Functional programming for nested data parallelism on GPUs

Antti Halme
Aalto University

Abstract—Recent advances in general purpose GPU computing technology allow new data parallel kernel jobs to be dispatched dynamically during kernel execution. This enables significantly more expressive programming using nested data parallelism (NDP), where the restrictive need for flat data structures and computation has been lifted.

Functional programming is fundamentally well suited for expressing data parallel computation. Expensive flattening and fusion transformations have so far been necessary in vectorizing irregular data structures and computation, causing functional NDP to be inefficient. With hardware supported nesting on massively parallel GPUs, high-abstraction functional nested data parallelism can now be explored further.

This paper introduces nested data parallelism and briefly makes a case for using functional languages to program GPUs for general purpose data parallel computing.

I. INTRODUCTION

Parallel processing capability is readily available in practically all modern computing platforms, but parallel programming is still underrepresented in software today. So far, the greatest success in the field has been within the data parallel programming model, where SIMD (Single instruction on multiple data) hardware and similarly built distributed systems do independent batch processing jobs at scale.

In recent years Graphics Processing Units (GPUs) have found new use in supporting general purpose parallel computing. Initially requiring a deep understanding of the graphics rendering pipeline, programming these “desktop supercomputers” has since gotten easier, but is still a major undertaking for the average developer. Today, GPUs are usually programmed using languages extended from C, where low-level hardware details and programming primitives empower the proficient programmer and deter the novice.

SIMD hardware enforces flat parallelism where data is regular and fixed, and operations are plain and orderly. While this flatness is a reasonable price to pay for parallel processing power, many general computing ideas fit this scheme poorly. Common algorithms, for example in scientific computing, must undergo a substantial transformation in order to comply with the flat parallel model.

New hardware architectures, namely NVIDIA Kepler GPUs [1], address this issue by supporting the far more expressive form of nested data parallelism (NDP), where irregular data structures and dynamic execution is possible. NDP provides a new view to parallelism, making parallel programs easier to write.

Coincidentally, the principled and powerful functional programming ideas from decades past have been rediscovered in contemporary programming languages. Well composing, side-effect free functional computations are inherently parallel and an excellent fit to the new nested data parallelism on the horizon. This paper explores this intersection of hardware supported nested data parallelism and functional programming.

Section II presents nested data parallelism and some implementation approaches. Section III argues in favor using functional languages in writing data parallel programs. Section IV discusses the effect of hardware supported NDP on parallel programming. The final section gives pointers to related work and offers some concluding thoughts.

II. NESTED DATA PARALLELISM

The two variants of data parallelism are flat data parallelism, where data is fixed and regular, and nested data parallelism, where data can be more irregular and its processing can be more dynamic and adaptive. Listing 1 illustrates the difference between the two. The example is from Jones [2].

Successful data parallelism builds on efficient data access patterns and well balanced work distribution. The “chunking”, even work distribution, available in flat data parallel programming, points to a simple, clean cost model, where the programming constructs map well into the available SIMD hardware. Here, the main challenge is in choosing the right granularity for work parallelization, so that data, the algorithm and targeted parallel hardware are as compatible as possible.

Nested data parallelism extends the notion of flat data parallelism. With NDP, some of the conceptual clarity is lost, as irregularities invalidate the idea of simple work distribution. NDP instead empowers the programmer in a different way, by allowing more expressive parallel programs to be written, NDP pushes the complexity down in compilers and hardware.

Listing 1: Flat and nested data parallelism

```
// flat data parallelism
for i = 1 to n
    for j = 1 to m
        compute(A, B, C, i, j)

// nested data parallelism
for i = 1 to n
    for j = 1 to x[i]
        compute(A, B, C, i, j)
```
NDP enables techniques like recursion and divide & conquer, as well as irregular data structures like trees and sparse matrices, very common in scientific applications and real world data.

Nesting computations result in irregular and fine-grained execution trees. Figure 1 visualizes various approaches to dealing with the nested computation presented in Listing 1.

A. Oversubscription

Oversubscription, Figure 1a, refers to allocating resources to all parallel workers based on the one that needs the most. Each of the $N$ initial workers launch $\max(x[i])$ new jobs. Straightforward to implement, this leads to inefficient parallel programs, where a great number of threads get wasted. In GPUs, this is equivalent to large scale branch divergence.

Oversubscription means that programmers knowingly write inefficient programs or the execution platform is relied upon to perform complex shape analysis. In the latter case, the overhead could well defeat the gains from the parallelism.

B. Serialization

Serialization, Figure 1b, refers to giving up parallelization in favor of sequential evaluation. In the example, $N$ initial workers do all of the inner loop themselves. This kind of outer-loop parallelism is also straightforward to do and has been standard practice for decades.

The clear downside is of course that with ill-balanced initial workloads, there is little parallelism overall. Many bulk parallelization mechanism are based on this concept, most notably the parallel pragmas in OpenMP.

C. Flattening

A step up from mere outer loop parallelization is the notion of flattened – or vectorized – parallelism, Figure 1c, where irregularities are evened out and work is only divided to workers after the whole task has been transformed into a flatly parallel one. After flattening, the new vectorized workload is again easy to chunk.

Automating this kind of a flattening transformation was first shown to be feasible by Blelloch [3] in the programming language NESL. Efficient flattening [4]–[7] requires careful design on the intermediate data structures and aggressive operator fusion to minimize the need for them. Fusion also improves the locality of reference and eliminates redundant synchronization points. Flattening depends on extensive bookkeeping to keep track of how the work gets divided.

D. Dynamic parallelism

Dynamic parallelism [1], [2], Figure 1d, happens when workers generate more jobs dynamically based on their designated workload. For the running example it means that the $N$ initial workers generate variable sized sub-work units to exactly cover their own need. It is then the task of a job scheduler to issue these jobs on the parallel hardware as it becomes available.

Parallel jobs with a fixed worker thread count lead to a work stealing design, where idling workers are assigned new work from those that have plenty, requiring careful coordination. Granularity is a critical issue, as choosing the level at which a worker should just finish the remaining job instead of branching it out further requires either runtime program analysis or hints from the programmer.

With hardware supported dynamic parallelism, runtime mechanisms play a central role. These can be mostly on the software side as work stealing constructs and job shuffling, but performance is gained when it is the hardware that manages the threads [8].

The main challenge with hardware supported nesting is making sure that the hardware can do all of the things that are outsourced to it. Scheduling and workload balancing must be very efficient despite the presence of thousands of worker threads. Memory management can easily defeat parallelism gains if implemented poorly. If thread communication is allowed, the necessary synchronization can also bring the overall performance down.
III. FUNCTIONAL DATA PARALLEL PROGRAMMING

Many programming languages support some kind of parallelism with the help of a dedicatedlibrary, but languages that support parallelism natively leave these far behind. Peyton Jones et al. [6] argue that there is a fundamental difference in starting with a by-default sequential language and working towards parallelism, as opposed to starting with a pure, functional language that is by-default parallel.

Pure languages require side effects to be explicitly requested, so they are highly suitable for expressing data parallelism, where computations must be free of global effects. The absence of side effects gives rise to referential transparency, which helps programmers reason about program behavior.

Lazy evaluation is beneficial for a parallel programming mindset as it rejects the luxurious notion of predictable execution order. Persistence, disallowing in-place data structure mutation, helps in reasoning about memory usage. Data parallel programming idioms are common functional combinators, such as map, zipWith and fold, that operate on data structures themselves. Collective operations based on combinators map directly to parallel job synchronization needs.

Functional languages support function composition and higher order programming constructs. These enable clean system layouts and lead to powerful abstractions, such as skeleton designs [5], [9], which encapsulate common parallel programming patterns. Higher level programming is also back-end independent.

Totoo et al. [9] discuss the parallelism support in the contemporary programming languages Haskell, F# and Scala, and give pointers to related work in the field. Chakravarty at al. [5] and Bergstrom and Reppy [4] give further pointers to recent research effort in functional data parallel programming.

IV. HARDWARE SUPPORT FOR NESTING

General purpose GPU computation (GPGPU) has transformed massively parallel local computing. GPGPU is based on using the graphics output optimized GPU hardware to evaluate general data parallel functions locally or in clusters. OpenCL is an open language for GPGPU, NVIDIA CUDA is the currently dominant proprietary framework.

So far, GPGPU has followed a programming model, where computational tasks are carefully set up as array computations in host code, and are then launched on the GPU(s) as needed. This model enables writing data-intensive programs where host-device memory shuffling is interspersed between kernel calls. GPUs are the ones doing the heavy lifting, so the challenge for the programmer is to keep the hardware busy. The main bottleneck in GPGPU computing today is the limited bandwidth between GPU and the rest of the system.

The fifth main revision of the CUDA architecture [1], Kepler, upgrades the GPGPU computing paradigm by introducing dynamic parallelism, which for the first time allows GPUs to generate and dispatch new work for themselves. Requiring no host program intervention, this capability minimizes the time spent in-between GPU kernel executions.

With hardware supported nested data parallelism using dynamic work generation, programmers can begin to write more expressive programs that feature more flexible algorithms and irregular data structures. Hardware-managed dynamic scheduling marks an excellent time to consider raising the level of abstraction in parallel programming and moving towards functional languages for GPGPU.

V. CONCLUSION

Flat data parallelism is the most successful parallel programming model today with applications ranging from desktop performance to scientific supercomputing. Nested data parallelism improves on the flat model by allowing irregular data structures and illness-balanced computations. NDP allows programmers to write more expressive parallel programs.

Implementing nested data parallelism has previously been limited to a costly whole-program flattening transformation, that renders a nested program compatible with the underlying flat hardware pipelines. Blelloch [3] introduced automated flattening, which has since been adopted in languages like Data Parallel Haskell [6]. Shaw [7] explores NDP further.

NVIDIA CUDA 5.0 [1] features hardware support for nested data parallelism in general purpose GPU computing. The dynamic parallelism in the CUDA architecture allows nested kernel launches and manages job scheduling dynamically without programmer effort. Herhut et al. [8] look into hardware managed threading for NDP.

Hardware support makes nested data parallelism practical. Functional programming offers a different view to data parallelism, which should be embraced with NDP, as it enables a higher level of abstraction for parallel programming.

REFERENCES