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

Nikolai Ponomarev, Vladimir Kutuev, **Semyon Grigorev**

St. Petersburg State University

October 7, 2025

General-Purpose Computing on GPUs

- Not only HPC and scientific computations
- Widely used in regular business applications
  - Images and video processing
  - AI/ML
  - Financial analysis
  - ...
- Highest performance vs development simplification
  - Native execution vs managed environments (JVM, .NET, etc)
  - Low-level languages vs high-level ones
  - ...
- Functional languages for GPGPU
  - Type-safety, flexibility, advanced optimizations
  - Futhark, Lift, AnyDSL, ...

.NET and F#

- .NET — one of popular platforms for business applications development
- F# — functional-first programming language for .NET
  - Statically strongly typed
    - Type-safe development of generic kernels
  - F# Code Quotations
    - Compile-time metarogramming
    - Runtime access to full-featured AST
    - Statically typed
    - Type-safe composition
  - F# Mailbox Processor
    - Integrated language-native primitive for message-passing-based asynchronous programming

F# for GPGPU Programming: Design Decision Points

- **Target technology**
  - CUDA (*e.g. Alea GPU*) — mature stable infrastructure but Nvidia only
  - OpenCL (*e.g. FSCL*) — wide range of devices and vendors
- **Translator input**
  - Bytecode (*e.g. ILGPU*) — one tool for all languages on the platform
  - Source code (*e.g. FSCL*) — an ability to use language-specific features
- **Control level**
  - Fine-grained (*e.g. ILGPU*)
  - black-box-like automatization (*e.g. FSCL*)
- **Type safety**
  - None
  - Manual type annotations
  - Automatic types inference and checking
- **Focus**
  - Alea GPU: HPC, AI/ML, bindings for existing Cuda libs
  - FSCL: automatic computation scheduling in heterogeneous systems
  - ...

is a **F# quotation to OpenCL translator** and respective **runtime** to utilize GPGPUs in F# applications

- **Target technology**: OpenCL
- **Translator input**: source code (using quotations)
  - Not only primitive types, but also discriminated unions, structs, records
  - Pattern matching, mutable and immutable bindings, nested bindings
- **Control level**: fine-grained memory management and kernels compilation process
- **Type safety**: automatic
  - Ten lines of types' manipulation magic that carefully build type for compiled kernel[1]
- **Focus**: safety, static checks, development simplification
- Mailbox Processor wrapper for command queue
  - Native way to integrate communication with GPGPUs into asynchronous pipelines

[1] . . . and brakes some F# compiler optimizations

Research Questions

- Can we create code portable across different devices and vendors?
- Can we create generic type-safe kernels?
- Can we utilize well-established optimizations if necessary?
- Does Mailbox Processor allows one to create asynchronous workflows that utilize heterogeneous devices?

An Example of Generic Matrix Multiplication Kernel

```
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 @>
```

An Example of Generic Matrix Multiplication Kernel

```
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 @>
```

An Example of Generic Matrix Multiplication Kernel

```
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 @>
```

An Example of Generic Matrix Multiplication Kernel

```
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 @>
```

An Example of Generic Matrix Multiplication Kernel

```
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 @>
```

An Example of Generic Matrix Multiplication Kernel

```
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 @>
```

An Example of Generic Matrix Multiplication Kernel

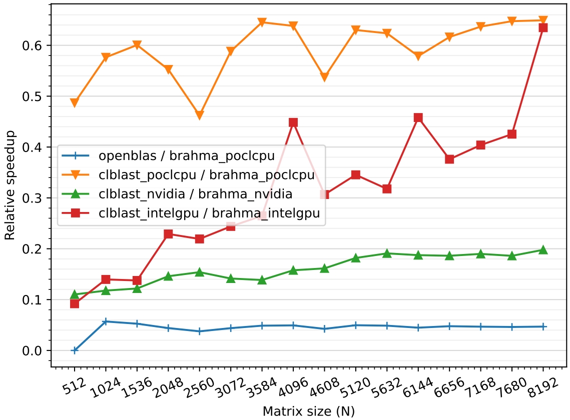
```
let mXmKernel (opAdd: Quotations.Expr<'a -> 'b -> 'a>)
              (opMult: Quotations.Expr<'e -> 'f -> 'b>)
              (zero: Quotations.Expr<'a>) ... (* other parameters *)  =
      ... // Supplementary code
      let kernel = <@ fun 2dRange m1 m2 res ->  // Quoted code
        ...
        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 @>
```

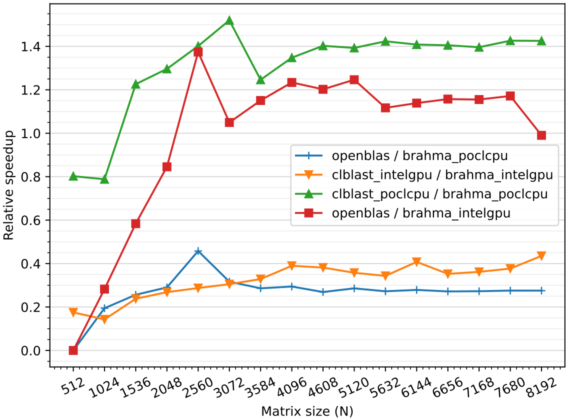
Matrix Multiplication Evaluation: Environment

- Optimizations
  - Tiling in local and private memory[2]
  - Square matrices and square tiles
- Hardware
  - **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
- Software
  - **CLBlast**: optimized OpenCL-based BLAS implementation tuned for low-power mobile GPUs
  - **OpenBLAS**: optimized CPU-based BLAS implementation
- **PoCL** to execute OpenCL-based solutions on CPUs

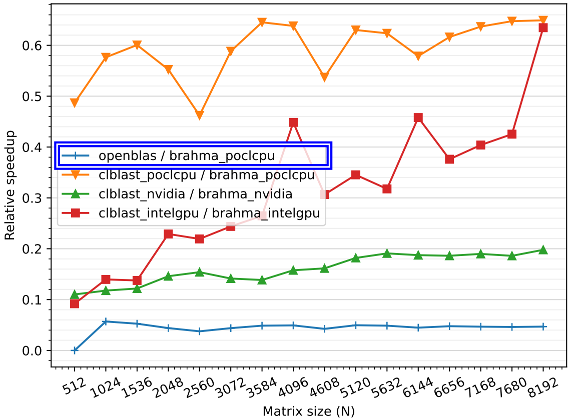
[2] Inspired by "Tutorial: OpenCL SGEMM tuning for Kepler" by Cedric Nugteren

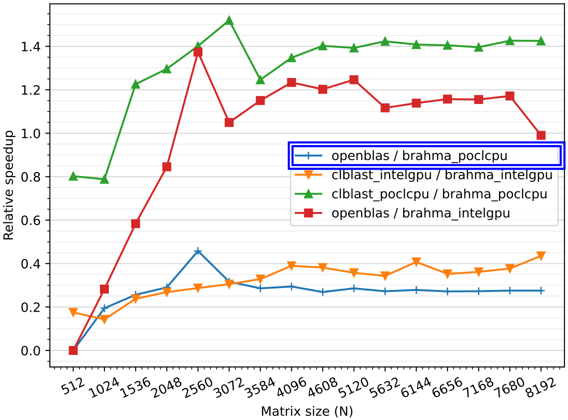
Matrix Multiplication Performance: **Lenovo** and **Zen**



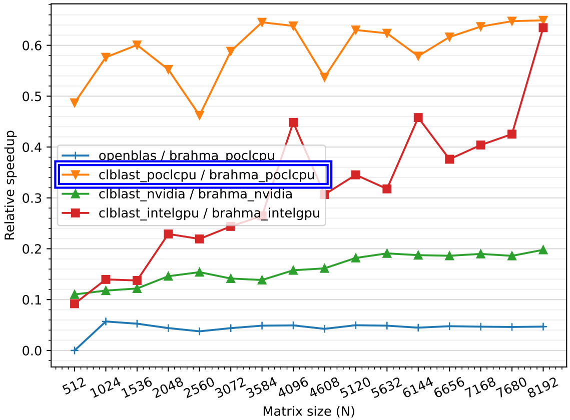


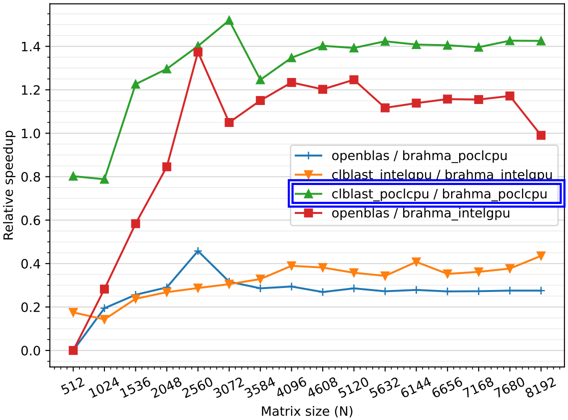
Matrix Multiplication Performance: **Lenovo** and **Zen**



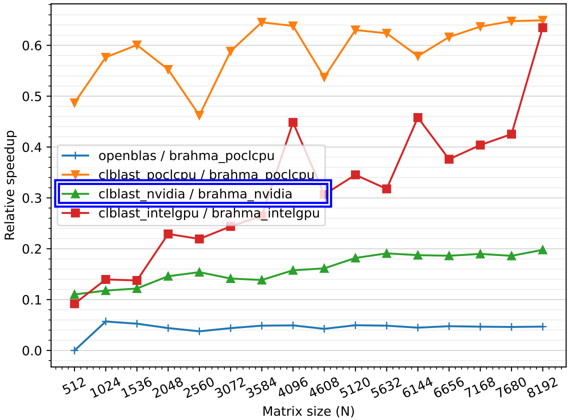


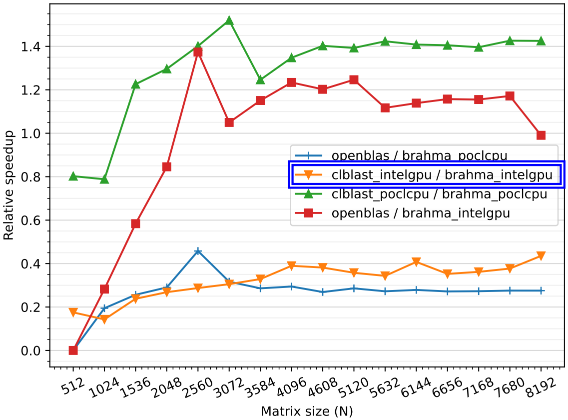
Matrix Multiplication Performance: **Lenovo** and **Zen**



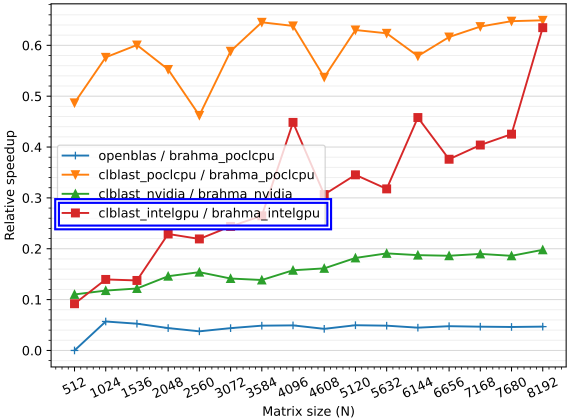


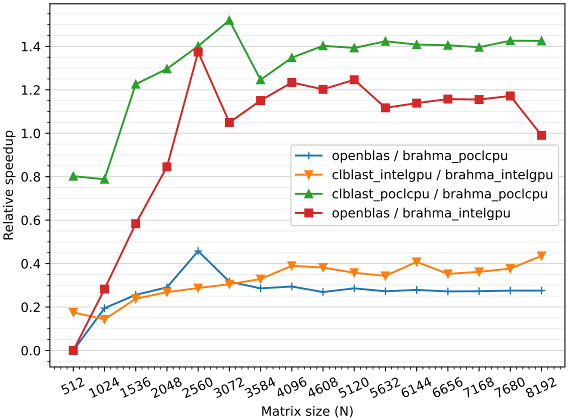
Matrix Multiplication Performance: **Lenovo** and **Zen**



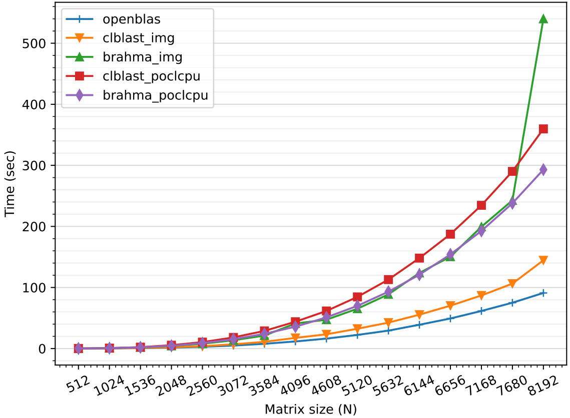


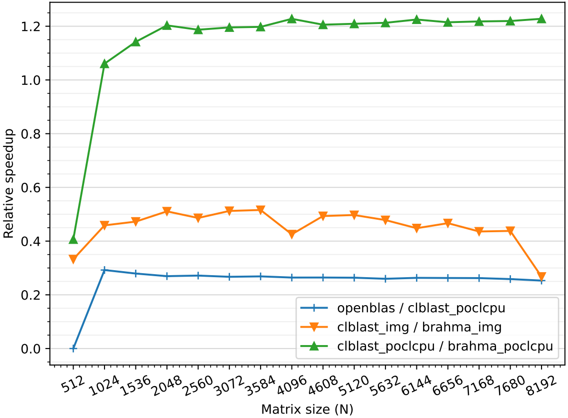
Matrix Multiplication Performance: **Lenovo** and **Zen**



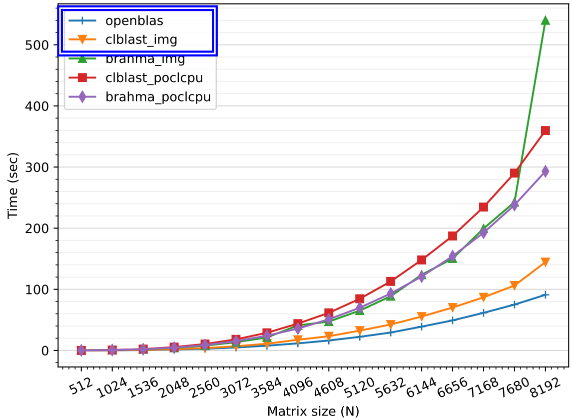


Matrix Multiplication Performance: **MILK-V**





Matrix Multiplication Performance: **MILK-V**



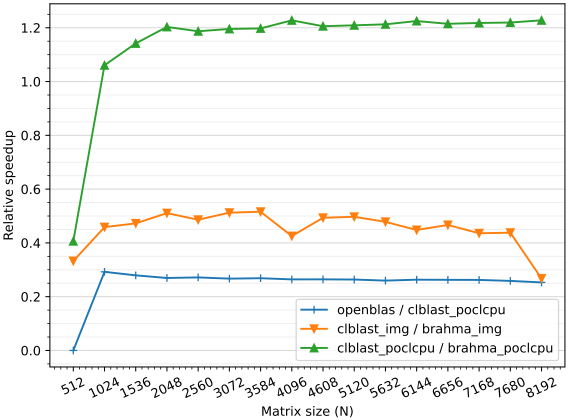


Image Processing Using Mailbox Processor[3]

```
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 ...)
```

[3] Basic version

Image Processing Using Mailbox Processor[3]

```
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 ...)
```

[3] Basic version

Image Processing Using Mailbox Processor[3]

```
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 ...)
```

[3] Basic version

Image Processing Using Mailbox Processor[3]

```
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 ...)
```

[3] Basic version

Image Processing Using Mailbox Processor[3]

```
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 ...)
```

[3] Basic version

Image Processing Using Mailbox Processor[3]

```
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 ...)
```

[3] Basic version

Image Processing Using Mailbox Processor[3]

```
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 ...)
```

[3] Basic version

Evaluaiton of Image Processing

- Environment
  - **Lenovo**: Intel Core i7-8550U CPU, NVIDIA GeForce MX150 and Intel UHD Graphics 620 GPUs
  - Stream processing: 420 images (1Gb of data), load–process–store
  - 5×5 kernels sequence: 3 Gaussian blur, 1 edge detection
- Results
  - 64 seconds on Nvidia GPU
  - 97 seconds on Intel GPU
  - 40 seconds using Intel and Nvidia simultaneously: **30% speedup**

Conclusion

- Brahma.FSharp allows one to create portable solutions
  - We want to create homogeneous code[4] to simplify development and portability process[5]
- Brahma.FSharp allows one to create highly-parameterizable type-safe kernels
  - Code support simplification: single highly-optimized and type-safe kernel[6]
- Brahma.FSharp allows one to utilize well-established optimizations
  - Including manipulations with local and private memory
- Mailbox Processor allows one
  - To create asynchronous workflows that utilize heterogeneous environment in F#-native way
  - To utilize heterogeneous devices to increase performance

- We should reduce runtime overhead
- In case of matrix multiplication, there is a room for kernels tuning

[4] Even .NET-only, without wrappers for native libraries

[5] Write once, run everywhere. In some cases.

[6] But without type-specific optimizations

Useful Links

- Brahma.FSharp sources:
  https://github.com/YaccConstructor/Brahma.FSharp
- Environment for evaluation and examples:
  https://github.com/gsvgit/ImageProcessing

- Email:
  s.v.grigoriev@mail.spbu.ru
- GitHub: gsvgit
- Google Scholar: Semyon Grigorev
- DBLP: Semyon V. Grigorev