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 provides ability to create GPU kernels 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 utilization of GPGPUs not only in scientific or dedicated applications, but also in regular business applications becomes more popular. In such cases not peak performance, but transparent offloading of computations to accelerator has come into focus. As a result, respective tools for integration of GPGPUs into such platforms as JVM [9,14,8] or .NET [2,5] are developed. Note that in real-world application the problem no only to offload some computations on GPGPU, but to orchestrate heterogenous asynchronous application that involves computations on possible several GPGPUs.

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

Nowadays there are very few combination of mature business application development platform and functional programming language. One of them is a

.NET platform and F# programming language. There are several tools, such as Alea.GPU [5], FCSL [2], ILGPU[3], that allows one to integrate GPGPUs into .NET application without using such unsafe and low-level mechanisms like string-level kernels creation. While FSCL and Alea.GPU use F# to create kernels, ILGPU works on IL level that limits ability to use high-level features and nontrivial optimizations.

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

2 Brahma.FSharp

Brahma.FSharp is a tool that allows one to utilize GPGPUs in .NET applications and write kernels and all supplementary code in pure F# [12] that is a functional-first multiparadigmal programming language for .NET platform. This language combines functional programming, including first-class functions, generics, static strong typing with automatic type inference, with transparent integration with the a platform for business applications development with mature infrastructure. At the same time, F# provides an ability to write imperative code that is native for kernel programming.

Core of the tool is a translator of F# subset to OpenCL C that is based on *code quotations* [11] that allows one to get access to annotated tree of the F# code and transform it during program execution. This tree can be transformed using regular F# functions: for example, it can be translated to other language, that is we do to generate OpenCL C code for kernels. Other words, code quotations is a runtime metaprogramming feature that allows us to create kernels configurable during program execution. For example, in opposite to compile time metaprogramming, it is possible to configure work group size dependent parts of kernel (e.g. local buffer size) without recompilation of whole program (look at line 9 of listing 1). The main feature is that everything is strongly and statically typed: no unsafe code that uses strings, pointers, objects, etc. At the user side, compiled quotation (compiled kernel) has the signature that requires parameters of types that are in agreement with initial quotation.

An example of quoted code (actually, part of the generalized mXm kernel) is presented in listing 1 (lines 6–12). This code also demonstrates typed composition of quotations: operations `opAdd` and `opMult`, and identity element `zero`, have agreed types and can be specified outside the kernel in run time. So, we can write highly configurable kernels generator and instantiate specific kernels later, as shown in lines 15–16.

The translator supports not only imperative subset of F# and primitive types, but also F#-specific features like structs, tuples, discriminated unions,

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

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

```
let mXmKernel
    (opAdd: Quotations.Expr<'a -> 'b -> 'a>)
    (opMult: Quotations.Expr<'e -> 'f -> 'b>)
    (zero: Quotations.Expr<'a>) ... (* other parameters *)  =
      ... // Supplementary code
      let kernel = <@ fun 2dRange m1 m2 res ->  // Quoted code
        ...
        let acc = %zero // Embedded identity value
        let lBuf = localArray lws // captured from context
        ...
        acc <- (%opAdd) acc ((%opMult) x y) // Embedded operations
        ... @>
      ... // Supplementary code

let intArithmeticKernel = mXmKernel <@ (+) @> <@ ( * ) @> <@ 0 @>
let intMinPlusKernel =
    mXmKernel <@ (min) @> <@ (+) @> <@ Int.MaxValue @>
```

Listing 1: An example of generic matrix multiplication kernel

pattern matching, nested bindings. Also it supports OpenCL specific features like atomic functions, barriers, local and thread-local arrays allocation. For data transferring and manipulation Brahma.FSharp provides `ClArray<'t>` and `ClCell<'t>` primitives that are F#-array-friendly wrappers for `ClBuffer`.

Brahma.FSharp provides typical workflow to run kernels and implements respective typed wrappers for it[5]. It is worth noting that F# is friendly to asynchronous programming and provides huge amount of parallel and asynchronous programming primitives [13]. Utilization of *MailboxProcessor*, that is F#-native massage passing primitive, to wrap command queue allows us to make communication with GPGPU friendly for asynchronous programming in F#.

3 Evaluation

In this section we provide experiments[6] with Brahma.FSharp platform which are aimed to demonstrate its main features on regular (not HPC) devices[7]. We evaluated Brahma.FSharp in two cases listed below and described in respective sections.

1. The first one is an image convolution that demonstrates utilisation of several GPU-s using F# MailboxProcessor.
2. Second one is a matrix multiplication that demonstrates ability to create generic strongly statically typed kernels, to utilize local and private memory

[5] Configuration of path to `libopencl` allows one to make the solution portable.
[6] Sources of benchmarking automation infrastructure: `https://github.com/vkutuev/matrix-benchmark`.
[7] Looks more suitable for business applications that use .NET.

for performance optimization, and to demonstrate portability across different devices.

3.1 Image Convolution

We implement image convolution in order to demonstrate multiple GPGPUs utilization. Native for F# asynchronous model, as was shown in [5], simplifies creation of complex workflows that include GPGs. We use F# MailboxProcessor because Brahma.FSharp provides it as an interface for communication with GPUs. Kernel is simply wrapped as shown in 2. Simple load balancer that send next image to an agent with less number of messages in input queue was implemented.

```
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

We evaluate this solution on a **Lenovo** platform with two GPUs: NVIDIA GeForce MX150 and Intel(R) UHD Graphics 620. We assume that all images are loaded into RAM and converted to grayscale. Typical chain of filters is applied: 3 gaussian blur (5×5 kernel), then edges 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. Thus we can see that even naive multi-GPU workflow allows one to achieve up to 30% speedup.

3.2 Matrix Multiplication

We evaluate generic kernel parametrized by types and operations 1, implemented in F#. Several basic optimizations, inspired by "Tutorial: OpenCL SGEMM tuning for Kepler" by Cedric Nugteren[8], were applied. Namely, we use tiling in

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

local and private memory. But current version supports only square matrices and square tiles.

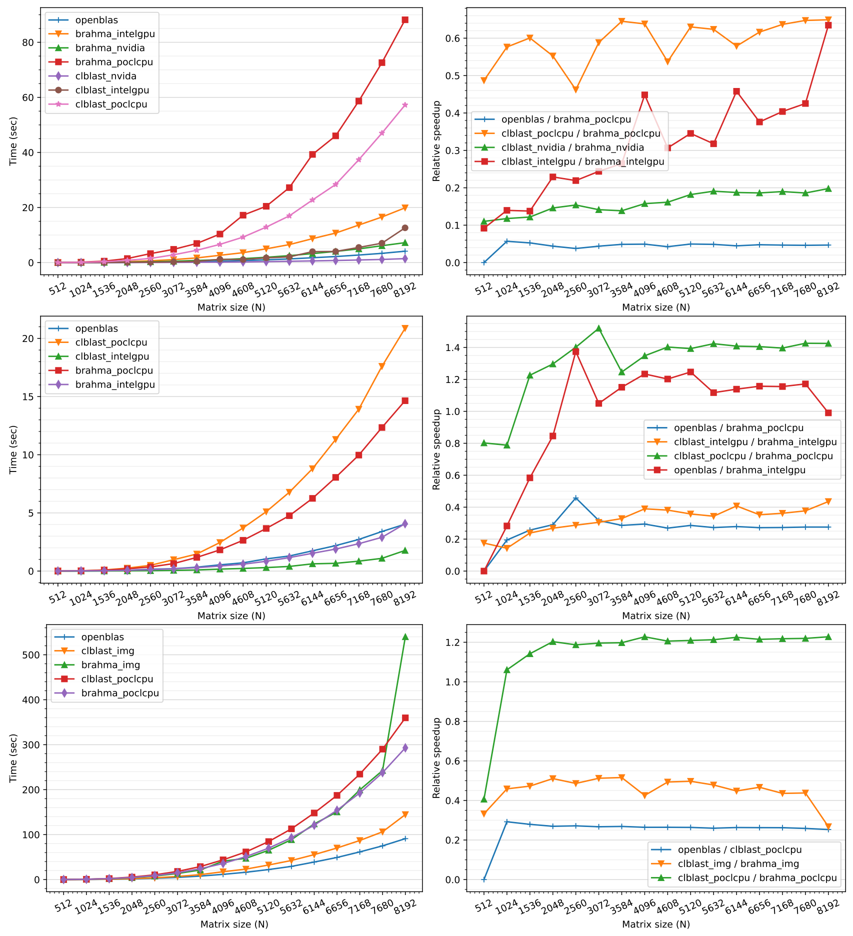

Fig. 1: Matrix multiplication performance: 1st row for **Lenovo**, 2nd for **Zen**, 3rd for **MILK-V**

We choose two competitors. The first one is the CLBlast[9] [7] that is a highly-tuned (even for liw-power mobile GPUPUs) OpenCL-based BLAS implementation. The second oe is the OpenBLAS[10] that is a highly-tuned BLAS implementation foe CPU. All competitors were compiled and run with default settings. Additionally, we run OpenCL-based solutions on CPUs using POCL [4].

[9] CLBlast source code: `https://github.com/CNugteren/CLBlast`
[10] OpenBALAS source code: `https://github.com/OpenMathLib/OpenBLAS`

We evaluate all competitors on several platforms listed below.

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

We generate random square matrices with elements of type `float32` and use typical arithmetic semiring because our competitors does not provide generic kernels. Time is measured as an average of 10 runs. We measure time of client function execution, so it includes datatransfer. Results of evaluation represented in figure 1: we show both time and relative speedup.

First of all, we show that Brahma.FSharp allows one to create portable solutions. Obviously, our kernel is not such optimized as kernel from CLBlast, but relative speedup analysis shows that more tuning required: in much cases performance gap decreases with data size increase (**Lenovo** esp. Intel GPU; **Zen**). But in some cases behavior is more complex: for **MILC-V** our solution on CPU using POCL demonstrates better performance that CLBlast, but on respective GPU performance gap slightly increases with data size increase.

Such behavior can be explained by differences in kernel tuning, but not by technology problem. So, while it is unlikely possible to hide dotnet overhead fully, it looks possible to minimize it to be comparable on big matrices. To do it we should to create finer tuned and more flexible kernel to allows one better fit performance-affecting parameters.

4 Conclusion and Future Work

Brahma.FSharp — a tool to create cross-platform GPGPU-enabled .NET application presented. We demonstrated portability of application by evaluating them on a set of platforms including RISC-V with PowerVR GPGPU, and embedded Intel GPUs.

While work still in progress, Brahma.FSharp allows one to create linear algebra related kernels performant enough to be integrated in libraries like Math.Net Numerics to offload generic linear algebra transparently to GPGPU. Such an integration is planned to the nearest future.

Also, within the translator improvements, it is necessary to improve performance of data transferring between managed and native memory for complex types such as discriminated unions.

While agent-based approach for communications is a native for both OpenCL and F#, MailboxProcessor processor may not be the best choice for it especially in cases with high-frequent CPU-GPU communications. It may be better to use more performant libraries like Hopac [11] or even provide light-weight wrapper for direct access to command queue for latency-critical code.

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

One of nontrivial problem for the future research is an automatic memory management. For now, GPGPU-related memory should be cleaned manually, but .NET has automatic garbage collector. How can we offload buffers management on it with ability to switch to manual control if required.

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.