Task and Data Distribution in Hybrid Parallel Systems

Frank Feinbube
Frank.Feinbube@hpi.uni-potsdam.de

This paper describes my work with the Operating Systems and Middleware group for the HPI Research School on "Service-Oriented Systems Engineering".

Computer architecture is shifting. The upper levels of the software stack are thus to be adapted in order to benefit from the current and future hardware capabilities. In this paper, we present the Hybrid.Parallel library. It is our approach to bridge the gap between state-of-the-art computer architecture and application developers (in contrast to performance engineers, tuning experts, ...) In order to exploit the full computing power of a system, all execution devices have to be identified and used. Tasks and data have to be distributed according to the specific features of the devices and the overall system. We present our analysis of tasks and data in the shared memory parallel domain and propose mappings onto execution units and memory types. Furthermore we discuss language constructs that allow developers to adjust these mappings to their needs. These findings provide the basis for the implementation of the scheduler for our Hybrid.Parallel library and may also be applied to the Single-Chip-Cloud-Computer.

1 Commercial Off-The-Shelf Computers are Hybrid

Todays computers are highly sophisticated and astonishingly complex systems. Only due to the devision into and the cooperation of various system levels is it possible that they have become such a present and useful part of our everyday life. There is the hardware level, hypervisor level, operating systems, middleware, libraries, tools and applications. These system levels were created by a vast number of specialists and cooperate via reliable API contracts that hide away the complexity of underlying levels. This separation of concerns allows development and optimization of parts of the hierarchy without influencing the others. This holds as long as there is no crucial shift in the underlying levels that results in changes of the API protocols. Such a shift is currently occurring in the lowest level: the computer architecture. Due to power density problems and memory channel speed restrictions, computer architectures changing. The dream of a single super-CPU and a single enormous chunk of shared memory is over. Systems consist of a large number of inhomogeneous execution units and have a huge amount of distributed memory. Execution units have different clock frequencies, different memory bandwidths, and even different capabilities and precision in their calculations. Memory is distributed in a memory hierarchy and accessing it results in differing latencies. Memory may even be read-only, non-cached or incoherent.

While conventional processors can still be regarded as homogeneous, they are not. Intel's Sandy Bridge architecture and AMD's Accelerated Processing Unit (APU)

technology vary processor clock frequencies at runtime, are NUMA-enabled, and incorporate a GPU compute devices in the system processor. GPU compute devices are special accelerators that execute data parallel algorithms in a SIMD fashion. To get the most out of these systems it is required to use specialized vendors-specific low-level APIs and have a deep understanding of the hardware characteristics. [1, 3, 6]

2 Bridging the Gap

In contrast to performance engineers and tuning specialists, many application developers can not make the effort to learn the details of vendor-specific and version-specific device characteristics and specialized APIs. Therefore we aim to encapsulate as much existing knowledge as possible into an easy-to-use library that supports these developers: Our *Hybrid.Parallel* library. Besides supporting the creation of new programs, it also allows refactoring existing code with minimal of code changes.

2.1 Related Work

Aparapi [2] is a library that allows writing Java code for AMD GPUs. Such code needs to inherit from the provided Kernel class and overwrite its Run method. If the kernel is invoked, it will run on the GPU if both a supported graphics card is installed and all features used by the program can be mapped to the GPU. Otherwise, it executes on the CPU instead. Aparapi supports only primitive Java types. (*double* and *char* is not supported). Arrays of primitive types are supported as long as they have only one dimension. They also cannot be used as arguments to calls or aliased within the same method. Static fields of objects cannot be used unless their value is known at compile time; instance fields can be read, not written. Creating new objects (including arrays) is not supported, neither are exception handling nor control flow statements like break, continue and switch.

GPU.NET [5] consists of a runtime library and post-processor that works on .NET assemblies. To execute code on a GPU, it needs to be within a static method and annotated with the Kernel attribute; such methods are extracted from compiled .NET binaries and processed during an additional compilation step. For buffers like arrays special annotation can be used to specify the memory space they should be placed in by the runtime. GPU.NET generates neither CUDA nor OpenCL code, but compiles directly to device-specific binary code. If no supported graphics card is installed, the code is executed on the CPU instead.

Both libraries mentioned follow a fundamentally different approach: they do not create OpenCL code, but GPU vendor specific binaries. Because they are tailored for specific vendor hardware, they may exhibit better performance on a subset of accelerators, but are also restricted to specific devices. They also do not hide the OpenCL API, so programmers still need to understand the notion of a kernel, to adjust grid sizes, and calculate thread and block indexes. This leads to code bloat and makes them less suitable for refactoring.

2.2 Our Approach

Our Hybrid.Parallel library (Figure 1) allows automatic transformation of .NET bytecode (i.e.; C#) for (parallel) execution on OpenCL-accelerators. New language constructs, such as a *Hybrid.Parallel.For* can be used from .NET. Resulting code can either be executed on a (multi-core) CPU or on an OpenCL-enabled accelerator. At the heart of our approach is a decompiler that transforms .NET bytecode into accelerator-neutral source code. This code can then be recompiled for specific accelerator types.

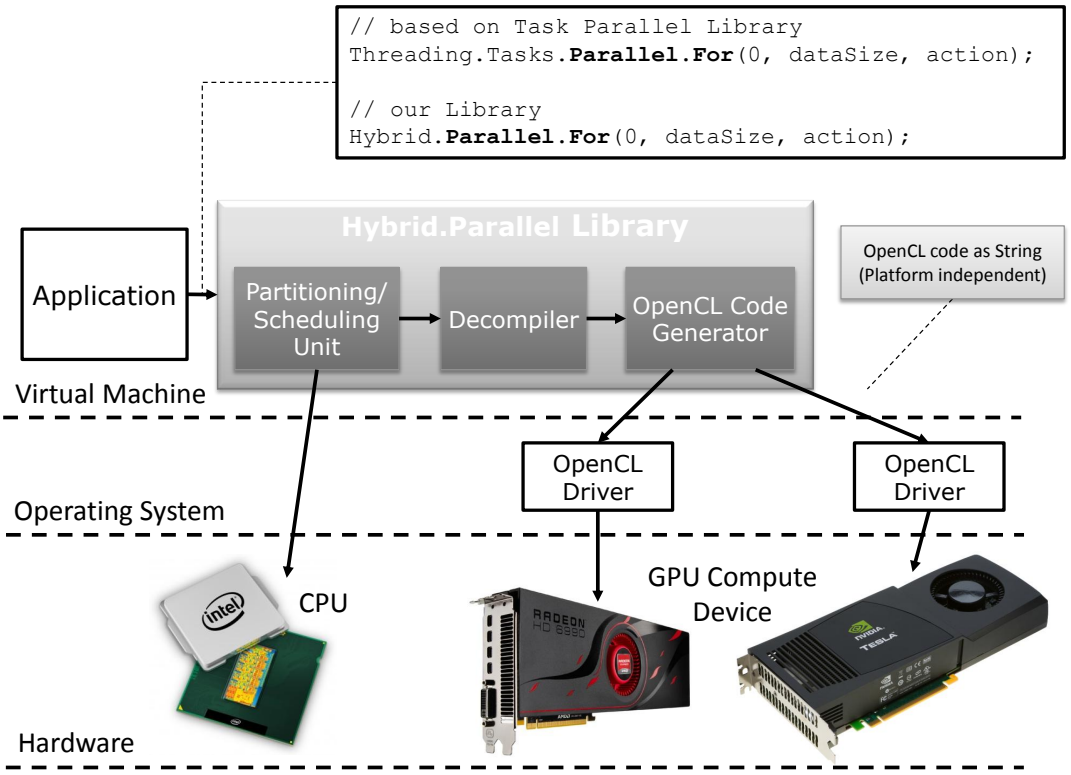

Figure 1: Architecture Overview

Figure 1 also shows how our library presents itself to developers. In order to make use of it, it is sufficient to use one static method that looks exactly like Microsoft's equivalent: *Hybrid.Parallel.For*. It will accept the same parameters as Microsoft's version, but allows transparent scheduling on CPU, GPU and other accelerators. Thus, refactoring an existing application for hybrid computing could be done by simply exchanging the namespace of *Parallel.For*. In addition to the one-dimensional construct, we provide a two dimensional version that can substitute two *for*-loops and allows a better mapping onto GPU hardware.

Most of the projects mentioned in section 2.1 require developers to modify a lot of code in order to fit to the model of the underlying systems. While that may be reasonable for developers who can afford the time to dig deep into a new programming model, it is a lot effort for programmers that already have another focus for their work, f.e. writing business logic. We provide a solution that requires a minimum of code

changes. In addition to this, we also want to support capabilities for fine tuning. The next sections provide further details about this.

3 Distributing Data

The requirement for parallel execution is the distribution of tasks and data on the resources available. In particular, the distribution of the data has a strong impact on performance. In this section we present some access patterns, and discuss their mapping to memory types. In addition, we suggest key words that allow developers to exert influence on these mappings.

3.1 Memory Hierarchy of Compute Devices

Compute Devices have a complex memory hierarchy. [1] Data locality, sizes and latencies are important for data distribution. The following list gives an overview of the memory types and their relevant characteristics:

- **Global Memory**: shared by all work items; read/write; may be cached (modern GPU), otherwise slow; huge
- **Private Memory**: for local variables; per work item; may be mapped onto global memory (arrays on GPU Compute Device)
- **Local Memory**: shared between work items of a work group; may be mapped onto global memory, otherwise fast; small
- **Constant Memory**: read-only, cached; additional special kind for GPUs: texture memory (2-dimensional cache)
- **Host Memory**: main memory of the host system

3.2 Data Access Patterns

In order to enable our runtime to distribute the input data on the execution units in a way that allows for efficient execution, it is crucial to know how data is accessed. There are various ways, data parallel execution can make use of its memory. Figure 2 provides an overview of common access patterns. The matrix represents in-memory data. Yellow indicates a read operation on the input data array. Red indicates a write operation to the output data array.

In simplest cases, accesses are restricted to a single cell of the input data array and produce a single entry in the output array. More complex cases also involve accesses to neighboring cells in a 1-, 2- or 3-dimensional fashion. Special attention has to be paid to the borders. Neighbors exceeding the borders may be ignored, default values may be used, or the access can be made in a ring fashion. In some cases, algorithms access only every n-th memory cell. This is called strided access.

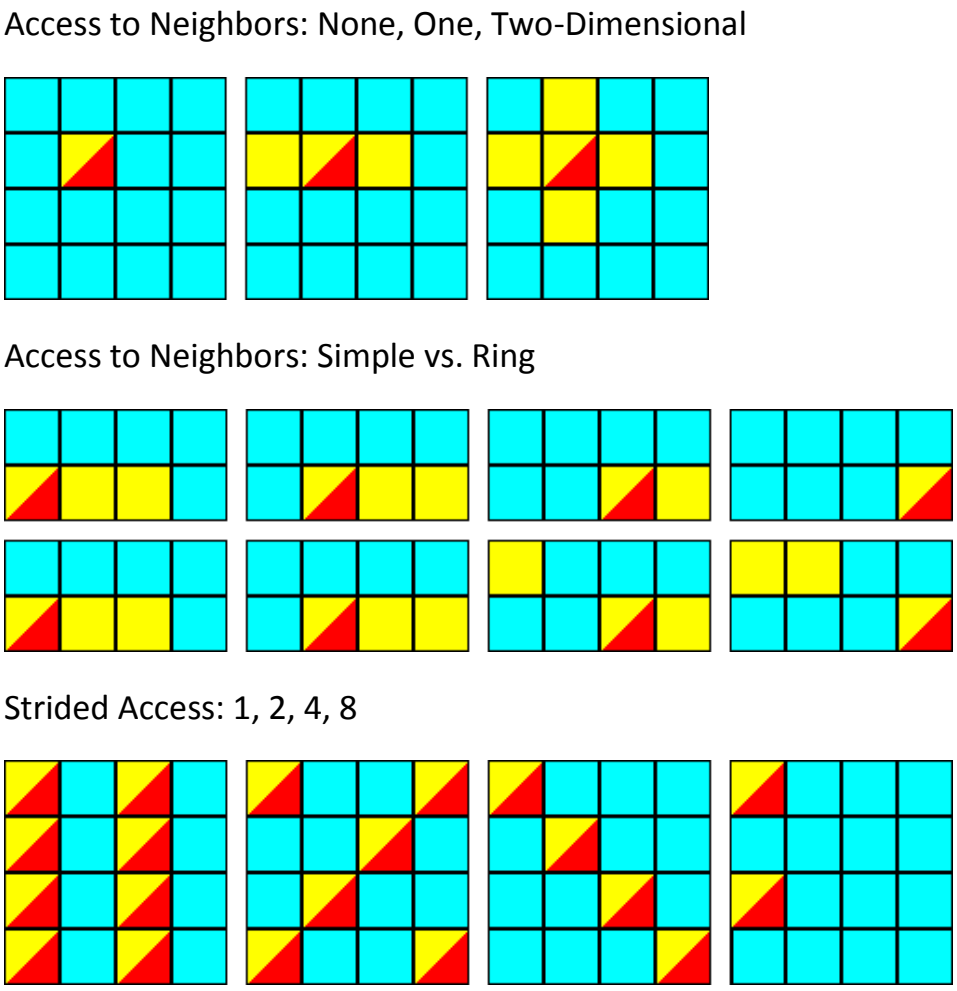

Figure 2: Data Access Patterns

3.3 Language Constructs for Data Access

Based on these patterns we propose a number of language constructs, to describe data access:

- **Pattern**: linear, complex, arbitrary
- **Access**: read only, write only, read write
- **Frequency**: never, once, exactly N times, seldom, frequent
- **Stride**: vector of non-negative integers; default 1
- **Neighbors**: vector of non-negative integers; default 0
- **OutOfBoundsValues**: ignore, ring fashion, next dimension, repeat last, fix value

Algorithms with linear access *pattern*s promise - depending on the other parameters - the biggest performance gains by optimizing data distribution, because only parts of the overall data have to be provided to the executive units. In contrast, arbitrary access can be supported only if all data are provided. Complex access patterns are in between the two extremes. To make these possible, developers must define a function to identify the required memory areas depending on the given thread id. *Access* marks in-/output variables. Read-only accesses could be cached in the constant memory to allow faster access. Since this memory is very small, it is important to identify, which memory should go there. *Frequency* gives a hint about the anticipated number of accesses per variable. *Neighbors* describes how many consecutive values are accessed per kernel execution. We have to distinguish between consecutive access to neighbors, access to left/top/right/bottom neighbors (as in a two dimensional layout), and more dimensional layouts. *OutOfBoundsValues* describes how to deal with accesses at the edges of the data space. Does the execution environment have to create additional memory cells and fill them with default values? Are values from the next row or the beginning of the current row to be copied in order to make sure the algorithm perform correctly? (The last example would be the case in the use of modulo operations.)

Using these key words, it is possible to describe memory access pattern in a way that allows the scheduler, to copy only data that are needed for local execution on the execution units. They also allow further optimizations. (see section 3.4) They represent only a first step. Much of the information can theoretically be derived automatically from the source code. This can be done by exploiting the reflection properties of .NET, for example.

3.4 Mapping onto Execution Units and Memory Types

For certain access patterns, the use of special types of memory is more advantageous than the use of others. [**?**] The objective for fast execution is to copy as little data as possible and to store it close to the execution units.

The most trivial case are data cells that are not used. These do therefore not need to be copied. This applies also to the cells, which are skipped by striding. Cells that are

only accessed once should remain in the memory of the host system and be used via PCI Express. Rare to normal access frequencies suggest a mapping onto the global memory, more frequent ones a mapping onto local memory. For very frequent accesses to neighboring cells, it makes sense to move the data into the constant cache. If these accesses to neighboring cells are within the two-dimensional space, data should be kept in the texture cache. Texture cache can also be applied for arbitrary access patterns.

The alignment of the data in memory must also be considered for the best possible access performance. This is particularly important to reduce memory bank conflicts and benefit from caching. Compaction could be applied if strong striding is used. Furthermore, additional data has to be copied when neighboring cells are accessed and out of border accesses are to be supported.

If the memory layout is changed due to these optimizations, the indexes for data access in the source code of the kernel are to be adapted accordingly. This is for some cases far from trivial. Therefore, it should be carefully considered whether an overhead for copying additional memory to maintain a simplified access pattern is tolerable. Even in these cases, the size of the copied memory area can usually be reduced drastically compared to the trivial solution.

3.5 Survey on Data Access Patterns

Our analysis of data access patterns for programs from the parallel computing domain can be seen in table 1.

Example	Pattern	Access	Frequency	Stride	Neighbors	Borders
Dot Product	linear	read only, write only	once	huge	many	ignore
Average	linear	read only, write only	exactly 3 times, once	0	2 (1D)	ring fashion
Summing Vectors	linear	read only, write only	once	0	0	ignore
Ripple	linear	write only	never, once	0	0	ignore
Heat Transfer	linear	read only, write only	exactly 5 times	0	4 (2D)	ignore
Histogram	linear, arbitrary	read only, read write	once, frequent	0	0	ignore
Convolution	linear	read only, write only	exactly 5 times	0	4 (2D)	fix value (5.0)

Table 1: Data Access Survey

Although these algorithms are relatively simple, they are good representatives of the parallel algorithms domain. In particular, they belong to the class of algorithms, which benefit greatly from SIMD architectures. These algorithms point out fundamental requirements for data distribution. If we manage to distribute and to accelerate them

appropriately, we pave the way forward to more sophisticated problems. For example, the *NQueens Problem* [4], the *Sudoku Validator* [**?**], or the exercises from our lecture Parallel Programming Concepts: *Game of Life* and *Crypt* that have already been ported to GPU compute devices using low-level vendor APIs.

4 Distributing Tasks

For the distribution of tasks within hybrid systems, the properties of the execution units are to be considered. Even different versions of similar execution devices can mean dramatic differences in the execution behavior. [4]

To predict the performance of an algorithm on an execution unit, it is unfortunately not enough to look at its clock frequency. Firstly, some units are generally not able to execute certain processor command words. Some GPU Compute Devices do not support atomics and double precision floating point operations, others do. Secondly, their execution performance can vary. There are different classes of execution characteristics:

- **fix**: Certain instructions have a fixed execution times in processor cycles. The expected performance is thus derived directly from the instructions used in the algorithm and the level of parallelization for a given processing element. Example: The NVIDIA F100 series can perform exactly 8 single precision floating point instructions per cycle on a processing element.

- **dynamic, predictable**: The execution behavior of a processing element also depends on its load and use. The number and grouping of the execution threads can be adjusted at run time. It affects synchronization overhead, memory coalescing and occurrence of memory bank conflicts. Thus it has considerable influence on the resulting execution times.

- **dynamic, potentially predictable**: The influence of caching is very difficult to predict.

- **dynamic, not predictable**: The overhead of error correction and dynamic frequency scaling depends on external factors such as temperature and can therefore not be predicted at all.

To ensure optimum performance, a static performance assessment should be taken based on the fixed and predictable execution characteristics that is then adjusted dynamically at run time. Therefore it is necessary to capture the execution behavior and evaluate it at runtime. As our library controls the distribution of work packages, this can be realized non-inversive and with good performance. Based on this knowledge it is then trivial to distribute the work packages suitable.

5 Conclusion and Outlook

The analysis of the performance characteristics of algorithms of parallel domain, particularly with respect to data access patterns, provides us with an important basis for the realization of an optimized scheduler. The required parameters can be determined from the code and fine tuned by key words. This work will be implemented in the next step in our Hybrid.Parallel library for GPU-accelerated systems. In addition, it also provides the basis for the optimized design and porting of algorithms for the single-chip cloud computer. Based on the portfolio of parallel algorithms that were created for the Hybrid.Parallel project, we are able to evaluate the actual performance of our scheduler in practice.

References

[1] The OpenCL Specification - Version 1.1, 6 2010.

[2] Advanced Micro Devices, Inc. Aparapi. `http://developer.amd.com/zones/java/Pages/aparapi.aspx`.

[3] AMD. ATI Stream Software Development Kit (SDK). `http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx`.

[4] Frank Feinbube, Bernhard Rabe, Martin von Löwis, and Andreas Polze. NQueens on CUDA: Optimization Issues. In *Proceedings of 9th International Symposium on Parallel and Distributed Computing, Istanbul, Turkey (ISPDC), 2010*, 2010. Istanbul, Turkey (to appear).

[5] TidePowerd Ltd. GPU.NET. `http://www.tidepowerd.com/`, 2011.

[6] NVIDIA. CUDA Developer Zone. `http://developer.nvidia.com/category/zone/cuda-zone`, 2011.