Brahma.FSharp: Power of Functional Programming to Create Portable GPGPU-enabled .NET Applications

Nikolai Ponomarev[1][0009−0000−2382−5687], Vladimir Kutuev[1][0000−0001−7749−4940], and Semyon Grigorev[1][0000−0002−7966−0698]

[1] Saint Petersburg State University, 7-9 Universitetskaya Embankment, St. Petersburg, Russia
[2] n.ponomarev@spbu.ru, v.kutuev@spbu.ru, s.v.grigoriev@mail.spbu.ru

Abstract. Widening of GPGPU applicability increases the interest to high-level languages for GPGPU programming development. One of the challenges is to create a portable solution which provides static checks and being integrated with application platforms. We propose a tool—Brahma.FSharp—that allows one to utilize OpenCL-compatible devices in .NET applications, and to develop homogeneous code using familiar .NET tools. Brahma.FSharp facilitates GPU kernel development using F# programming language that is functional-first statically typed .NET language. Compile-time metaprogramming techniques, provided by F#, allows one to develop generic type-safe kernels. We show portability of the proposed solution by running several algorithms developed with it across different platforms and devices.

Keywords: Functional programming · GPGPU · FSharp · .NET · OpenCL

1 Introduction

Last decades, the utilization of GPGPUs has become more popular not only in scientific or dedicated applications, but also in regular business applications. In such cases, the focus has shifted from peak performance to transparent offloading of computations to accelerators. As a result, respective tools for integration of GPGPUs into platforms such as the JVM [9,14,8] or .NET [2,5] have been developed. Note that in real-world applications, the problem is not only to offload computations to GPGPUs, but also to orchestrate heterogeneous asynchronous applications that may involve computations on several GPGPUs.

At the same time, the utilization of existing functional languages and creation of new ones for GPGPU programming looks promising due to their safety, flexibility, ability to use advanced optimization techniques, and capacity to create high-level abstractions. This has led to projects such as Futhark [3], Lift [10], AnyDSL [6], and Accelerate [1].

Currently, there are very few combinations of mature business application development platforms and functional programming languages. One of them

is the .NET platform with the F# programming language. There are several tools, such as Alea GPU [5], FCSL [2], and ILGPU[3], that enable integration of GPGPUs into .NET applications without using unsafe and low-level mechanisms like string-level kernel creation. While FSCL and Alea GPU use F# to create kernels, ILGPU works at the IL level, which limits the ability to use high-level features and non-trivial optimizations.

Alea GPU specifically targets CUDA-only environments, focusing on HPC and machine learning workloads through existing CUDA libraries. This design inherently prevents support for non-NVIDIA GPUs, including a wide range of low-power devices. FCSL—an F# to OpenCL translator—originates from a research project primarily investigating automatic computation scheduling in heterogeneous systems. Despite useful features like automatic caching of compile kernels and an extensible compiler framework, it abstracts away certain low-level details that are critical for performance tuning, including parameter binding specifics, memory allocation control, and other optimization-related configurations.

In this work, we propose **Brahma.FSharp**[4]—a tool for portable GPGPU-enabled .NET application development that provides transparent and safe integration with accelerators—and demonstrate its portability across a variety of platforms and devices.

2 Brahma.FSharp

Brahma.FSharp is a tool that enables GPGPU utilization in .NET applications, allowing developers to write both kernels and supporting code in pure F# [12]—a functional-first, multiparadigmal programming language for the .NET platform. This language combines functional programming features (including first-class functions, generics, and static strong typing with automatic type inference) with seamless integration into a mature business application development platform. Additionally, F# supports imperative coding patterns that are natural for kernel programming.

The tool's core is an F#-subset-to-OpenCL-C translator based on *code quotations* [11]—language feature that provides access to an annotated syntax tree of F# code and to transform it during program execution. This tree can be processed using standard F# functions: for instance, we transform it to generate OpenCL C kernel code. In other words, code quotations enable runtime metaprogramming for creating configurable kernels during program execution. Unlike compile-time metaprogramming, this approach enables dynamic configuration of work-group-dependent kernel aspects (such as local buffer sizes) without full program recompilation (see line 9 in Listing 1). Crucially, all operations remain strongly and statically typed, eliminating the need for unsafe code involving strings, pointers, or object manipulation. From the user's perspective, a compiled quotation (compiled kernel) exposes a type signature that enforces

[3] ILGPU project web page: `https://ilgpu.net/`

[4] Sources of Brahma.FSharp: `https://github.com/YaccConstructor/Brahma.FSharp`

```
1   let mXmKernel
2       (opAdd: Quotations.Expr<'a -> 'b -> 'a>)
3       (opMult: Quotations.Expr<'e -> 'f -> 'b>)
4       (zero: Quotations.Expr<'a>) ... (* other parameters *)  =
5           ... // Supplementary code
6           let kernel = <@ fun 2dRange m1 m2 res ->  // Quoted code
7               ...
8               let acc = %zero // Embedded identity value
9               let lBuf = localArray lws // captured from context
10              ...
11              acc <- (%opAdd) acc ((%opMult) x y) // Embedded operations
12              ... @>
13          ... // Supplementary code
14
15  let intArithmeticKernel = mXmKernel <@ (+) @> <@ ( * ) @> <@ 0 @>
16  let intMinPlusKernel =
17      mXmKernel <@ (min) @> <@ (+) @> <@ Int.MaxValue @>
```

Listing 1: An example of generic matrix multiplication kernel

parameter consistency with the original quotation. In other words, the compiled kernel retains full type information about its arguments, enabling the compiler to verify parameter binding correctness at compile-time without needing additional type annotations.

Actually, creating a truly generic `compile` function that converts high-level code to compiled kernels is a nontrivial challenge. As a result, many other tools either erase all type information or require manual type specifications. For example, ILGPU requires manual specification of the compiled kernel type.

Listing 1 (lines 6-12) shows an example of quoted code (part of a generalized matrix multiplication kernel). This example demonstrates typed quotation composition: the operations `opAdd` and `opMult`, along with the identity element `zero`, maintain type consistency and can be specified externally at runtime. Thus, we can create highly configurable kernels generator and instantiate specific kernels as needed (lines 15-16).

The translator supports not only the imperative subset of F# and primitive types, but also F#-specific features including structs, tuples, discriminated unions, pattern matching, and nested bindings. Additionally, it implements OpenCL-specific functionality such as atomic operations, memory barriers, and allocation of local/thread-local arrays. For data transfer and manipulation, Brahma.FSharp provides two key primitives—`ClArray<'t>` and `ClCell<'t>`— that are F#-array-friendly wrappers for `ClBuffer`.

The framework implements a standard kernel execution workflow with typed wrappers[5] It is worth noting that F# is friendly to asynchronous programming and provides huge amount of parallel and asynchronous programming

[5] Portability is achieved through configurable `libopencl` path specification.

primitives [13]. By utilizing *MailboxProcessor*—F#'s built-in message passing primitive—to wrap command queues, we achieve GPGPU communication patterns that naturally complement F#'s asynchronous programming model.

3 Evaluation

In this section, we present experimental evaluations[6] of the Brahma.FSharp platform, demonstrating its core capabilities on conventional (non-HPC) devices[7]. We assess Brahma.FSharp performance in two representative use cases, detailed in the following sections[8].

1. The first one is an image convolution that demonstrates utilization of several GPU-s using F# MailboxProcessor.
2. **Image convolution**: showcases smooth multi-GPU utilization through F#'s MailboxProcessor for efficient tasks distribution.
3. **Matrix multiplication**: demonstrates creation of generic strongly statically typed kernels, utilization of local and private memory for performance optimization, and portability across different devices.

3.1 Image Convolution

We implemented image convolution as a demonstration of multi-GPU utilization. As established in [5], F#'s native asynchronous programming model significantly simplifies the creation of complex nonlinear computational workflows combining GPU computations with CPU processing and I/O operations. Our implementation leverages F#'s MailboxProcessor, which Brahma.FSharp exposes as the primary interface for GPU communication. The kernel is simply wrapped as illustrated in Listing 2. For workload distribution, we developed a basic load balancer that dynamically routes each new image to the GPU agent with the fewest pending messages in its input queue.

We evaluate this solution on a **Lenovo** platform with two GPUs: NVIDIA GeForce MX150 and Intel(R) UHD Graphics 620. We assume all images are loaded into RAM and converted to grayscale. A typical sequence of filters is applied: 3 Gaussian blur operations (5×5 kernel) followed by edge detection (5×5 kernel). 420 images (1gb of data) was handled in 40 seconds with two GPUs, in 64 seconds using Nvidia GPU only, and in 97 seconds using Intel GPU only. These results demonstrate that even a naive multi-GPU workflow can achieve up to 30% speedup compared to using only the fastest single GPU (NVIDIA) in the system.

[6] Benchmarking automation infrastructure sources: `https://github.com/vkutuev/matrix-benchmark`.

[7] This configuration is particularly relevant for business applications built on .NET.

[8] Respective code is available on GitHub: `https://github.com/gsvgit/ImageProcessing/tree/matrix_multiplication`.

```
let imgProcessor filter (imgSaver: MailboxProcessor<_>) =
    MailboxProcessor.Start(fun inbox ->
        let rec loop ... = async { // Async message processing loop
            let! msg = inbox.Receive() // Load message
            match msg with
            | EOS ch -> // Handle end of stream
                imgSaver.PostAndReply EOS
                ch.Reply()
            | Img img -> // Handle image
                let filtered = filter img // Convolution
                imgSaver.Post (Img filtered)
                return! loop ... }// Got to next message
        loop ...)
```

Listing 2: MailboxProcessor-based wrapper for kernel to make it easier to integrate it to complex workflow

3.2 Matrix Multiplication

We evaluate a generic kernel parametrized by types and operations (see Listing 1), implemented in F#. Several basic optimizations, inspired by "Tutorial: OpenCL SGEMM tuning for Kepler" by Cedric Nugteren[9], were applied. Namely, we use tiling in local and private memory. However, the current version supports only square matrices and square tiles.

We selected two competitors for evaluation. The first is CLBlast[10] [7], a highly optimized OpenCL-based BLAS implementation tuned even for low-power mobile GPUs. The second is OpenBLAS[11], a highly optimized CPU-based BLAS implementation. Additionally, we executed OpenCL-based solutions on CPUs using POCL [4]. All competitors were compiled and run with their default configurations.

We evaluate all competitors on several platforms listed below.

- **Lenovo**: Intel Core i7-8550U CPU, NVIDIA GeForce MX150 and Intel UHD Graphics 620 GPUs.
- **Zen**: Intel Core i5-1340P CPU, Intel Iris Xe Graphics G7 80EUs GPU.
- **MILK-V**: SpacemiT M1 CPU, IMG BXE-2-32 GPU.

We generate random square matrices with elements of type `float32` and use the typical arithmetic semiring because our competitors do not provide generic kernels. Time is measured as an average of 10 runs. We measure time of client function execution, so it includes data transfer. Results of the evaluation are represented in Figure 1, where we show both time and relative speedup calculated as the ratio of corresponding average execution times.

[9] "Tutorial: OpenCL SGEMM tuning for Kepler": `https://cnugteren.github.io/tutorial/pages/page1.html`

[10] CLBlast source code: `https://github.com/CNugteren/CLBlast`

[11] OpenBLAS source code: `https://github.com/OpenMathLib/OpenBLAS`

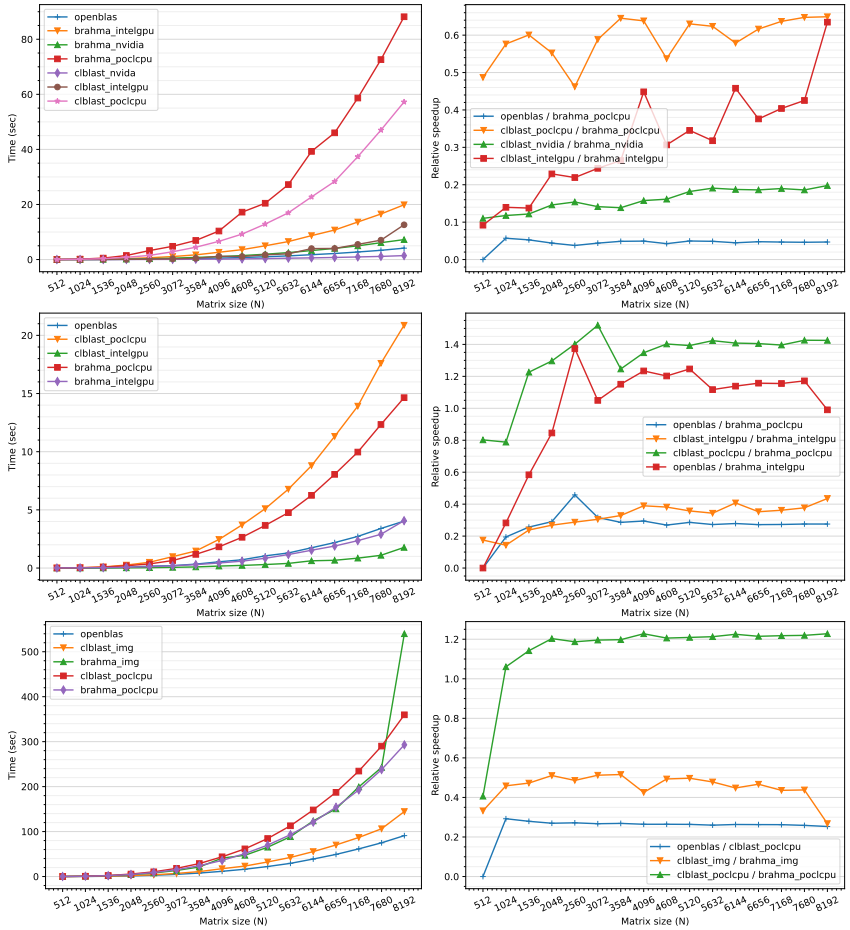

Fig. 1: Matrix multiplication performance: 1st row for **Lenovo**, 2nd for **Zen**, 3rd for **MILK-V**

First, we show that Brahma.FSharp allows one to create portable solutions. While our kernel is not as optimized as the kernel from CLBlast, relative speedup analysis shows that there is a room for tuning: in many cases, the performance gap decreases with data size increase (**Lenovo** esp. Intel GPU; **Zen**). However, in some cases the behavior is more complex: for **MILK-V**, our solution on CPU using POCL demonstrates better performance than CLBlast, but on the respective GPU the performance gap slightly increases with data size increase.

This behavior can be explained by differences in kernel tuning rather than by a Brahma.FSharp technology problem. Thus, while it is unlikely possible to fully hide .NET overhead, it appears possible to minimize it to become comparable with competitors on large matrices. Following the methodology proposed in [7], we plan to support additional kernel parameters enabling more precise tuning of performance-critical aspects.

4 Conclusion and Future Work

Brahma.FSharp—a tool for developing cross-platform, GPGPU-accelerated .NET applications—is presented. We demonstrated applications portability by evaluating performance across multiple platforms, including RISC-V with PowerVR GPGPU and embedded Intel GPUs.

While the work remains in progress, Brahma.FSharp already enables creation of linear algebra kernels sufficiently performant for integration into libraries like Math.NET Numerics, allowing transparent offloading of generic linear algebra operations to GPGPUs. Such integration is planned for the near future.

Although the agent-based communication approach aligns naturally with both OpenCL and F#, MailboxProcessor may not be optimal for high-frequency CPU-GPU communication. Alternative solutions like Hopac[12] or lightweight command queue wrappers could provide better performance for latency-critical code.

A significant challenge for future research involves automatic memory management. Currently, GPGPU memory requires manual cleanup despite .NET's garbage collector. Developing a hybrid approach that leverages automatic garbage collection while retaining manual control when needed remains an open problem.

Acknowledgments

We sincerely thank the anonymous reviewers for their thorough evaluation, insightful comments, and valuable suggestions that helped improve this work. This research has been supported by the St. Petersburg State University, grant id 116636233.

[12] Hopac and MailboxProcessor performance comparison: `https://vasily-kirichenko.github.io/fsharpblog/actors`

References

1. M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 3–14, New York, NY, USA, 2011. Association for Computing Machinery.
2. G. Coco. *Homogeneous programming, scheduling and execution on heterogeneous platforms*. PhD thesis, Ph. D. Thesis.\Gabriel Coco.-University of Pisa, 2014.-254 p, 2014.
3. T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571, June 2017.
4. P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43(5):752–785, Aug. 2014.
5. P. Kramer, D. Egloff, and L. Blaser. The alea reactive dataflow system for gpu parallelization. In *Proc. of the HLGPU 2016 Workshop, HiPEAC*, 2016.
6. R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt. Anydsl: a partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018.
7. C. Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18, New York, NY, USA, 2018. Association for Computing Machinery.
8. N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for gpus in scala. In *ACM SIGPLAN Notices*, volume 47, pages 107–116. ACM, 2011.
9. P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. Rootbeer: Seamlessly using gpus from java. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 375–380. IEEE, 2012.
10. M. Steuwer, T. Remmelg, and C. Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 74–85. IEEE Press, 2017.
11. D. Syme. Leveraging. net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.
12. D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Springer, 2012.
13. D. Syme, T. Petricek, and D. Lomov. The f# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*, pages 175–189. Springer, 2011.
14. Y. Yan, M. Grossman, and V. Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *European Conference on Parallel Processing*, pages 887–899. Springer, 2009.