the access pattern given by the Analysis.

Firstly, we must fix the ISL space parameters with the appropriate kernel launch arguments (like `N`) and configuraion (grid size, block size, etc.).

```
# Here we take an ISL space object to set the space parameters
param_set = all_acc.params()
# We assume a grid of 2x2x1 blocks with a size of 16x16x1 threads each
size_x = 2*16
size_y = 2*16
size_z = 1
N = 2*16 # per definition the array width
param_vals = [size_x, size_y, size_z, N]

for i,v in list(enumerate(param_vals)):
    # type cast
    isl_signed_int = islpy.Val.int_from_si(param_set.get_ctx(), v)
    # fix space parameter
    param_set = param_set.fix_val(1, i, isl_signed_int)

# Fix space parameters in our isl_union_map
launch_acc = all_acc.intersect_params(param_set)
```

To achieve that we must get the ISL space object first, which contains e.g. information about space parameters and number of dimensions. Subsequently, we can cast our numerical value, which we want to fix, into an `isl_value` to set it in the `isl_space` object. Then the `isl_space` of the given access map must be updated with our new fixed version. After that `launch_acc` contains information about accessed points of a specific kernel launch.

Secondly, if we are interested in the accessed points of a specific partition, we must set the boundaries of the partition in the ISL map. To continue the example we assume that we have two GPUs and two partitions – one for each GPU. We set the size and offset of a partition manually; and split the super grid at the y-dimension.

```
#               offset  ,    size    , gpu_id
p0 = Partition([0, 0, 0] , [32, 16, 1], 0)  # GPU0
p1 = Partition([0, 16, 0], [32, 16, 1], 1)  # GPU1
```

To proceed further we will introduce a help function, which applies the limits of a partition to an `isl_map`.

```
def setPartitionBoundaries(islmap, partition):
  space = islmap.get_space()
  ls = islpy.LocalSpace.from_space(space)
  ctx = islmap.get_ctx()
  for i in range(3): # for every grid dimension
    u = islpy.Constraint.alloc_inequality(ls) # upper
    l = islpy.Constraint.alloc_inequality(ls) # lower
    # set values of lower limit
    vl = islpy.Val.int_from_si(ctx, -partition.offset[i])
    l = l.set_constant_val(vl)
    vl = islpy.Val.int_from_si(ctx, 1)
    l = l.set_coefficient_val(2, i, vl) # 2 -> isl_dim_in
    # set values of upper limit
    vu =
    islpy.Val.int_from_ui(ctx,partition.offset[i]+partition.size[i]-1)
    u = u.set_constant_val(vu)
    vu = islpy.Val.int_from_si(ctx, -1)
    u = u.set_coefficient_val(2, i, vu)
    islmap = islmap.add_constraint(u)
    islmap = islmap.add_constraint(l)
  return islmap
```

The integration of constraints into an ISL map is cumbersome, as you have to create `isl_constraints` first, which is only possible with ISL values. An ISL constraint allocated as an inequality has generally a form of $a \cdot x + const. >= 0$, where $a =$ coefficient and $const. =$ constant. The variable x can be either a space parameter or a domain variable. That means you can also impose a constraint on e.g. `N`. Therefore you have to specify the space where the chosen variable lives in. This is done by the line marked with `2 -> isl_dim_in`. That means if you want to have a constraint like $x >= 10$, you must set the coefficient to one and the constant to -10.

Now we must apply the help function to the maps which are contained in our `isl_union_map` object named `launch_acc`. Each union map contains a set of `isl_map` objects, which must be imposed with the constraints of each partition separately.

```
# init variable acc_part1
acc_part1 = islpy.UnionMap.empty(launch_acc.get_space())

def setPartition1Boundaries(m):
    # type cast
    global acc_part1
    m = islpy.UnionMap.from_map(setPartitionBoundaries(m, p1))
    # union
    acc_part1 = acc_part1.union(m)
    return 0 # ISL status return code for "ok"
# call help function for every isl_map
launch_acc.foreach_map(setPartition1Boundaries)
```

Above mentioned `acc_part1` represents now an `isl_union_map`, which contains all accesses of `partition1` on array `in` for our specific kernel launch. Nevertheless it gives us only the accessed points, which are in our case two dimensional. Moreover `acc_part1` is still a map and *not* a set with points, thus we proceed with getting the image space of a map.

```
# Get image space
points_part1 = islpy.Set.from_union_set(acc_part1.range())
# We simplify their representation
points_part1 = points_part1.coalesce()
```

The range of the `isl_union_map` is per definition a `isl_union_set`, but as all maps in our `isl_union_map` must have the same image space (which is defined the array `in` and its dimension), thus the resulting `isl_union_set` *must* be simply convertible to an `isl_set`. Furthermore the simplification of the representation is an important step, to get contiguous intervals in case of one dimensional access maps. If you omit this step it could happen that your index calculation results in one interval $[0, 10]$ and another one with limits $[11, 20]$, although this is obviously one single interval $[0, 20]$. To make it even worse omitting this function call can result in an overwhelming amount of small CUDA memory copy operations while resolving inter kernel dependencies.

Lastly, we delinearize the two dimensional accessed points:

```
def getMinAndMax(islset, dim):
  ...
def twoDimSet_to_1D_intervals(islset, width):
    intervals = []
    min_y, max_y =  getMinAndMax(islset, 'y')
    for y in range(min_y, max_y + 1):
        val = isl.Val.int_from_ui(islset.get_ctx(), y)
        # crop the set to one specific y-value
        line = islset.fix_val(3, 0, val)
        min_x, max_x = getMinAndMax(line, 'x')
        intervals.append([min_x + y * width, max_x + 1 + y * width])
    return intervals
# Get the actually accessed indices on array ‘in‘
intervals_part1 = twoDimSet_to_1D_intervals(points_part1, N)
```

To get the indices we have to delinearize the set into intervals with knowledge of the array width `N`. We achieve that by iterating over all y-values contained in the set, fixing that y-value, and searching for the minimum and maximum in the x-dimension. This must always be a contiguous interval, as this is done for every `isl_basic_set` which is always a contiguous set.

The stencil used for this example has a shape of a '➕', which results in a shape of '▪' for the access area of our attended partition. That is a rectangle with the corner elements cut off. In fact the access intervals of the second partition can be written as `intervals_part1`$= \{[s, e) | s = 481 \wedge e = 511 \vee s = 993 \wedge e = 1023 \vee s = 512 \wedge e = 992\}$.

Remark on the Generality satisfied by Mekong's Runtime Library

All the examples, which have been shown in this chapter, are highly customized for one single case. They should help you to get an impression of the key tasks, which are done in the runtime library. Another not mentioned task is the calculation of indices due to inter kernel dependencies. E.g. if we started the the partitioned stencil kernel another time, after swapping the two device buffers, there would be a non-empty empty intersection between the written elements in the first launch and the read elements in the second launch. These border elements can be calculated in a similar fashion like the examples shown above. The runtime library can calculate for any number of devices such dependencies between any kernel launches; and for arbitrary chosen partitions.

If you want to try the calculations by yourself, they are available in one python code snippet[6]. The next section deals with the inter kernel dependency resolution, which is basically built of the calculations done above.

3.4.3. Inter Kernel Dependency Resolution

This section will firstly explain why there arise inter kernel launch dependencies while partitioning an application to multiple GPUs. Secondly, it will present a solution addressing the stated problem.

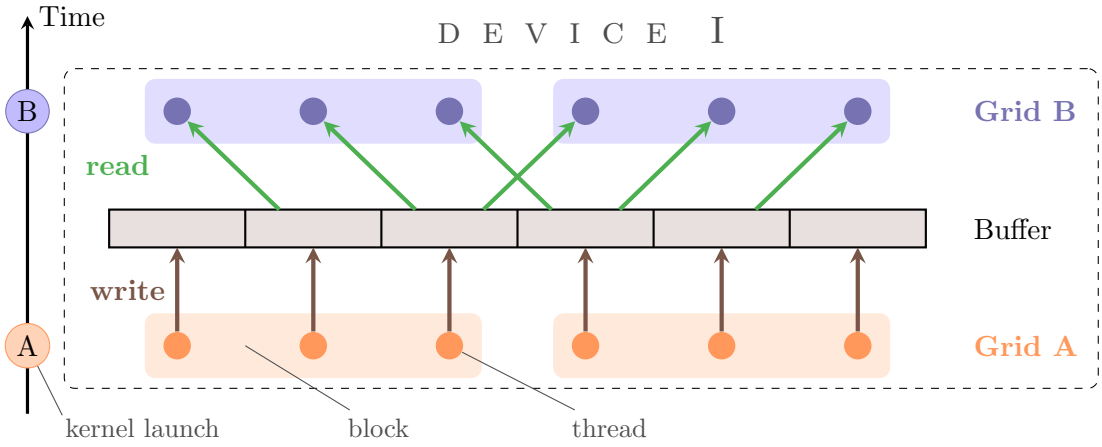


Figure 8.: Two different subsequent kernel launches with an equal execution grid, but different access pattern on the same device buffer. Kernel A writes, whereas kernel B reads the buffer.

Reason for Dependencies Arising

To state the problem appropriately, we have to define what differs kernel launches from each other. We consider two kernel launches to be **different**, if they differ at

[6] `https://gist.github.com/codecircuit/8cbcc76335a86b0298239a9a9e694dbf`

least in one of the following points: (1) shared memory size, (2) grid size, block size, (3) kernel function, and (4) values of arguments. In case of a stencil code that means, that the second kernel launch, after swapping two device buffers, differs from the first one, because two kernel argument values have been changed.

We will now introduce a simple example, which emphasizes the occurring of inter kernel dependencies. In figure 8 you can see a sketch of two subsequent different kernel launches on one GPU. There is kernel launch A, which writes to the drawn buffer, and kernel launch B, which reads from the same buffer, after kernel launch A was launched. We assume that the programmer instructed a device synchronization between both kernel launches, to ensure the validity of the data, which is read by kernel launch B. You can see the chronological behaviour along the timeline on the left side of the picture. Moreover both kernel launches have the same 1D kernel launch configuration 2×3 (grid size $\times$ block size). The buffer is used in both kernels as the same array with six elements of equal type. Kernel launch A simply instructs every thread to write at its own position, e.g. its ID. Subsequently, kernel launch B orders its threads to read the data in a cross pattern, as it is shown in figure 8. What calculations might follow with the read data in kernel launch B is not relevant. We assume that kernel launch B reading the data is a mandatory process for successive calculations.

All in all the above exemplified situation is not astonishing, but an inevitable preparation for a modified version. As explained in section 3.4, our workflow partitions the original grid in a way , which is determined in the analysis part. Suppose there are two GPUs in the system and as there is only one dimension greater than one in both kernel launches, it is the only dimension the grid can be partitioned at. We do not split block borders, thus we end up with one device executing one block. This situation is shown in figure 9. In section 3.4 was mentioned that every allocated device buffer within the simulated one GPU environment in the application will result in an allocation on every available device within the Mekong context. That means that the buffer on device I & II are physically different, but of equal size. While partitioning the access pattern of a kernel is not modified, thus as every block is aware of its position on the super grid, the first block of launch A writes the first part of the buffer on device I, whereas it is vice versa on the second device. As it is shown in figure 9, one thread on each device of kernel launch B reads an element, which was not written by kernel launch A on the same device previously.

This represents the main reason for inter kernel dependencies arising while partitioning kernel launches to different GPUs. The following section explains how we detect and resolve inter kernel dependencies.

Solution

As the runtime library has been written in an object oriented fashion we will address the problem theoretically on a similar approach. This is useful if you want to understand the functionality of the runtime library by heart. For a more abstract functional view on the problem you can read section 3.4.5.

As the project is in a state of development, we started to address the inter kernel dependency resolution by making a simplifying assumption, that

Every kernel which writes to a buffer invalidates all previous writes to the same buffer.

Buffer in that sense means the device buffer in the application context. Keep in mind that every buffer in the application context belongs to `numGPUs` buffers in Mekong's context. For several applications (e.g. Stencil Code, N Body Code, Matrix Multiplication, Reduction) this assumption is *not* a restriction, and they can be processed in our workflow as if the assumption has not been made. But applications, which write to different parts of a buffer using several kernel launches, like e.g. a mesh refinement code are excluded by our assumption. Figure 10 emphasizes the writing behaviour of a mesh refinement code on 2D buffer. You can imagine the presented access to belong to a 2D stencil code, which dynamically adjust its writing window. The result is a buffer with elements written last by *different* kernel launches. The first and second iteration represents a partitioned kernel launch to three GPUs, which wrote to their indicated area. If the red marked third iteration takes place subsequently, and wants to read the area of the rectangle it is necessary to save the last writing GPU for every element of the buffer.

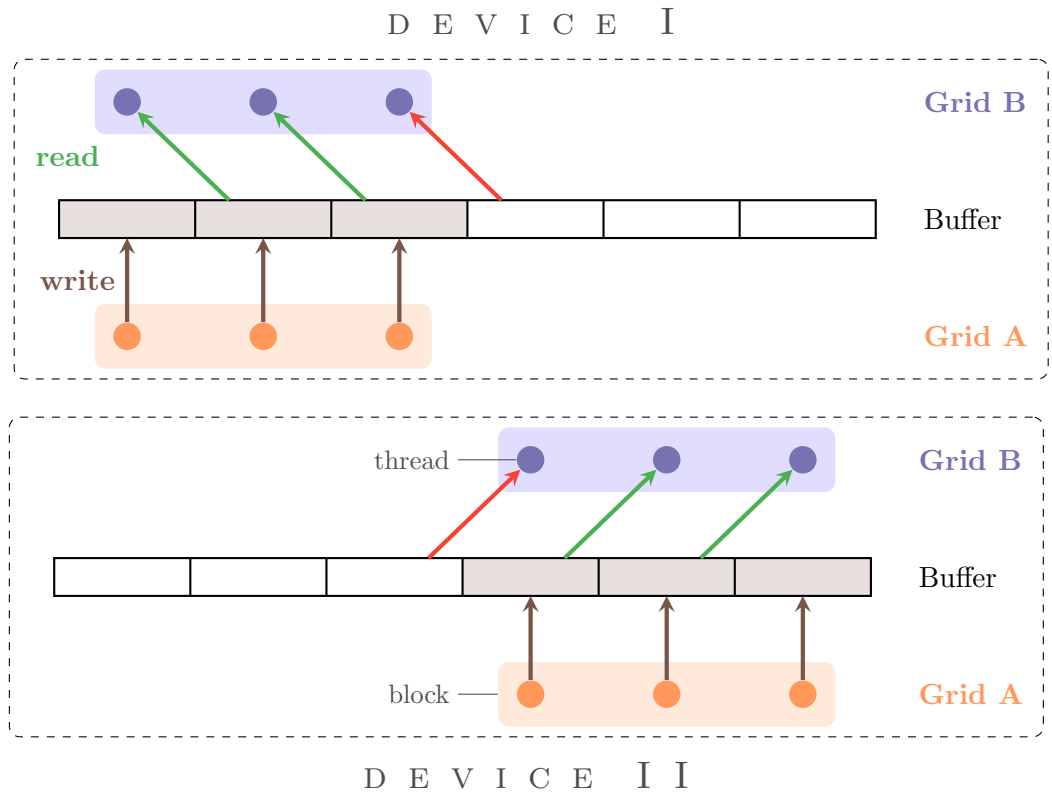


Figure 9.: Two different subsequent kernel launches with an equal execution grid, but partitioned to two GPUs by Mekong's workflow. The figure emphasizes what would happen if we omitted inter kernel dependency resolution. The non-partitioned version is shown in figure 8.

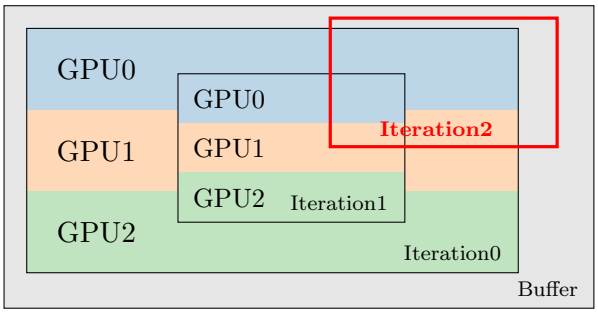


Figure 10.: 2D Stencil Code as a mesh refinement code, which was partitioned to three GPUs.

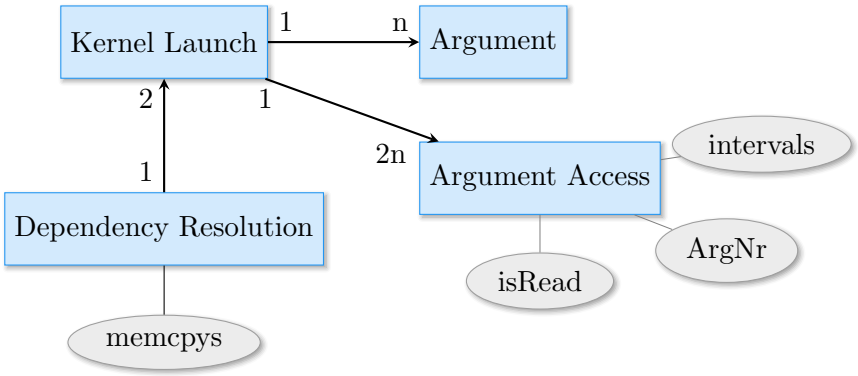


Figure 11.: Conceptual class diagram representing a part of the runtime library.

To continue with the benefits of our assumption we can now declare that there might be a dependency resolution between two kernel launches necessary. Thus we can uniquely identify an dependency resoltion object by considering both kernel launches, it is associated with, as a composed primary key. Figure 11 shows a conceptual class diagram of the runtime part concerning the dependency resolution. As it is shown every kernel launch scheduled in the application context, will be modeled as a kernel launch object in Mekong's context. Each kernel launch has n arguments, which can be accessed by a write or read instruction. An object of class `ArgumentAccess` describes either the read or the write access on *one* kernel argument by all GPUs. We define the access on a non-pointer argument as empty. You can imagine an object of class `ArgumentAccess` to be a set of accessed intervals[7] on a `kernel_array`. This object is created on demand to avoid unecessary calculations.

A dependency resolution object itself does not contain points or indices anymore, but rather the actual memory copy operations, which are mandatory to resolve dependencies between two kernel launch objects. In comparison to an argument access object a dependency resolution object can concern memory copy operations on various device buffers. E.g. if kernel launch B reads two buffers, which have previously been written by kernel launch A, it is mandatory to execute resolving memory copy

[7]This is very similar to the `isl_set` we calculated at the end of section 3.4.2.

operations regarding both buffers. When creating a dependency resolution object, we need two different kernel launches, which indirectly contain their used buffer and argument access objects. The procedure of creating a dependency resolution object is: (1) Get a list of buffers, which are read by the second and written by the first kernel launch, (2) get the respective argument access calculated (this includes delinearization if necessary), (3) compute the intersection of accessed points between every possible GPU pair, (5) with the intersected intervals and the corresponding buffer the memory copy operations can be set up.

A dependency resolution object can be created on demand our wrapping function for the `cuLaunchKernel` function. This is done by comparison of the buffers, which are read by the current kernel launch, with a map, where we save all previous writes to all device buffers. If a read buffer is already contained in that map, it was written by a previous kernel launch and had been linked to the appropriate kernel launch object. By this procedure we are able to get all kernel launches, which wrote to a buffer last the current kernel launch reads. Consequently all required dependency resolution objects can be created.

The above outlined procedure might seem to you computational expensive, and maybe as time consuming as the actual calculations done on the GPUs. The next section will explain the hierarchical caching system, which was implemented to enable an efficient execution of the above mentioned calculations.

3.4.4. Hierarchical Caching System

In order to implement an efficient caching system the compute intensive parts of the runtime must be located first. This was done by evaluating benchmarks, which has been done continuously while working on the software. The most compute intensive parts in decreasing order of their complexity were determined to be: (1) delinearization of 2D access maps, (2) calculation of accessed intervals from the corresponding ISL access map, (3) creation of the dependency resolution objects, and (4) partition creation. Point one and two belong in general to one comprehensive point: the calculation of accessed intervals. This is due to the fact that point one is a requirement for point two in case of two dimensional access patterns. Moreover both points can hardly be implemented separately in software. Mekong's runtime library addresses both points with its argument access class (`ArgAccess`), which at the moment of creation calculates the accessed intervals in **parallel** on host side.

The third point means the calculation of the inter kernel dependency resolving memory copy operations. This can only be done while making use of data computed by an object of class `ArgAccess`. The result of this process is saved in an object of class `DepResolution`.

The last point regards the partitioning of the super grid to different GPUs. Which is done while creating an object of type `KernelLaunch`. For every different[8] kernel launch, which is scheduled in the application context, Mekong's runtime library will create an appropriate kernel launch object to handle further calculations.

Up to now we stated three classes which handle the most compute intensive

[8] If you want to know what makes kernel launches different, you should read section 3.4.3

parts of the runtime library. Exactly these three classes will be cached and thus not redundantly created. In specific that means that our wrapping function for `cuLaunchKernel`, which receives the kernel launch configuration from the application, checks if a kernel launch object has already been created with the specific given kernel launch configuration. The caching of the kernel launch class includes automatically the caching of the `ArgAccess` class, because an argument access object must belong to a specific kernel launch object and can not exist on its own.

An object calculating the inter kernel dependency resolving memory copy operations can uniquely be identified with its two belonging kernel launch objects. The order of both kernel launch objects matter, because the chronological order between kernel launches matters when resolving inter kernel dependencies. While implementing the caching of `DepResolution` objects, we can impressively benefit from the caching of kernel launch objects. Each kernel launch object describes one single uniquely kernel launch, thus each pointer to a kernel launch object points to a different kernel launch. In consequence we can basically answer the question: 'Does there already exist a dependency resolution between kernel launch A and B?' by a simple comparison of two pointers. You can imagine the both kernel launch object pointers to be a composed primary key for the dependency resolution class.

If we apply our caching scheme to a simple stencil[9] code as a test case, there will be only two different kernel launch objects. Although always the same kernel function is called, a buffer swap after each launch changes the kernel arguments. In consequence there is a kernel launch A and B. While the application is running, the runtime will create at maximum two different dependency resolution objects (A$\longrightarrow$B & B$\longrightarrow$A). Moreover the runtime recognizes that the kernel launches are different, but have the same `ArgAccess`, because the argument access does not depend on non-scalar kernel arguments. Thus only two argument access objects will be created. One for the read access on the input array and another for the write access on the output array.

3.4.5. Discussion: Bulk of Functionality Contained in Runtime

After reading chapter 3.4 you might think that a bulk of Mekong's functionality is located in its runtime library, which is up to now the case. That this does not contradict to the original design goal of Mekong: 'do as much as possible at compile time' is explained in this section.

The design goal is held if we identify mandatory tasks, which are done in the runtime library, on a theoretical level and show that they can not be transferred to an execution at compile time. An elementary part of the runtime library is the detection and resolution of inter kernel dependencies. On an abstract level you can think of a function calculating the necessary memory copy operations, which resolve the dependencies between two kernel launches, to be like $R(K_{\text{prev}}, K_{\text{curr}}, S_{\text{prev}}, S_{\text{curr}}, G_{\text{prev}}, G_{\text{curr}}, M_{\text{prev}}, M_{\text{curr}})$, which depends on K =current or previous kernel function, S = curr. or prev. scalar kernel arguments, G = prev. or curr. grid, and M =mapping between kernel array and used buffer. M is mandatory as a buffer can be used for different kernel arrays, like it is done in a stencil code (see section 4.2). Function R in turn depends

[9] This refers to a stencil code presented in section 4.2

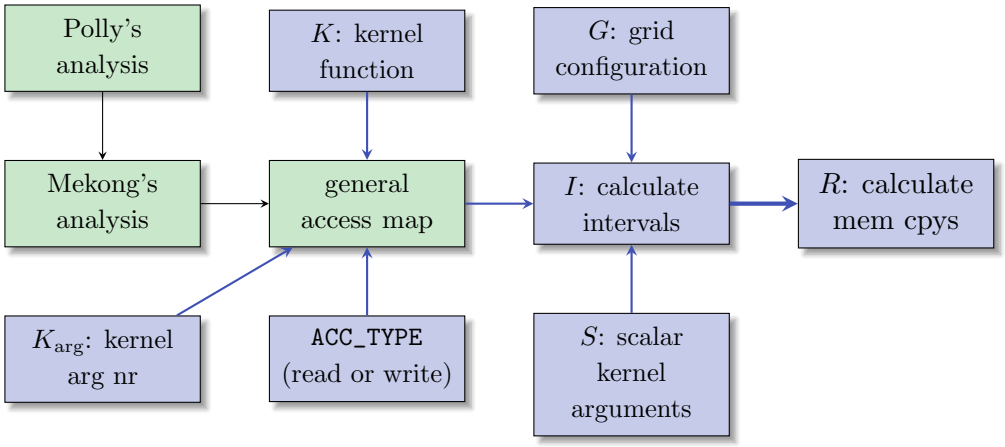


Figure 12.: Dependency graph for the calculation of memory copy operations in case of a inter kernel dependency resolution. Teal entities are available at compile time, whereas blue entities can only be obtained at runtime.

on another function I, which calculates the accessed intervals of a specific kernel launch on a `kernel_array`. Function $I(K, S, G, K_{\text{arg}}, \texttt{ACC\_TYPE})$ depends moreover on K_{arg} = kernel argument number, and on `ACC_TYPE`, which can be either a read or a write access. Figure 12 shows a sketch of the dependency relations for above mentioned functions. Essential is that nearly all arguments of both functions are earliest available or chosen at application runtime. For that reason the dependency resolution as a main part of Mekong's runtime library can only be done at application runtime.

3.5. Usage

This section provides information necessary for running the software yourself. Feel free to download the software[10] yourself.

3.5.1. Requirements

If you want to compile and use Mekong's workflow in its current state, you will need:

1. CUDA 7.5 Toolkit
2. LLVM 3.9.0. Even better with git hash 314a1b6403bc971a9d68d7eed3a0ce1359d03656
3. clang compiler of version 3.9.0. Even better with git hash f847bad18bcc46b5b47d32274a9d53a50209e555
4. the Polly library of version 3.9. Even better with git hash 48731f9f119b03d7dc8b73aa7dd24720c4859645

[10] `https://github.com/codecircuit/apdpp`

5. ISL library of version 0.16.

3.5.2. Procedure: New Test Case

I recommend the following procedure when introducing a new test case:

1. Write the kernel code in pure C and check Polly's analysis on that code if there are any reasonable domains.

2. Wrap the body of the in-C-written kernel with three for-loops to simulate the execution on a grid. And check Polly's analysis on that code. If Polly is not able to detect any access patterns for this case, there will be hardly any reason to proceed with further steps. Thus if Polly fails with its analysis you should find out which part of the code makes Polly aborting (comment certain lines out). It might be possible to write the code in a manner, which is analizeable for Polly.

3. Do the latter step again, but without writing the three for-loops by hand. Use the provided loop wrapping pass contained in Mekong's development repository instead.

4. Do step two again, but now with the OpenCL kernel.

5. Do step two again, but now with the OpenCL kernel and without wrapping the three for-loops by hand. Use the loop wrapping pass for that purpose. If this step succeeds, the test case will be handled correctly by Mekong's workflow.

6. Use the analysis and transformation passes to modify the application, and link the runtime library to it. Do not forget to set the partitioning dimension in the analysis pass, as this is not done automatically; and to choose the wanted runtime library configuration in `CONFIG.txt` before compiling the runtime library.

3.5.3. Constraints

The input program must up to now hold the following constraints:

1. The host code must not be multi threaded

2. No pointer shifting and overlapping; That means if you want to start a kernel, you *must not* use any device memory pointer, which was *not* allocated using `cuMemAlloc`. Especially you must not use a device pointer, which was calculated from another device pointer (e.g. `CUdeviceptr devPtr_1 = devPtr_0 + 10`).

3. All CUDA Driver function calls must be done directly in the main function of the program; or must be inlined.

4. The device kernel must have only regular memory accesses on global memory. We consider a memory access to be regular if the accessed index is a quasi affine function of scalar kernel arguments, the thread or block IDs, and the kernel launch configuration only (grid size, block size, etc.), and compile time

constants. In consequence it is not allowed to access global memory like e.g. `devPtr[map[idx]]` or `devPtr[idx * idx]`.

5. Any kernel argument, which is used to calculate any global memory access must be an integer[11] or floating point type obeying IEEE754 standard.

6. The host code must be written with CUDA Driver API and the device code in OpenCL.

7. The size of a logical multi dimensional `kernel_array` must be dependent on only one single scalar kernel argument. For example you are not allowed to write a two dimensional access like `devPtr[x + (N + 1) * y]`, because this would imply an array width of `N + 1`. An array of width `N` is fine.

8. You must not choose a 3D partitioning scheme.

9. The runtime library makes the assumption that a kernel writing to a device buffer invalidates all preceding write processes. Thus you will not get back any data to the host, which was written before the last kernel launch wrote to that buffer. You will get only the data from a buffer, which was produced by the last kernel launch writing to that buffer. This assumption is obsolete if we redesign the dependency resolution model (see section 5.4 for more about that).

10. The size of all device buffer allocations must not exceed the memory limits of the smallest GPU memory within the node.

[11] `char` & `bool` are also an integer type in LLVM IR, thus they are permitted.

4. Test Cases & Benchmarks

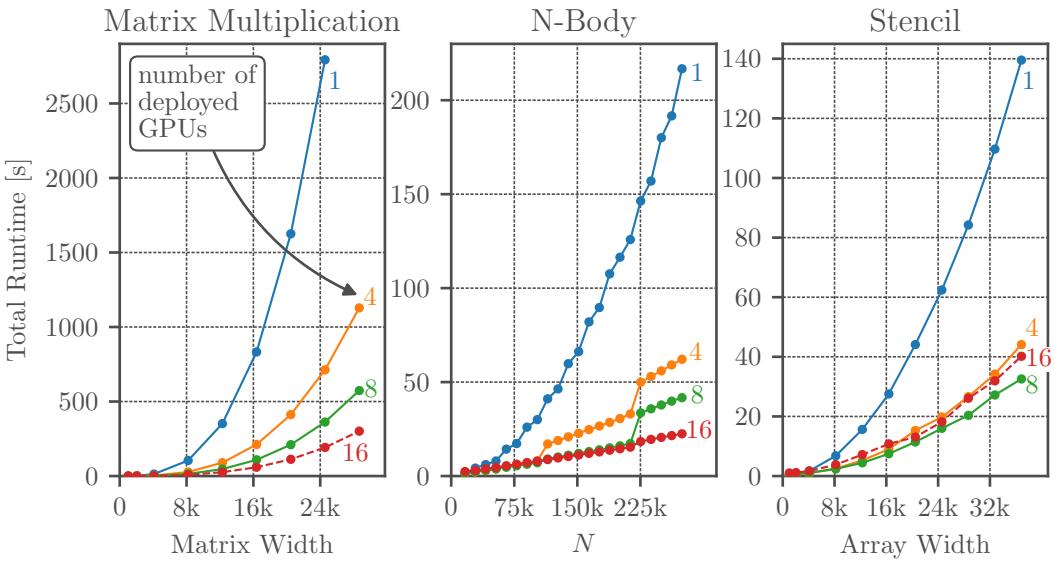


Figure 13.: Performance overview of all three applications. The y-axis represents always the total application runtime in seconds.

We decided to make three large benchmarks to check the functionality and performance of our workflow: (1) Matrix Multiplication, (2) Stencil Code, and (3) N Body Code. You can see an overview of the overall application runtime in figure 13. The single GPU line was measured with the respective original code, whereas the other lines represent measurements of the transformed code. The device code of every application has been written in an intuitive way, thus we used one thread per output element and did not exploit shared memory usage. All benchmarks presented in this thesis were executed on a node, which was equipped with eight NVIDIA K80 graphic cards (= 16GPUs) and two Intel Xeon E5-2667 v3 @ 3.2 GHz and 264GB RAM @ 2133 MHz.

To have a common ground we must define the timespans, which have been investigated in our benchmarks:

Application HtoD Time time needed for all host to device memory copy operations measured in the original code. E.g. the code which measures this time could look for the stencil code like:

```
double timestamp = Clock::now();
cuMemcpyHtoD(..., N*N*sizeof(float));
cuMemcpyHtoD(..., N*N*sizeof(float));
double app_htod_time = Clock::now() - timestamp;
```

Application Kernel Time represents the time measured in the original application code, which has been spent for calculations on the device. E.g. for an iterative kernel launch the application kernel time could be measured by the code snippet below:

```
double timestamp = Clock::now();
for (int i = 0; i < 1000; ++i) {
  cuLaunchKernel(...);
  ...
}
double app_kernel_time = Clock::now() - timestamp;
```

Application DtoH Time the time measured in the original application for *all* device to host memory copy operations. A code snippet measuring this time could look for a stencil code like:

```
double timestamp = Clock::now();
cuMemcpyDtoH(..., N*N*sizeof(float));
double app_dtoh_time = Clock::now() - timestamp;
```

Kernel Time means the time which was needed for the true kernel calculation time, thus it represents the time for kernel calculations in Mekong's context.

Dependency Calculation time needed to calculate the accessed intervals *and* memory copy operations, which are mandatory to resolve inter kernel dependencies. This time is a subset of the application kernel time.

Synchronization and Communication Time represents the time which was needed to execute the memory copy operations due to inter kernel dependencies. This time is a subset of the application kernel time.

Interval Calculation Time time to calculate accessed intervals with the usage of memory access patterns. This time includes the linearization in case of 2D access patterns. Moreover this time can be *partially* contained in the dependency calculation time, but also in the application device to host time. This is the case, because it can happen that there is a device to host memory copy operation on a buffer, which has not been used for any inter kernel dependency calculations yet. This happens for a Matrix Multiplication, as there is only one kernel launch, which does not cause any inter kernel dependency resolutions. Nevertheless in order to get the correct elements back to the host, there must be an interval calculation.

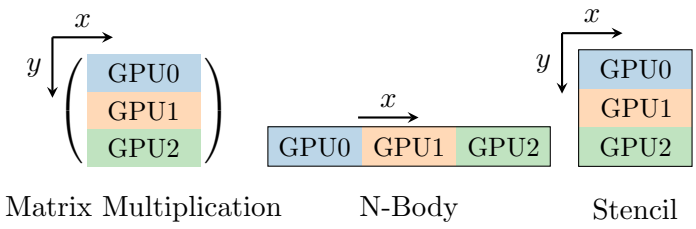


Figure 14.: Example partitioning in case of three GPUs.

If you want to investigate the benchmark results by yourself, you can download the bare data for the matrix multiplication[1], stencil code[2], and N body code[3].

4.1. Matrix Multiplication

In this section we will take a look at the performance of a squared matrix multiplication, which has been transformed by our workflow. A multiplication of two matrices, each with a size of $N \times N$, results in a matrix with an equal size. In figure 15 it is illustrated that each element in the output matrix is calculated by adding n products between elements of matrix A and B. Thus the complexity of a matrix multiplication in general is located in $\mathcal{O}(N^3)$.

The original application multiplicates two floating point matrices each of size N^2, and uses the kernel which is shown in figure 16. Besides choosing the splitting dimension (we split along the y-dimension, see figure 14), we had to use an analogous C kernel instead of the OpenCL kernel as an input for Mekongs analysis part. This temporary solution was essential, because Polly's analysis fails on the OpenCL matrix multiplication kernel, but succeeds on the C version. That the analysis works on an analogous C kernel, shows that this is an issue with the LLVM intermediate representation of OpenCL code in combination with the usage of the Polly library. As the LLVM IR experienced various changes in its shape along the last version updates, it is probable that this is also the case for future versions, which could directly affect Polly's memory access pattern analysis. Moreover as Polly is at the moment in a state of active development, we expect drastic changes in their implementation too. As our analysis is bounded to Polly and LLVM IR, which development is out of our scope, we have decided to concentrate our effort on progress, which represents the core logic of Mekong, rather than working on something, which might be fixed in a better way by external libraries anyway.

The matrix multiplication was chosen as the first test case, because of its simplicity, which in consequence does not require any inter kernel dependency resolution, because the application consists of one single kernel launch. In consequence the time for dependency calculations or communication and synchronization between GPUs will be equal to zero.

[1] `https://gist.github.com/codecircuit/d8d58eab0a0a591b5c625b0b5eae7459`
[2] `https://gist.github.com/codecircuit/d1067257e453f2500be68c24c06799ac`
[3] `https://gist.github.com/codecircuit/ea60dccff3ee2643fcddfdf889ce7ac7`

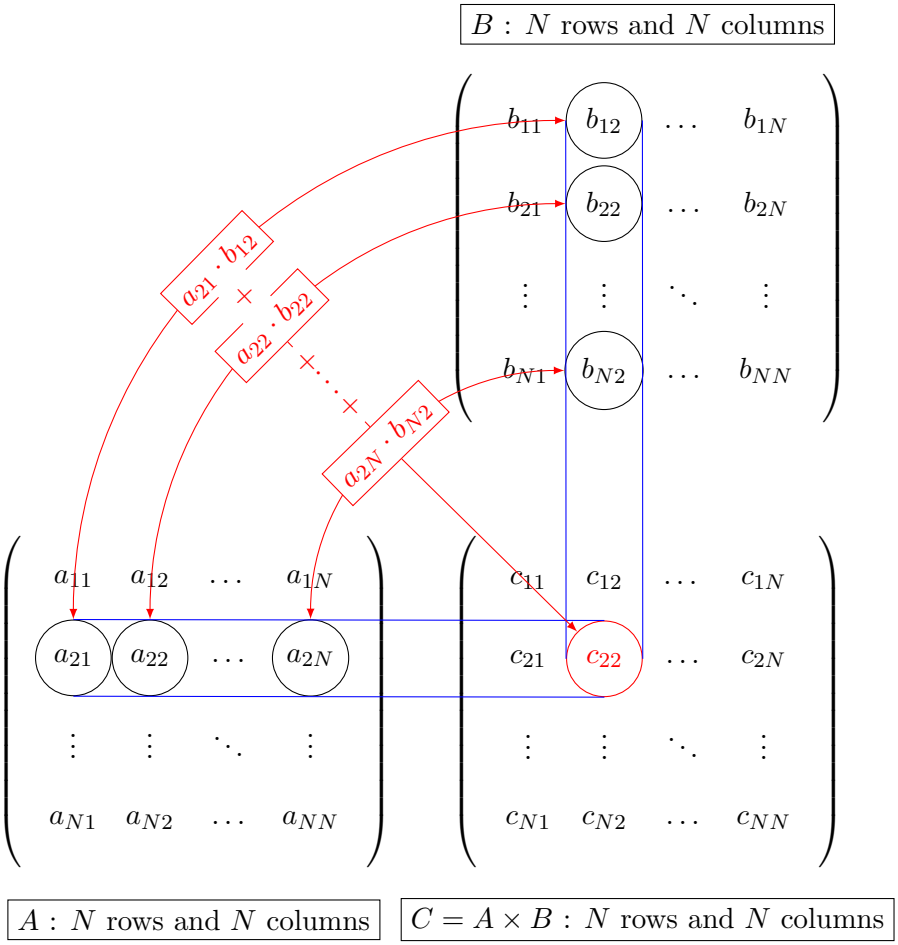


Figure 15.: Sketch of a squared matrix multiplication.

```
1  __kernel void mat_mul(__global float* C,
2                        __global float* A,
3                        __global float* B,
4                        int N) {
5    int tx = get_global_id(0);
6    int ty = get_global_id(1);
7
8    int m;
9    float acc = 0;
10   for (m = 0; m < N; ++m) {
11     acc += A[tx * N + m] * B[m * N + ty];
12   }
13
14   C[tx * N + ty] = acc;
15 }
```

Figure 16.: OpenCL kernel code for the squared matrix multiplication.

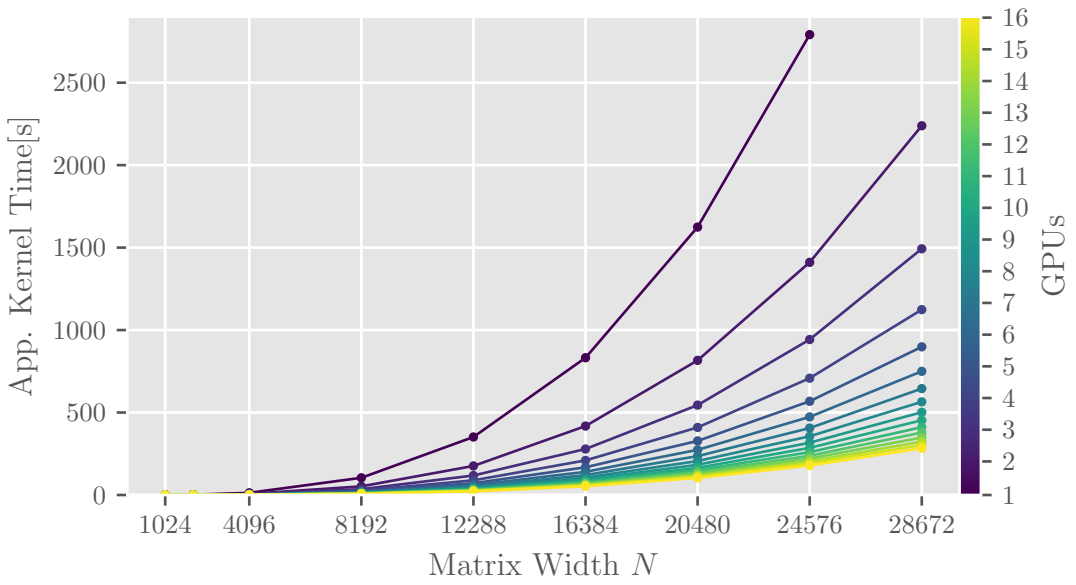


Figure 17.: Application kernel time of the matrix multiplication benchmark.

The time measurements have been done for matrix widths of $N = 1024 \cdot i$ for $i = 1, 4, 8, ..., 28$, and deploying one to sixteen GPUs.

4.1.1. Results

Figure 17 shows the application kernel time, defined in the beginning of this chapter. One point was excluded from the benchmark, as the computing time exceeded a appropriate limit. You can clearly that the lines represent the behaviour of a exponentiation, which we expect to be cubic. A quantity regarding the scalability of the transformed application is the speedup to the original single GPU execution, and the kernel speedup efficiency (E_{KS}) in dependency of the number of deployed GPUs (N_{GPU}). If $E_{\mathrm{KS}}(N_{\mathrm{GPU}}) = kt(1)/N_{\mathrm{GPU}}/kt(N_{\mathrm{GPU}})$, $kt(i)$ will be the time spent for dependency resolution and kernel calculation deploying i GPUs. You can see the speedup of the *total* application runtime and its corresponding kernel speedup efficiency in figure 18. For $N \gtrsim 8\mathrm{k}$ we achieve except for $\approx 2\%$ a linear scaling in N_{GPU}, which is probably caused by the partition creation.

The next point is that we want to evaluate the fraction, which is spent in Mekong's runtime libary, and is thus an transformation overhead. This can be well presented in form of a bar diagram, which you can see in figure 19. The height of a bar denotes the total runtime, whereas the color indicates the time, which was spent in a specific part of the application or in Mekong's runtime library. It is striking that the time for the application device to host memory copy operations is not visible at all[4]. This is due to the high complexity of the kernel calculation, which exceeds with its cubic complexity the squared complexity of the memory copy operation for large matrices. It is remarkable that the device to host memory time is negligible small in comparison

[4]'What about the other timespans mentioned in the legend?' → There is only one kernel launch.

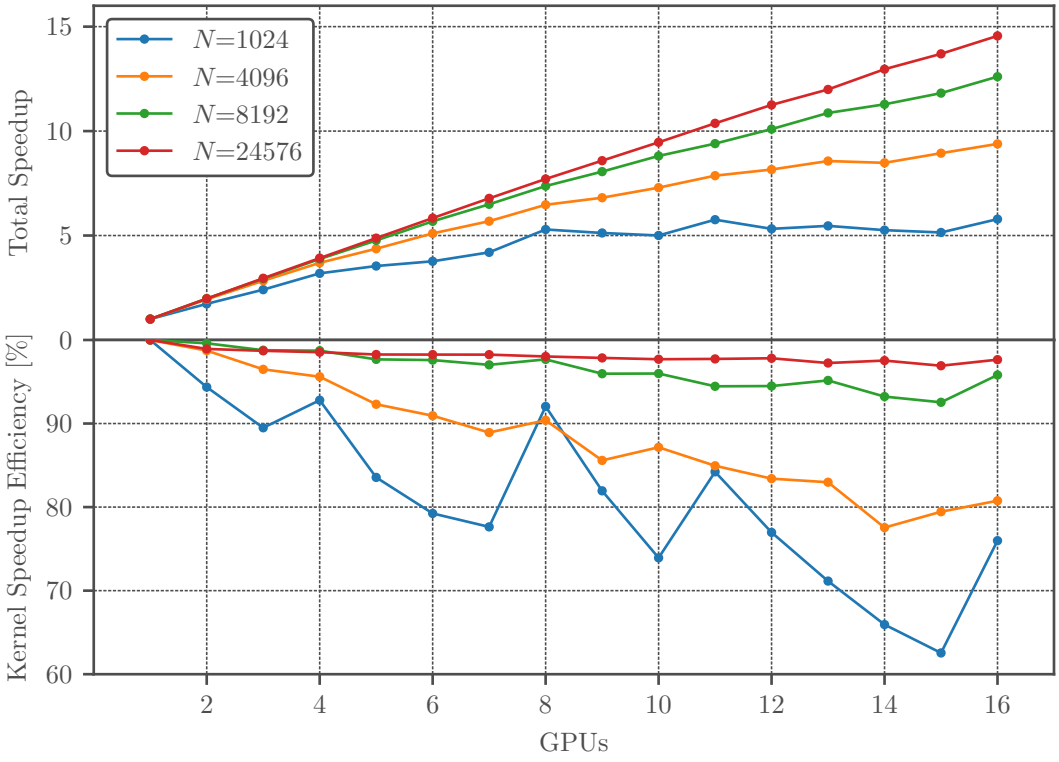


Figure 18.: Speedup and kernel speedup efficiency of the matrix multiplication benchmark.

to the kernel time, although it contains the interval calculation of Mekong's runtime library, which is necessary to get the correct elements back to the host.

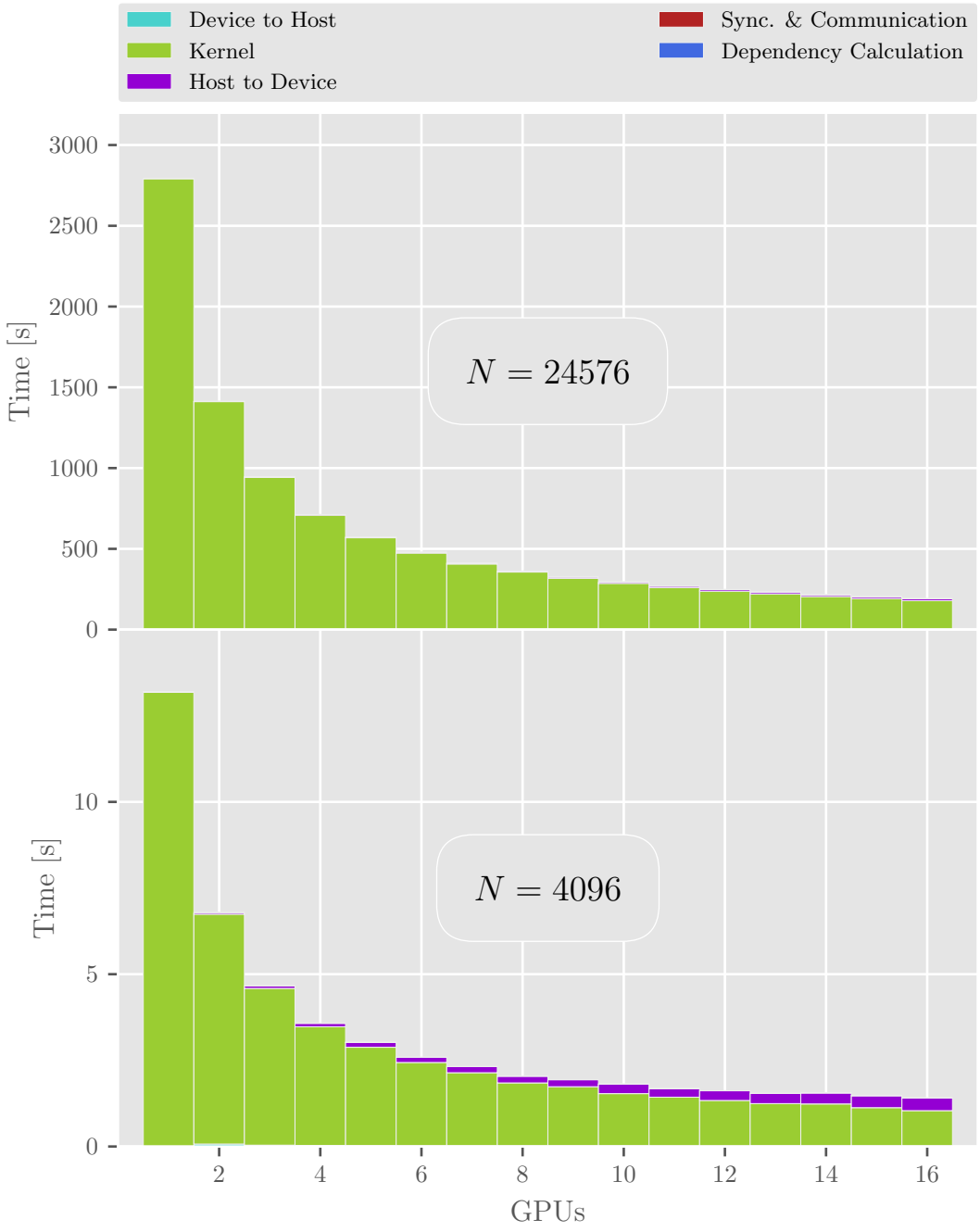


Figure 19.: Runtime breakdown of the matrix multiplication benchmark.

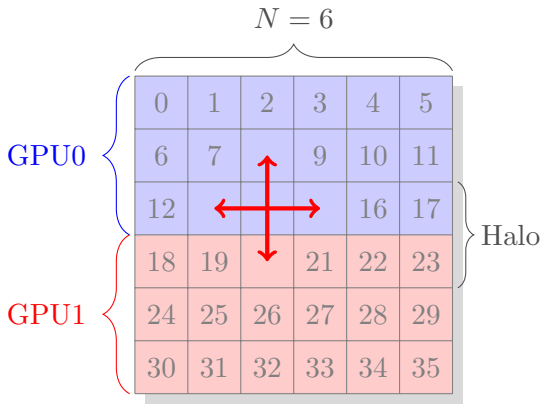


Figure 20: Example partitioning scheme for the used stencil code with two GPUs. Each number represents a super global thread ID, which works on a certain cell. The red arrows represent the data dependencies for one cell.

4.1.2. Conclusion

The matrix multiplication as a well know application for scalability and computational intensity shows its characteristics also in the benchmarks within Mekong's automatic partitioning workflow. The scalability to multiple devices is nearly linear, although the partitioning was done automatically. This underlines the productivity and performance increase for applications, which are as simple as a matrix multiplication. Nevertheless one should keep in mind, that the used kernel was written as simple as possible, without exploiting shared memory usage.

The next chapter will deal with a slightly more complex test case: a stencil code, which requires inter kernel dependencies to be solved.

4.2. Stencil Code

A code categorized as a stencil is used to solve problems, which evolve over time on a regular grid (e.g. the solution of the heat equation). Thus the kernel of such a problem statement is the rule how a cell's value can be calculated in dependency of its surrounding cells' values.

Figure 20 shows the data dependencies for a two dimensional five point stencil code, which was used for time measurements in this work. The figure shows that five memory locations must be accessed to calculate the new value of a cell. You can see the used 2D kernel in figure 25. This kernel is launched iteratively, after both device buffers have been swapped. The kernel is always launched with a block size of 32x32 threads.

Mekong's analysis part was completely able to analyze this application. As the partitioning dimension we have chosen the y-dimension, which was the only user input to transform the code successfully. Figure 20 shows an example partitioning scheme for two GPUs if $18 \mod \text{blockSize}_x = 0$. Otherwise the GPUs would have an unequal amount of work, as computational thread blocks are not split in Mekong's partitioning scheme.

If the partitioned stencil should be calculated for more than one time step, the GPUs need to synchronize and communicate after each time step. Figure 20 illustrates that consistent state of the GPUs will be preserved if they exchange the halo

region (elements [12, 23]) after each stencil computation. The dependency resolution management is done automatically by Mekong's runtime part, which uses the access and write patterns given by Mekong's analysis part. Other partitioning schemes, e.g. splitting contrary to the memory alignment for a kernel with a two dimensional access pattern, could lead to a performance decrease, as a halo region would not reside linear in memory. Thus many small memory copy operations per halo region would be scheduled, which could be more expensive than scheduling less but large copy operations.

The time measurements were made for 1000 kernel iterations, 1 to 16 deployed GPUs, where 1 GPU refers to the program execution in its original state, and the width equal to $N = 1024, 4096, 8192, 12288, 16384, 20480, 24576, 28672, 32768, 36864$ cells.

4.2.1. Results

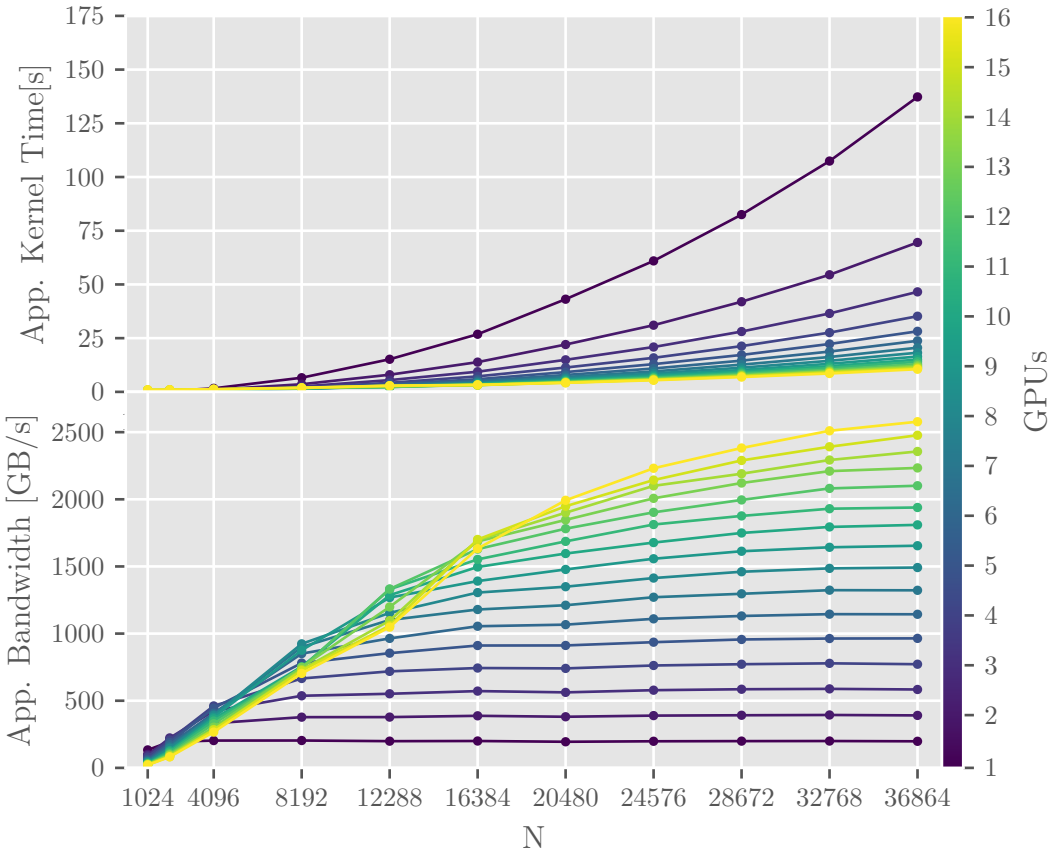


Figure 21.: *Top*: application kernel time represents the kernel plus synchronization plus communication time for 1000 kernel iterations in dependency of the array width N. *Bottom*: average total bandwidth on GPU memory.

First of all figure 21 top shows the time for the kernel plus the time needed

for communication and synchronization in dependency of the array width N. The quadratic behaviour, which is clearly visible for a low number of GPUs, is as expected, because the number of array points increases quadratic in N.

To visualize the performance increase the bandwidth on GPU memory is presented in figure 21 bottom, which was calculated with the time measurements shown in figure 21 top and the fact that one kernel launch accesses $N^2 \cdot 5 \cdot 4$ byte (five floating point numbers per cell) on GPU memory. As the used kernel has an operational intensity of 0.35 FLOP/byte, it is a memory bound kernel, thus the bandwidth on GPU memory was chosen as a performance comparison quantity. Furthermore the single GPU line points out that the bandwidth saturation for one GPU is reached at $\approx$ 200GB/s, which is equal to $\approx$ 82% of the K80's theoretical peak performance. Moreover the nearly equal distant lines for large N indicate a linear scaling of the bandwidth in dependency of the deployed GPUs.

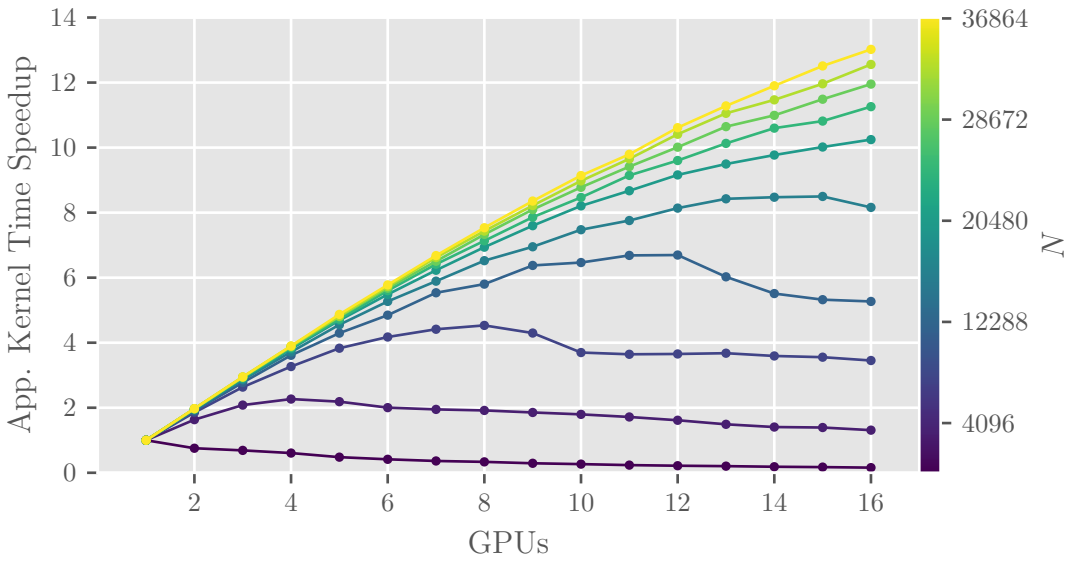


Figure 22.: Speed up to single GPU execution with the time measurements shown in figure 21.

Figure 22 shows the speed up to the single GPU execution calculated with the time measurements visible in figure 21 top. Thus this is not the total runtime speedup, but speedup of the application kernel time. It emphasizes the nearly linear scaling between the speed up and the amount of GPUs for large stencil widths.

Something which was not considered yet is the time needed for the memory copy operations scheduled by the user. First sending the initial data to the GPUs and second getting the final results back. Figure 23 shows that the time, which is needed to send the data, increases, whereas the the time needed to get the results back stays equal in dependency of the used GPUs. An exception is the one GPU line, because it represents only a simple CUDA memory copy operation. Of course is the complexity of a simple CUDA memory copy also located in $\mathcal{O}(N^2)$, but it is not clearly visible in this figure, as the other lines contain additionally the time needed to calculate

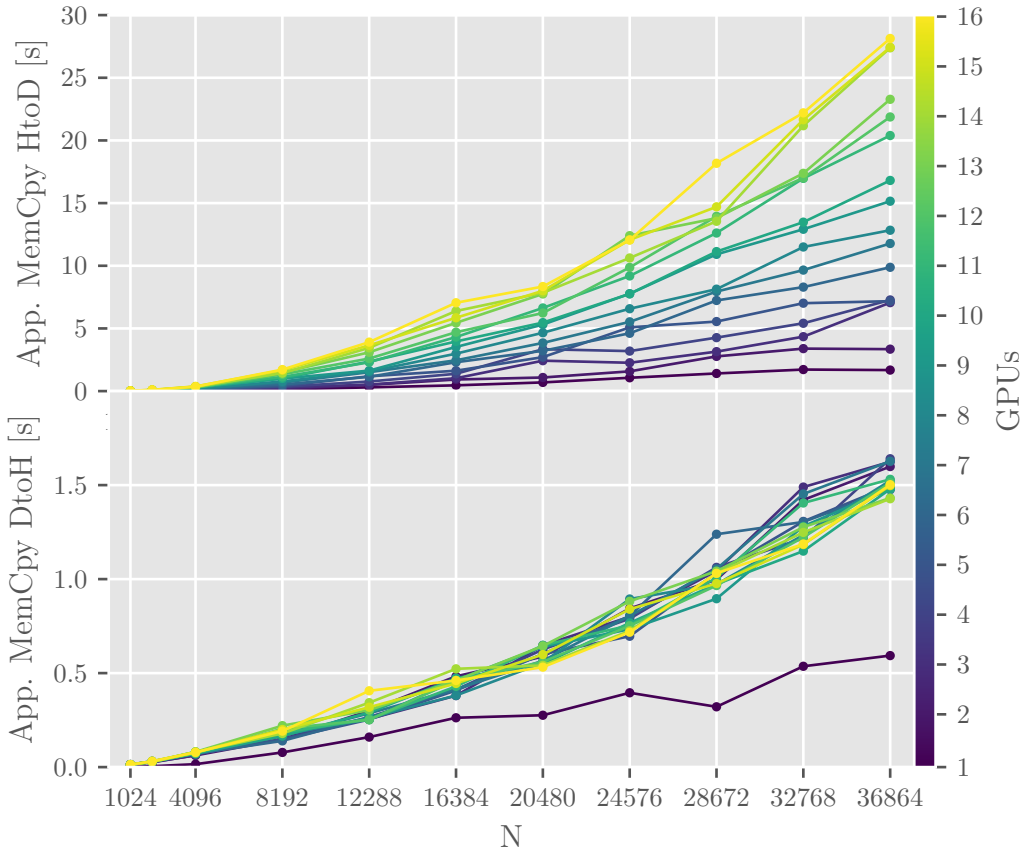


Figure 23.: Memory copy time for operations scheduled by the user in dependency of the array width N. *Top*: time for sending the initial stencil to the GPUs. *Bottom*: time for getting the result back.

intervals. The host to device copy time increases with the number of deployed GPUs, because sending the data is carried out in a broadcast, whereas getting the results back leads to a copy operation, which gets only the calculated results per GPU. This means that the total amount of copied data for sending increases linearly, though for getting data results, the amount of copied data stays equal in the number of used GPUs.

As it is not known a priori which thread will reside on which GPU, the broadcast ensures a consistent state for every kernel launch. Moreover it is visible in the figure that the time for sending grows relatively large for many GPUs in comparison to the kernel, communication & synchronization time shown in figure 21 top.

In figure 24 a bar's height denotes the total execution time of the program in dependency of the deployed GPUs. The total runtime is shown for two different array widths N. Both plots show similar trends in the total speed up, thus the total time does not decrease for more than ≈ 6 GPUs, as the host to device memory copy time grows accordingly. This could already be seen in figure 13. Nevertheless it should not be forgotten that the ratio between the host to device memory copy time and

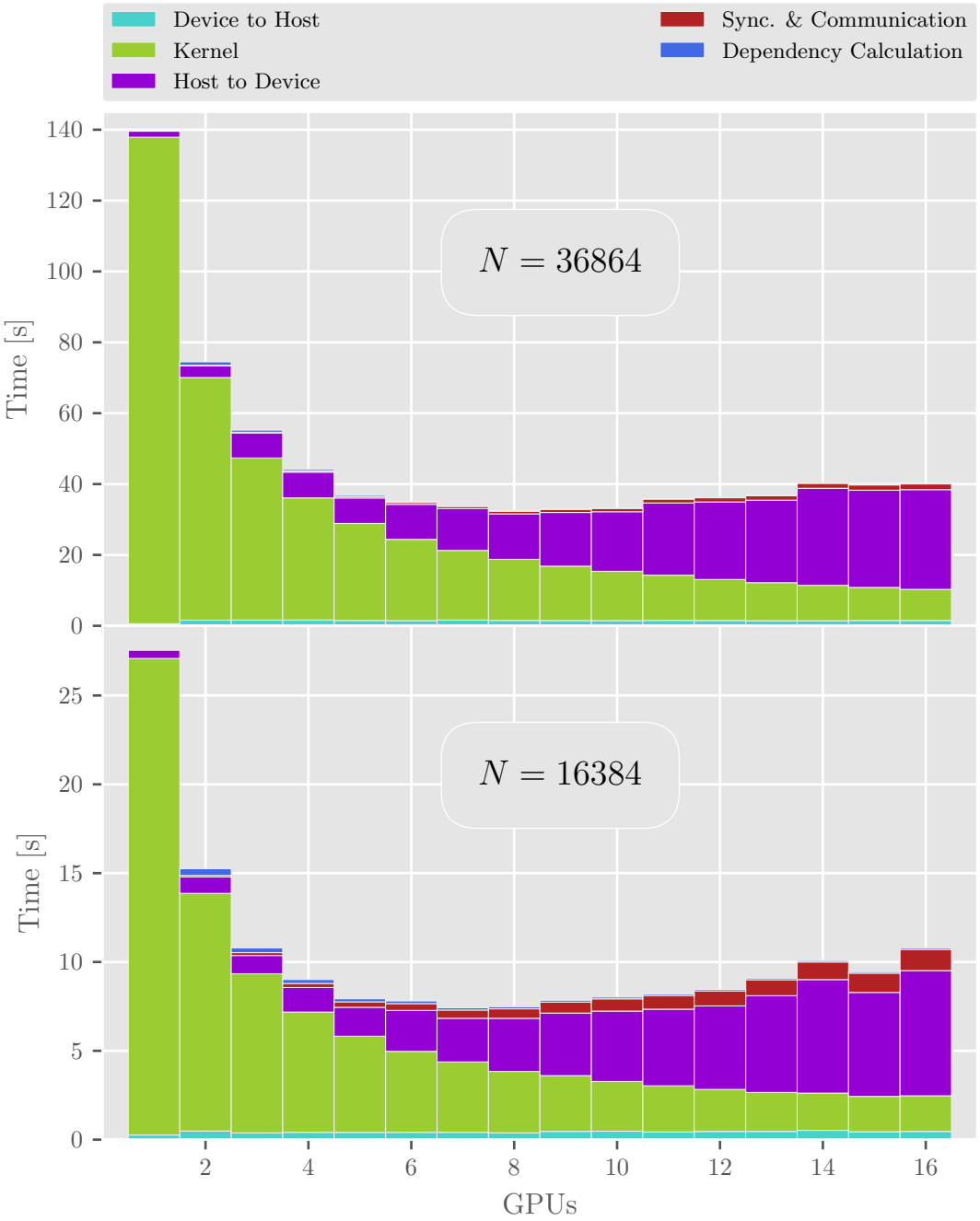


Figure 24.: Runtime Breakdown; The height of a bar denotes the total execution time of the stencil code. Moreover it is indicated by color how much time was spent in a certain part of the code.

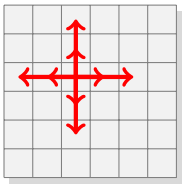
the kernel plus synchronization and communication time depends on the number of stencil iterations. The more iterations the smaller goes that ratio. In contrast the device to host memory copy time at the bottom of the bars stays nearly equal for

different numbers of GPUs. Moreover due to the efficient caching system (see section 3.4.4) we implemented, the time for dependency calculation is hardly visible.

```
1  __kernel void stencil5p_2D(__global float* in, __global float* out, int N) {
2    float res;
3    int x = get_global_id(0);
4    int y = get_global_id(1);
5    int id;
6    if (x > 0 && y > 0 && x < N - 1 && y < N - 1) {
7      id = x + y * N;
8
9      // the order of execution affects the result, as the operator
10     // precedence is not clearly defined in OpenCL, thus we list
11     // the order in an explicit way
12     res = in[id - N];
13     res += in[id + N];
14     res += in[id - 1];
15     res += in[id + 1];
16     res += -4.0f * in[id];
17     res *= 0.24f;
18     res += in[id];
19
20     res = res > 127 ? 127 : res;
21     res = res < 0 ? 0 : res;
22     out[id] = res;
23   }
24 }
```

Figure 25.: OpenCL kernel code for the used five point stencil.

4.2.2. Conclusion



First of all the measurements for the kernel plus synchronization and communication time show nearly a linear scaling in the speed up to the single GPU execution. This result underlines the usability of Mekong's tool stack for similar workloads. Not even increasing the number of cell dependencies to e.g. a 9 point stencil (picture on the right), would decrease the speed up significantly, as this only doubles the time needed for synchronization and communication, which would still be small in comparison to the kernel time. A huge impact on the overall time had the initial host to device memory copy, due to the used copy to all scheme. In contrary to the matrix multiplication benchmark, where the complexity of the initial data distribution is the same ($\mathcal{O}(N^2)$ with N = array/matrix width), the complexity of the matrix multiplication kernel lies in $\mathcal{O}(N^3)$, whereas the complexity of the stencil kernel is in $\mathcal{O}(N^2)$. Therefore the initial host to device memory copy operation has a more intense impact on the performance of the stencil code than on the the matrix multiplication. Nevertheless one should keep in mind, that an increase of the iterations reduces the performance drawback due to the initial broadcast.

Therefore this should be the starting point for an optimization. An idea is to

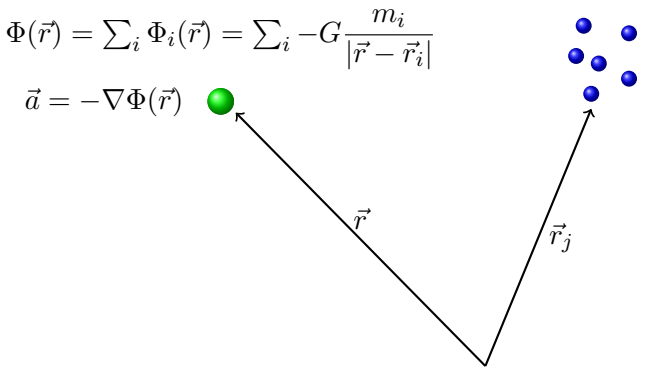


Figure 26.: Conceptual view of gravitational forces influencing a particle's acceleration due to changes of the gravitational potential.

stall the host to device memory copy until a kernel launch is scheduled. At that moment it is possible to copy only the data, which will be read. This would lead to a performance, which was shown for the device to host memory copy time. Nevertheless this would require putting more effort into keeping a consistent state, especially if there are several different kernel launches in a program.

4.3. N Body Code

An N body code in general deals with the movement of a various number of particles, which move in a space and interact with each other. Usually it is applicable for physical problems, which are stated by differential equations. For example the movement of particles in empty space due to the Newtonian gravitational force represents a well known case, which can approximately be solved with an N body code. You can see in figure 26 that the acceleration of one particle is influenced by the position of every other particle. The example assumes that we have point particles without any size. The formulas next to the green particle describe how the gravitational potential Φ at the position of the green particle can be calculated of the position of every other particle, and subsequently how this is related to the acceleration of the green particle. There exist several schemes how this differential equation can numerically be solved. We choose to have the Euler method for that purpose, as it is one of the most simplest methods, which is an appropriate choice for one of the first test cases. You can imagine the Euler method to be similar to a chess player, who displaces his chess pieces in a discrete way. Thus the Euler method takes a look onto the current board, calculates all new velocities and subsequently all new positions of the particles. After this have been done, the procedure starts from the beginning.

In our single GPU code we applied one thread to the calculation of the velocity and position of each particle. Therefore we have N threads in case of N particles, which represents a 1:1 relation. The particle positions and velocities are saved in three device buffers each: `float* pos_[x|y|z]` and `float* vel_[x|y|z]`. While a thread is calculating the new velocity for its particle, it must read all other particle positions. That implies that all velocities must be calculated before the positions can

be updated. A global synchronization between both calculations is the consequence, which is implemented by using two different kernels: one to calculate the velocities and the other concerning the position update.

```
auto timestamp = Clock::now();
for (unsigned i = 0; i < 50; ++i) {
  cuLaunchKernel(update_velocities, ...)
  cuCtxSynchronize(); // synchronize with device
  cuLaunchKernel(update_positions, ...)
  cuCtxSynchronize(); // synchronize with device
}
double app_kernel_time = Clock::now() - timestamp;
```

You can see the used kernels in figure 27. For the velocity update kernel, there was a gravitational softening implemented, which prevents velocities from growing suddenly too large. That can happen if two particles are positioned very close to each other due to discretization of time.

Mekong's analysis part succeeded completely on this test case, and the only user input was the choice of the splitting dimension, which had to be x-dimension, because we always launched a grid with a global size in threads of $N \times 1 \times 1$. The benchmark was done for particle numbers $N \in \{(16 + k) \cdot 1024 \mid k \bmod 12 = 0 \wedge k \leq 268\}$, and one to sixteen GPUs. The number of calculated time steps[5] was equal to 50.

4.3.1. Results

In figure 28 you can see a magnification of the application kernel time, which was already visible in the performance overview (figure 13). First of all you might recognize the step structure of every line, which is an artefact of the original single device code. Understanding this behaviour is delicate if we imagine to have an example GPU, which can process 10 thread blocks of our N body velocity update kernel[6] at once. Consequently the execution time would be equal if we started ten or less blocks. That means, for very compute intensive calculations, that the computation time of 11 blocks would suddenly increase, because the GPU would finish the first 10 blocks simultaneously, whereas the last block will be calculated afterwards. Except for one streaming multiprocessor all others are stalling, while the last block is processed. The above mentioned theory can be proved, because a consequence would be a doubling in the step distance if the number of GPUs is duplicated. You can see exactly the expected behaviour in figure 29, which shows a more fine granular measurement in N.

The inter kernel dependency resolution scheme caused by this test case differs from the stencil code. Whereas one threads reads a constant number of elements from global memory in case of a stencil code, a thread reads now the position of every particle. For that purpose the runtime library calculates an all to all memory copy scheme between the GPUs, and executes that between the iterations. You can

[5] One time step causes a velocity and position update for all particles
[6] The kernel which updates the positions is negligible small in its execution time

```
__kernel void update_positions(__global float* pos_x,
                               __global float* pos_y,
                               __global float* pos_z,
                               __global const float* vel_x,
                               __global const float* vel_y,
                               __global const float* vel_z,
                               float dt,
                               int N) {
  int id = get_global_id(0);
  pos_x[id] += vel_x[id] * dt;
  pos_y[id] += vel_y[id] * dt;
  pos_z[id] += vel_z[id] * dt;
}

__kernel void update_velocities(__global const float* masses,
                                __global const float* pos_x,
                                __global const float* pos_y,
                                __global const float* pos_z,
                                __global float* vel_x,
                                __global float* vel_y,
                                __global float* vel_z,
                                float dt,
                                float epsilon,
                                int N) {
  int id = get_global_id(0);
  // the body's acceleration
  float acc_x = 0;
  float acc_y = 0;
  float acc_z = 0;
  // temporary register
  float diff_x;
  float diff_y;
  float diff_z;
  float norm;
  int j;
  for (j = 0; j < N; ++j) {
    diff_x = pos_x[j] - pos_x[id];
    diff_y = pos_y[j] - pos_y[id];
    diff_z = pos_z[j] - pos_z[id];
    norm = diff_x * diff_x;
    norm += diff_y * diff_y;
    norm += diff_z * diff_z;
    norm = sqrt(norm);
    norm = norm * norm * norm;
    norm = norm == 0 ? 0 : 1.0f / norm + epsilon;
    norm *= masses[j];
    acc_x += norm * diff_x;
    acc_y += norm * diff_y;
    acc_z += norm * diff_z;
  }

  vel_x[id] += acc_x * dt;
  vel_y[id] += acc_y * dt;
  vel_z[id] += acc_z * dt;
}
```

Figure 27.: OpenCL kernels used for the N body code.

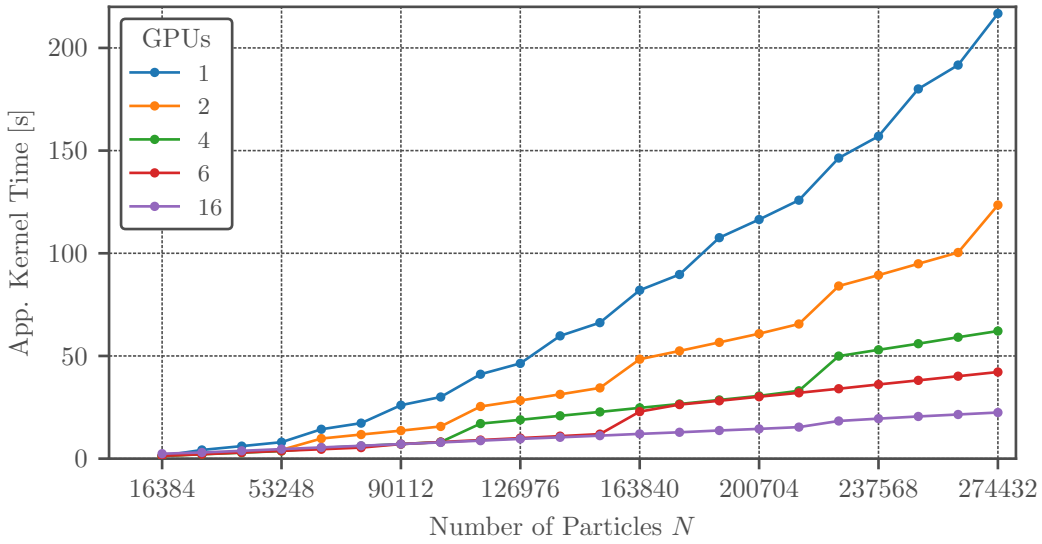


Figure 28.: Application kernel time for the N body code.

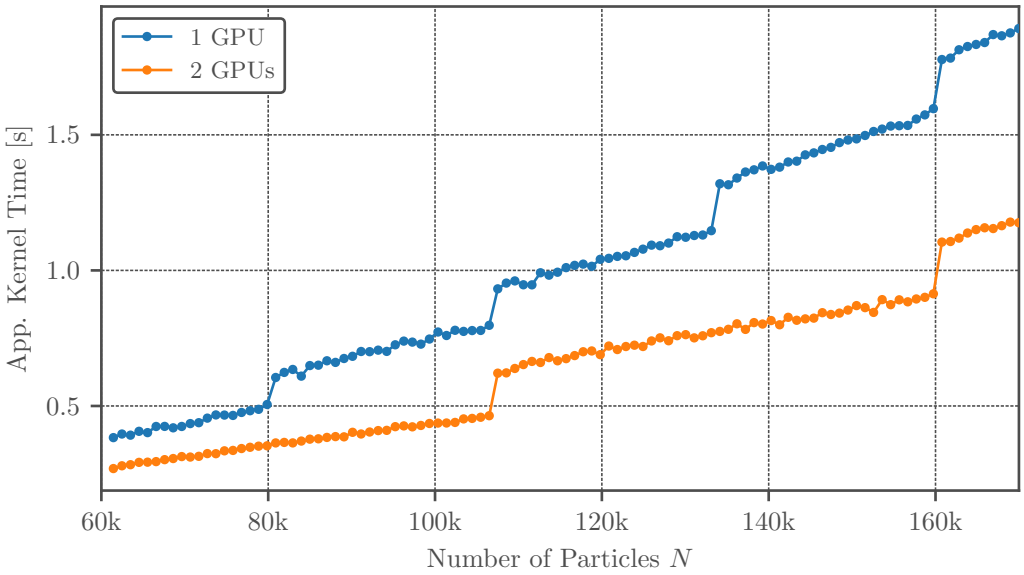


Figure 29.: Zoom of the application kernel time for the N body code. Here you see the calculation of one time step.

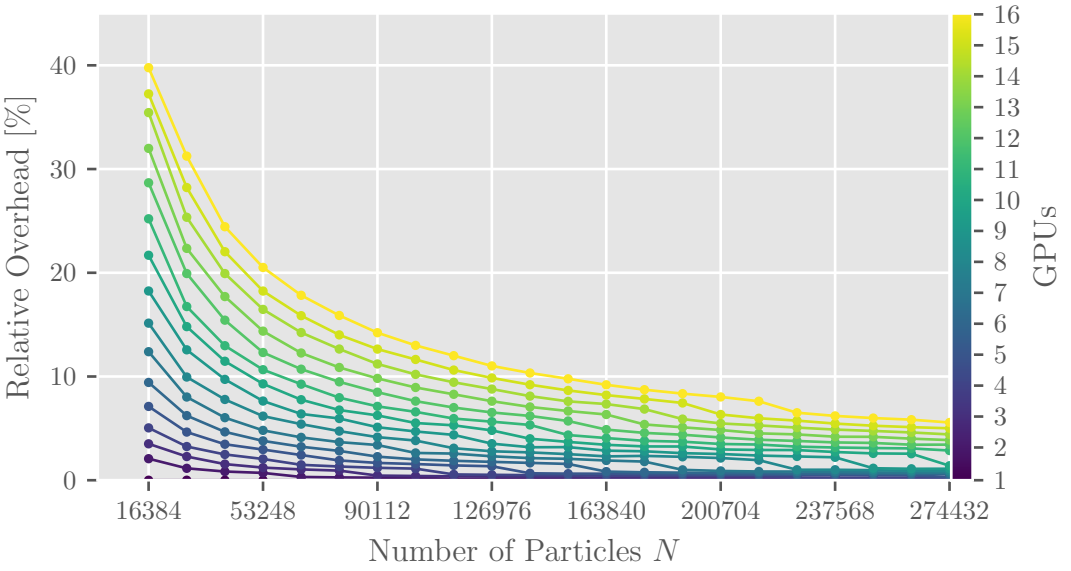


Figure 30.: Overhead of Mekong's runtime library relative to the total runtime. This includes communication, synchronization, and calculation due to inter kernel dependency resolution.

see the dependency resolution time caused by communication, synchronization, and calculations relative to the total application execution time in figure 30. You can think of the summation as the time, which has been artificially added by Mekong's runtime library due to the automatic parallelization of the application, and represents thus an overhead of our workflow. The line, which represents 16 GPUs, starts for small N at 40%, but decreases rapidly until it reaches a relative overhead of $\approx 6\%$. Moreover the vertical distance between two lines for different numbers of GPUs, seems to shrink with a decreasing number of GPUs, which meets our expectation, because an additional GPU must receive the data from every other GPU. The consequence for a trivial implemented all to all copy scheme is the location in $\mathcal{O}(\texttt{numGPUs}^2)$.

To compares the three test cases with each other, you can see the runtime breakdown for the N body Code in figure 31. Likewise to the matrix multiplication is anything else with exception of the kernel calculation time hardly visible. As well as the complexity of the matrix multiplication lives the complexity of the N body code also in $\mathcal{O}(N^3)$. For the same reasons the other timespans are negligible small in comparison to the cubic growing calculation time. Nevertheless the N body code differs clearly in its scalability in comparison to the matrix multiplication. You can see that there is no decrease in the total application time for 11 GPUs upwards in case of the large N, and even for 7 GPUs in case of the smaller N.

4.3.2. Conclusion

The N body code as the most complex test case, presented a good scalability *with respect* to the original single device code. In order to increase the scalability the

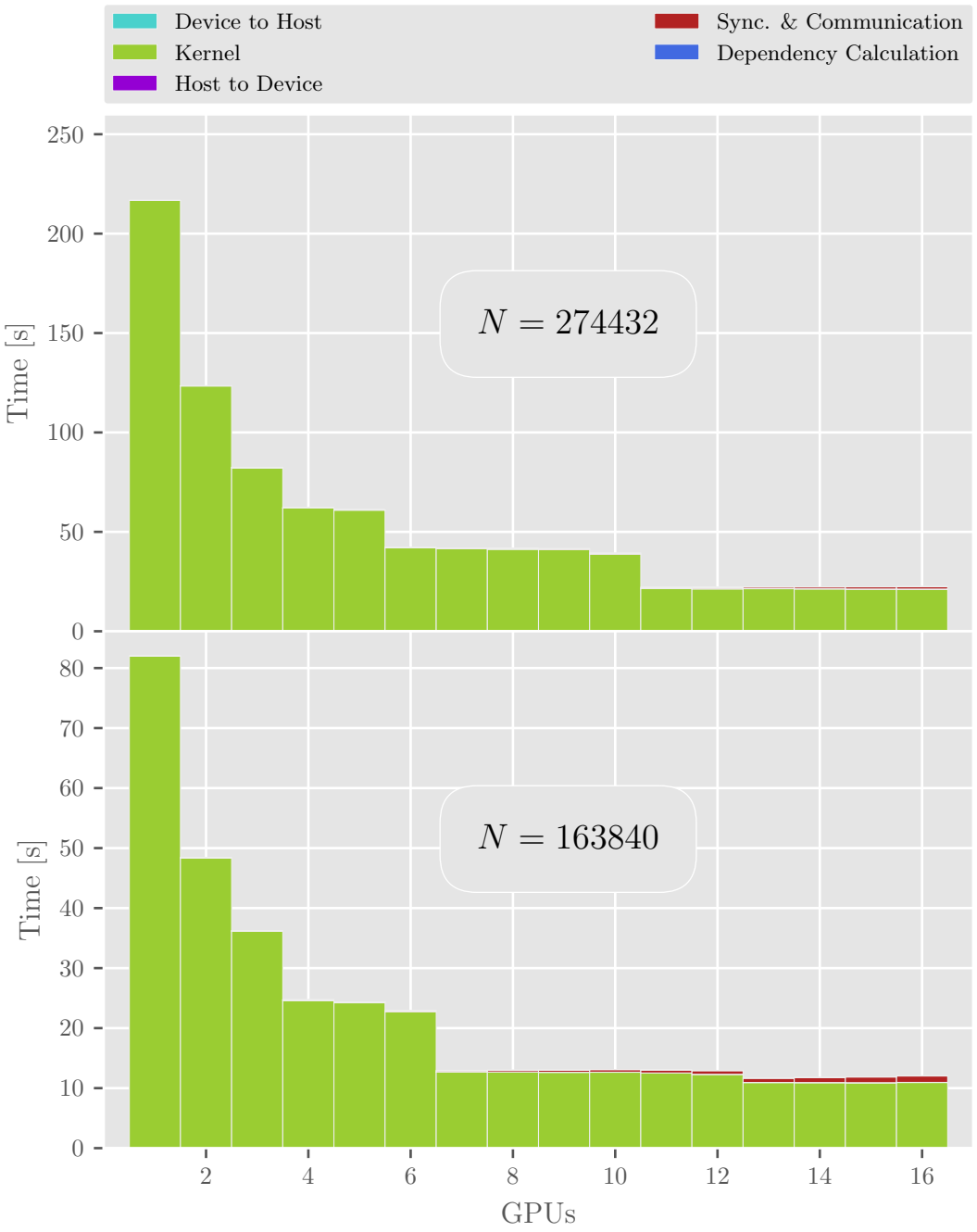


Figure 31.: Runtime breakdown for the N body code.

problem size N must be sufficiently large, to exploit all available computational power. The total runtime limits, which are visible in the graphs discussed in this section, emerge from the original device code, rather than from Mekong's runtime library.

5. Future Work

5.1. Cropping Code to Enable Polly's Analysis

Section 3.2 explained why Mekong's analysis part highly depends on the Polly library. If Polly's access pattern analysis fails, Mekong's complete workflow will fail too, or the user must write the access patterns himself. Most kernels contain memory accesses which will be analyzed by Polly, but are not relevant for Mekong's workflow. It can happen that Polly aborts its analysis, because of a non-affine shared memory access. Moreover it is not able to prioritize the analysis of a specific kind of memory accesses or to disable the analysis of others at all.

Writing a pass which removes all accesses on shared memory before analyzing the memory accesses on global memory can improve the overall probability of a successful access pattern analysis. But simply removing the LLVM IR instructions, which contain the accesses to shared memory, is in many cases not possible. Because of the SSA model, which LLVM IR obeys, any used value must be defined before it is used. Thus removing shared memory instructions removes all values which were calculated using shared memory. The only reason to use shared memory at all, is to use it temporary calculating a result value, and writing the result back into global memory. In that case the store instruction to global memory is dependent on the shared memory instruction in LLVM IR. As we definitely do not want to remove the global memory instructions to keep the SSA model, the approach must be slightly modified.

Instead of simply removing shared memory instructions, we can replace all loads from local memory with the LLVM value holding the thread ID of any dimension. It is beneficial to use the thread ID, instead of any constant value, because a store instruction to global memory of a value, which is constant at compile time, can be highly modified by any optimization passes. In the worst case this can modify the access pattern, which was used in the global memory instruction we wanted to analyze. Figure 32 illustrates the typical usage of shared memory inside a kernel. Usually, (1) some data is stored to shared memory, (2) data is loaded from shared memory, (3) calculations with the loaded data are made, (4) either results are written to global

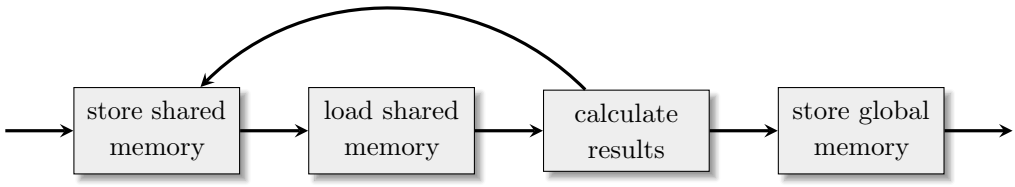


Figure 32.: Typical control flow within a part of a kernel deploying shared memory.

memory and next kernel part will be processed, or the chain will start again from step one. In consequence a replacement of the shared memory values with constant at compile time constant values, will state the calculation part as constant too. As the calculation part is constant the store to global memory will also be constant at compile time. Now any optimization pass, which is a requirement for Polly's analysis pass, is likely to change the IR, and thus the memory access patterns vigorously.

The replacement of values loaded from shared memory with the thread ID can simplify the IR to make Polly's analysis possible, but does not oversimplify the code, that the memory access patterns get destructed by optimization passes.

The pattern described above can be extended to replacing load instructions of global memory. Replacing a load instruction of a `kernel_array` automatically rises the assumption that every thread must load every element contained in that `kernel_array`. This assumption might increase the communication between GPUs, but the validity of results will not be broken. This procedure is not possible regarding store instructions on global memory.

5.2. Stalling Memory Copies

As as you could see in section 4.2, the major bottleneck for the scalability of the stencil code was represented by the initial host to device memory copy operation, which results in a broadcast in Mekong's runtime library. This section proposes three schemes (ordered by their priority) to avoid implementing the initial host to device copy as a broadcast. We will limit the presented model to a simple single kernel launch application.

Scheme A

In figure 33 you can see an example of a host code, which instructs a GPU. The boxes denote basic tasks, which have to be done in a GPU deploying CPU code. The major problem causing a broadcast, when the initial host to device memory copy operation is scheduled, is represented by the indefinite values of the kernel launch parameters at that moment. Thus the runtime library does not know yet, which kernel launch will be scheduled, and how its scalar kernel arguments and used buffers will look like. These are all mandatory values to calculate the read intervals of a kernel launch, therefore the first approach is to be aware of the needed values before the initial host to device memory copy is called. In order to achieve that, Mekong's host code transformation should change the position of the instructions in LLVM IR, which set arguments and grid size of the kernel launch. These instructions should be moved upwards as far as possible. If it is possible to move them in front of the data initialization on the host, we apply transformation scheme A, which is shown in blue in figure 33. As the data initialization process is the only time consuming task, which has to be done in front of the host to device memory copy, it is worthwhile to start an additional thread, which calculates the read intervals of the kernel launch. This can be achieved with further host code transformations, e.g. a simple function call insertion at the sketched position would be sufficient.

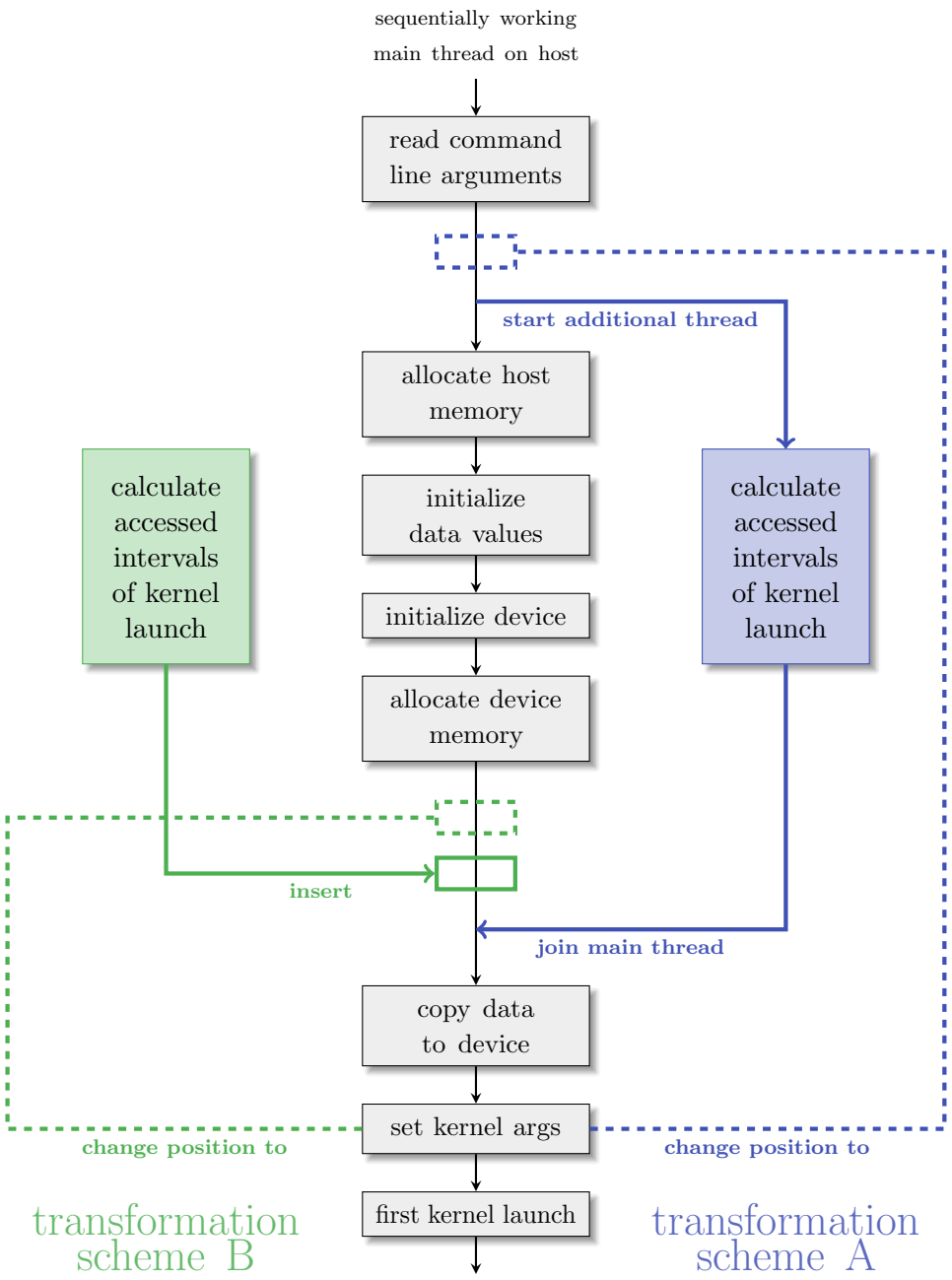


Figure 33.: *Middle:* example tasks for a host code which instructs a GPU. *Side:* transformation schemes to move the calculation up, which emits the accessed intervals of a kernel launch. This could prevent a host to device copy operation from resulting into broadcast in Mekong's runtime library.

It is important that the main thread synchronizes with the additional thread before it starts executing the initial host to device copy. In consequence the read intervals are known at the moment of the copy operation, and Mekong's runtime library is able to copy only the data, which is actually read by each GPU.

Nevertheless the above exemplified scheme might not always be possible. The next sub section will outline an alternative scheme.

Scheme B

If it is not possible to shift the setting of the kernel launch arguments and grid size as upwards as it is mandatory for scheme A, we apply an alternative scheme. This scheme requires the setting of the kernel arguments and grid size to be relocatable in front of the initial host to device memory copy. In that case Mekong's host code transformation can also insert a function call to calculate the read intervals of the kernel launch, after the arguments etc. has been set (fig. 33).

Scheme B increases the overhead, which is caused by Mekong's runtime library, in front of the initial host to device copy, but can overall reduce the overhead. A performance increase can be gained in that case if the calculation of the read intervals is less expensive than the replication of the data to all GPUs. In case of a stencil code there shall be an array width N, and a number of GPUs N_{GPU}, which cause an performance increase, if we apply transformation scheme B, because the calculation of read intervals has a complexity of $\mathcal{O}(N)$[1], whereas the initial host to device copy treated as a broadcast is located in $\mathcal{O}(N_{\text{GPU}} \cdot N^2)$.

Nevertheless both above exemplified schemes might not always be possible. The next sub section will outline an alternative scheme.

Scheme C

Scheme C, as the last alternative, does not exploit host code transformation at all, but includes static host code analysis. If the host code analysis can certainly determine that the host buffer, which should be copied to the device, to transfer the initial data, is not written between the initial host to device copy and the kernel launch, Mekong's runtime library is able to stall the initial host to device copy until kernel launch. At kernel launch the arguments and grid size are definitely available, thus the read intervals can be computed, and successively only read data, which is actually read by each GPU, can be copied. After that has been done, the kernel launch can be processed as usual.

The future work presented in this chapter represents an opportunity to deploy the host code analysis in a profitable way.

5.3. Array Reshaping

Mekong in its current state is limited to the memory available for one GPU. Although a buffer allocation in the application context results in an allocation on every GPU in

[1] This is basically caused by the linearization of the 2D access patterns within a stencil kernel.

Mekong's context, it can happen that most parts of a buffer are neither required nor used by a GPU. Allocating only the space in memory, which is actually needed by a GPU would further increase Mekong's usability. This requires changes in the host and device code at compile time, plus additional logic implemented in Mekong's runtime library. The changes which must be done in the host code are very similar to section 5.2, as the accessed intervals of a kernel launch on an array, must be known at the moment of buffer allocation. It might also be possible to design a hybrid solution of stalling some application functions[2] and to change positions of instructions at compile time.

Instructions located in the device code, which accesses arrays represented by device buffers, must be modified, because the size of the buffers has been changed. This includes the transformation of index calculations. In case of reshaping a buffer, which is only written by kernel launches, we basically must *invert* the access pattern on the belonging kernel array, which is possible, because a write access must always be at least injective. This inverted access pattern has to be inserted into LLVM IR as an additional device function and should be wrapped around every index calculation regarding the belonging kernel array.

5.4. Dependency Resolution Generalization

Up to now Mekong's inter kernel dependency treatment assumes any write on a device buffer caused by a kernel launch to invalidate all previous write accesses on that buffer. For various applications (matrix multiplication, stencil code, N body code, reduction) this is true, but e.g. a mesh refinement code can not be treated correctly by making this assumption. If you want to know why a mesh refinement code contradicts to that assumption, you might want to read section 3.4.3. This section proposes a new scheme to process inter kernel dependencies in a more general way.

Mekong's programming model simulates a one GPU environment to the programmer, consequently he handles only one device buffer, if he allocates one on the virtual device. Mekong's runtime library has the possibility to treat this virtual buffer in various ways. Up to now it gets allocated on every device. The first step to introduce our new dependency resolution model, is the definition of a virtual buffer write object W, which saves the last writing GPU for every element saved by a buffer.

Definition 1. *A **virtual buffer write object** $\omega($`devPtr`$)$ saves the last writing GPU for every element saved by a virtual buffer* `devPtr`*. The written points can be saved using* `numGPUs` *(multi-dimensional)* `isl_set` *objects.*

This implicitly assumes that a virtual buffer can only be used for kernel arrays with an equal element type. That means that the programmer is not allowed to use a buffer as a `float*` pointer and subsequently as a `int*` pointer in another kernel launch. Moreover the buffer must always represent a kernel array, which has the same dimension and size. Moreover we define:

[2] E.g. to delay the actual buffer allocation until the host to device memory copy is called in the application context

Definition 2. *A **kernel launch writing chain** χ(`devPtr`) saves the chronological order of kernel launches $A, B, C, ...$, which wrote all to the **same** virtual buffer `devPtr`. In fact the appropriate container for χ is a queue. If a kernel launch A is inserted in the front of χ, because he writes to `devPtr`, but the same[3] kernel launch A is already contained in the queue χ, the old A will be erased.*

Up to now Mekong's dependency resolving objects DR can uniquely be identified with two different kernel launches A & B. In case of a generalization we can identify a DR uniquely with a current kernel launch C, which will read buffer `devPtr`, and the current kernel launch writing chain χ(`devPtr`) beloning to that buffer. If a dependency resolution object for a buffer `devPtr`, χ, and a current kernel launch C has not been created yet, the appropriate virtual buffer write object ω(`devPtr`) has to be calculated of χ, because DR can be calculated of C's read access and the current ω regarding the same buffer. In fact that is possible, because every chronological order of kernel launches writing to the same buffer causes the shape of that virtual buffer. Therefore DR is only a function of $DR(\chi(\texttt{devPtr}), C)$. The great advantage of using that model based on χ is that we can delete old kernel launches in χ if we insert the same kernel launch in the front. That deletion has no effect on the shape of $\omega(\chi)$, which can be calculated of the current χ.

The above exemplified model is well suited for applications, which does not met Mekong's current assumption regarding the inter kernel dependency resolution, and schedule different kernel launches cyclically (e.g. $A, B, C, A, B, C, A, B, ...$). An integration of the above mentioned model into the existent hierarchical caching system is mandatory to keep the overhead induced by Mekong's runtime library low.

[3]Definition of equal kernel launches is located in section 3.4.3

6. Conclusion

This thesis presented a workflow, which analyzes and changes a single GPU program at compile time deploying polyhedral techniques, to partition the application automatically to multiple GPUs within one node. This includes index transformations in the device code and function replacements in the host code, which links the input code with a runtime library, which contains additional logic regarding the resolution of inter kernel dependencies. The code changes at compile time have been implemented in self-written custom LLVM passes, whereas the memory access pattern extraction and their management was based on the Polly & Integer Set Library. Deploying polyhedral compilation techniques to detect memory access patterns in kernels, and to use them for the result merging process & inter kernel dependency resolution, embodies the property, which makes this project unique in comparison to the presented related work.

The static analysis of OpenCL device code worked well for two of three test cases, due to issues of the Polly library with OpenCL code in LLVM intermediate representation. The other test cases have automatically been partitioned with the splitting dimension as sole user input.

The overhead induced by the self-written runtime library, which addresses the inter kernel dependency resolution, and the result merging process deploying the Integer Set Library, was with exception of the initial data distributing broadcast negligible small. The transformation of the initial host to device memory copy operation to a broadcast has shown significant scalability limitations for the stencil code. A formalism to avoid the initial broadcast was proposed in section 5.2. The total performance of the applications has been increased deploying 16 GPUs, and a speedup to single GPU execution of 14.9 for the matrix multiplication, 4.2 for the stencil code, and 9.9 for the N body code was achieved.

The implemented partitioning algorithm distributes the original grid into equally sized parts, which is an appropriate choice for nodes with equally performing GPUs. The used formalism for partitioning is generally suited for distributed memory systems. It would be possible to apply the concept of the workflow to a multi node CPU cluster as well as to its original purpose. Moreover it would be possible to adapt our concept to an OpenMP similar usage. If an iteration domain has been marked with a compilation Parallel-Pragma within the program code, one could handle the different iteration domains as kernel launches, which could be treated and partitioned in our workflow. Rather inappropriate systems for the presented concept are MICs or CPUs within one node, as they do not obey the BSP model and have a shared memory system.

6.1. Comment

To my mind compilation techniques supporting the growing fraction of parallelism implemented in hardware are inevitable to ensure a reasonable usability by the programmer. Up to now it seems as if the hardware parallelism grew to fast for the compilation techniques to keep up in the recent years. This can be seen as a semantic gap, which is currently growing between programmer and architecture. Moreover many programmers think of a compiler as a black box, which is out of their bounds. Mekong could generally represent a communicator between scientific programmer and compiler; and is meant to close the semantic gap. For example it could be of interest to estimate the results of a simulation in dependency of statistically omitted synchronization, which should be chosen by the programmer.

Nomenclature

device memory	memory of *one* GPU. NVIDIA's K80 has two GPUs thus two RAM, but is one device
host memory	memory of the CPU
`kernel_array`	An array used in a kernel function; Can have multiple dimensions and a size. A buffer might be a different `kernel_array` in a subsequent kernel launch.
sub block ID	a block ID in the coordinates of a sub grid
sub global thread ID	a global thread ID in the coordinate system of a sub grid
sub grid	the grid launched per partition on one GPU
super block ID	a block ID in the coordinates of the super grid
super global thread ID	a global thread ID in the coordinate system of a super grid
super grid	the original grid defined in the unmodified input code at kernel launch
`nvcc`	NVIDIA's CUDA compiler for host and device code[19]
BSP	bulk-synchronous parallel model[25]
CPU	central processing unit
FLOP	floating point operation
GPU	graphical processing unit
ISL	Integer Set Library[26]; Used to represent memory access patterns
islpy	python wrapper for the ISL
kernel	a function which is usually written for a GPU. With specific declarations an OpenCL or CUDA kernel can also be executed by a CPU.
LLVM	Low Level Virtual Machine; Sometimes it is a shortcut for 'The LLVM Compiler Infrastructure'[4]

Polly	compiler based polyhedral optimization library, which can analyze memory access patterns[5]
PTX	means Parallel Thread eXecution which is a pseudo assembly language. It is used by NVIDIA's CUDA compiler `nvcc`.
SSA	Static Single Assignment model[24]; Is obeyed by LLVM IR

A. Bibliography

[1] *OpenMP Application Programming Interface*, July 2013, Version 4.0.

[2] *The OpenACC Application Programming Interface*, October 2015, Version 2.5.

[3] *clang: a c language family frontend for llvm*, 2016, http://clang.llvm.org/.

[4] *The llvm compiler infrastructure*, October 2016, http://llvm.org/.

[5] Raghesh Aloor, Johannes Doerfert, Tobias Grosser, and Andreas Simbürger, *Polly, llvm framework for high-level loop and data-locality optimizations*, October 2016, http://polly.llvm.org.

[6] Cilardo, Alessandro, Gallo, and Luca, *Improving multibank memory access parallelism with lattice-based partitioning*, ACM Transactions on Architecture and Code Optimization (TACO) **11** (2015), no. 4, 45.

[7] M. Flynn, *Some computer organizations and their effectiveness*, IEEE Trans. Comput. **C-21** (1972), 948–960.

[8] Khronos Group, *OpenCL*, 2016, www.khronos.org/OpenCL.

[9] Open Group, *Pthreads*, 2016, http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html.

[10] Mark Harris, *Unified memory in cuda 6*, November 2013, https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6.

[11] ECM International, *The json data interchange format*, October 2013, Standard ECMA-404.

[12] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and et al., *Shoal: smart allocation and replication of memory parallel programs*, USENIC ATC'15, July 2015.

[13] Jongwon Kim, Honggyu Kim, Joo Hwan Lee, , and Jaejin Lee, *Achieving a single compute device image in opencl for multiple gpus*, PPoPP'11, Februrary 2011.

[14] Andreas Kloeckner, *islpy*, 2016, https://github.com/inducer/islpy.

[15] Lee, Janghaeng, Samadi, Mehrzad, Mahlke, and Scott, *Orchestrating multiple data-parallel kernels on multiple devices*, International Conference on Parallel Architectures and Compilation Techniques (PACT), vol. 24, 2015.

[16] Lee, Janghaeng, Samadi, Mehrzad, Park, Yongjun, Mahlke, and Scott, *Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration*, ACM Transactions on Computer Systems (TOCS) **33** (2015), no. 3, 9.

[17] Gordon Earle Moore, *Cramming more components onto integrated curcuits*, Electronics **38** (1965).

[18] NVIDIA, *Cuda*, 2016, https://developer.nvidia.com/cuda-zone.

[19] ———, *nvcc*, 2016, http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[20] ———, *Parallel thread execution instruction set architecture (ptx isa)*, 2016, http://docs.nvidia.com/cuda/parallel-thread-execution.

[21] Prasanna Pandit and R. Govindarajan, *Fluidc kernels: Cooperative execution of opencl programs on multiple heterogeneous devices*, CGO'14, Februrary 2014.

[22] Mahesh Ravinshankar, Roshan Dathathri, Venmugil Elango, and et al., *Distributed memory code generation for mixed irregular/regular compuations*, PPoPP'15, Februrary 2015.

[23] Sebastian Schaetz and Martin Uecker, *A multi-gpu programming library for real-time applications*, January 2013.

[24] J. Singer, P. Brisk, and F. Rastello et al., *Static single assignment book*, Springer, 2015.

[25] Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM **33** (1990), no. 8, 103–111.

[26] Sven Verdoolaege, *Integer set library*, 2016, http://repo.or.cz/w/isl.git.

[27] ———, *Integer set library: Manual*, 2016, Version 0.16.

[28] John von Neumann, *First Draft of a Report on the EDVAC*, University of Pennsylvania, 1945.

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 24. Oktober 2016

Christoph Julian Klein