Pattern-Based Optimization of Dense Linear Algebraic Computations

Lambda Days

Krakow, 22 February 2018.

Dániel Berényi

GPU Lab, Wigner Research Centre for Physics

András Leitereg, Gábor Lehel, Máté Ferenc Nagy-Egri

Wigner Research Centre for Physics, Budapest

- GPU Laboratory
- Developer support

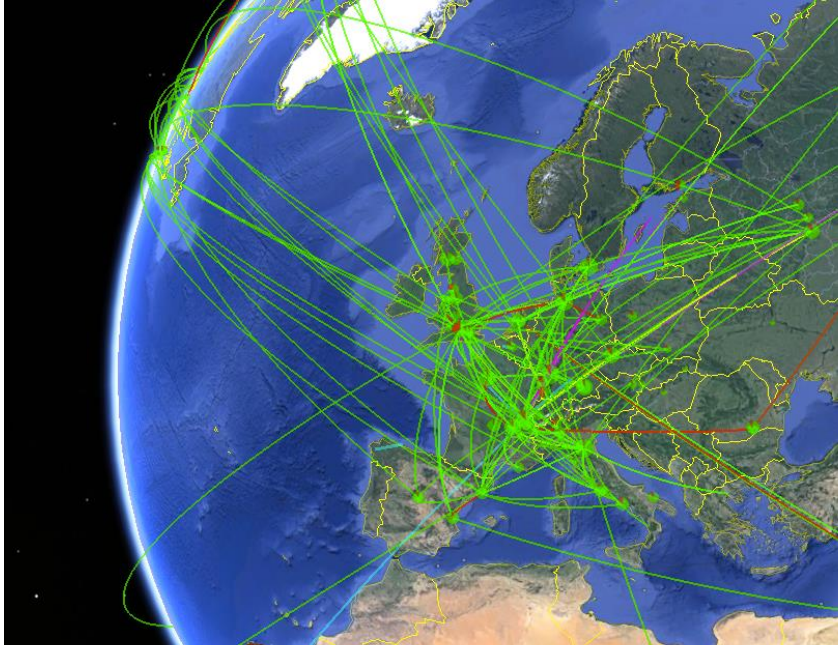
What we face day to day:

Domain experts, who have no programming or hardware expertise

Who need to develop efficient computations, but have no time to delve into hardware details and programming interfaces

The result:
Lots of inefficient badly structured code written by non-experts

Hardware hierarchies



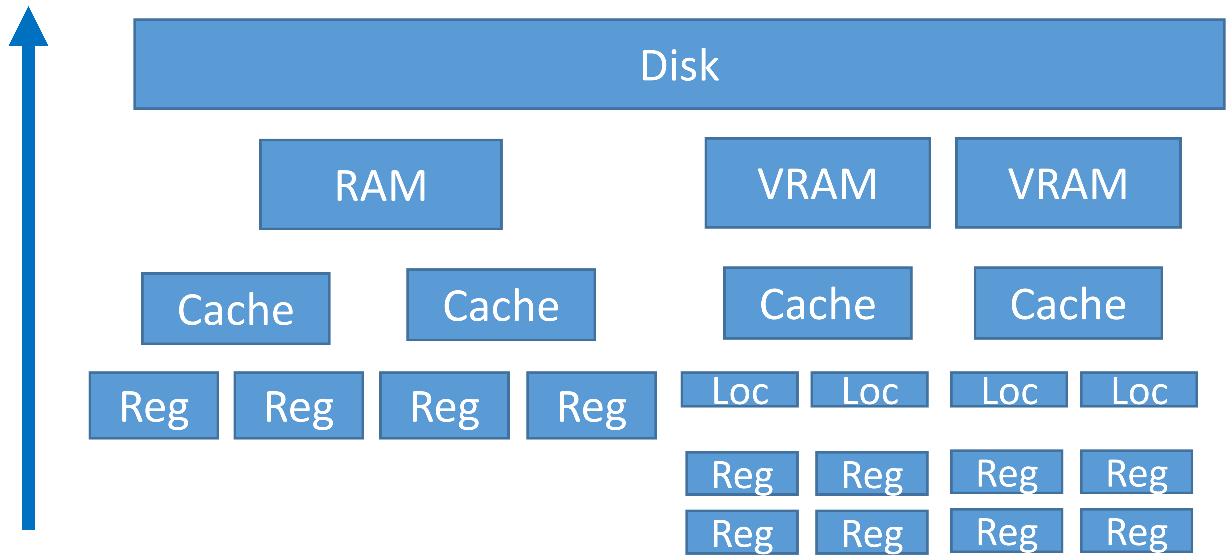
- Computing center
- Clusters of computers
- Multiple devices (CPU, GPU, FPGA)
- Multiple execution units
- Groups of threads

Memory hierarchies

Speed

- Storage
- Device memory
- Caches
- Shared memory
- Registers

Size



Specific example: linear algebra

The heart of simulations, neural networks, modeling and more… must be very efficient!

Hand tuned libraries exists:

- BLAS – fixed primitives, not composable

C++ template libraries:

- Eigen, Armadillo – too specialized on matrices and vectors, what if we need some little extension?
  E.g. tensor contractions?

Specific example: linear algebra

Can we get more flexible,
yet well optimizable primitives?

- That cover existing features of linear algebra and more
- Has primitives that are expressive, yet composable
- Automatic tools can be constructed to optimize them
- Maybe… Just MAYBE…
  Can we have some theoretical basis for it?

Naperian Functors

Fixed shape indexible containers:

```
class Functor f ⇒ Naperian f where
  type Log f
  lookup :: f a → (Log f → a)
  tabulate :: (Log f → a) → f a
```

John Naperian

Details in: Jeremy Gibbons - APLicative Programming with Naperian Functors

European Symposium on Programming, LNCS, vol. 10201::568-583; 2017

Naperian Functors

Fixed shape indexible containers:

```
class Functor f ⇒ Naperian f where
  type Log f
  lookup :: f a → (Log f → a)
  tabulate :: (Log f → a) → f a
```

Like an index

Like an indexer

Like a constructor

Examples:

- Fixed size Arrays, (mathematical) Vectors, …

Counterexamples:

- Maybe, List, …

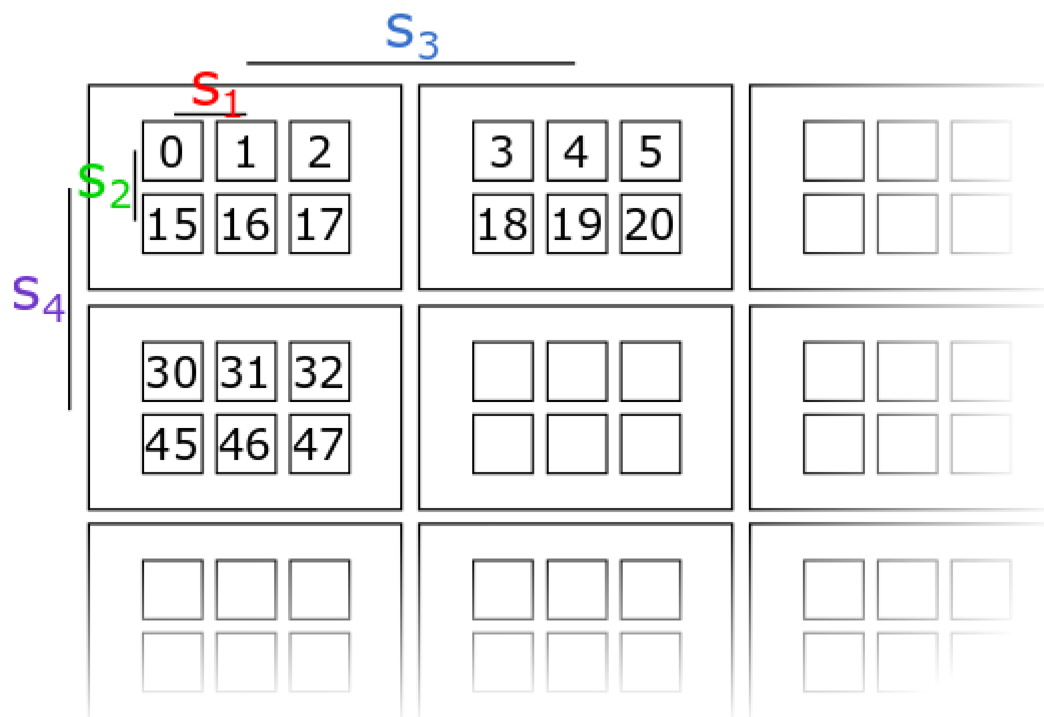
Useful property: transposition $f\ (g\ a) \simeq g\ (f\ a)$

Multidimensional tensors

- We can nest Naperian functors, but can they represent multidimensional and subdivided tensors?

- We can add strides at type level:

- $a^{(120)}$
- $a^{(15)(8)}$
- $a^{(3)(2)(5)(4)}$
- $a^{(3,\mathbf{1})(2,\mathbf{15})(5,\mathbf{3})(4,\mathbf{30})}$



Higher order function primitives

On arrays we may consider the usual primitives:

```
map :: (a → b) → f a → f b

zip :: (a → b → c) → f a → f b → f c

reduce :: (a → a → a) → f a → a
```

What could go wrong?

Higher order function primitives

What happens when we try to compose them?

```
map f . map g = map (f . g)
```

```
map f . zip g = ???
```

Well, seems like we are not closed…

Higher order function primitives

What is the way out? Generalize to n-ary arguments:

`nzip ::` $(a_1 \to a_2 \to \ldots \to b) \to$
$(f\ a_1) \to (f\ a_2) \to \ldots \to$
$\to f\ b$

nzip is closed under compositions

`reducezip ::` $(b \to b \to b) \to$
$(a_1 \to a_2 \to \ldots \to b) \to$
$(f\ a_1) \to (f\ a_2) \to \ldots \to$
$\to b$

We can compose arbitrary nzips before the reduce

Higher order function primitives

How we can optimize them?

Exchange rule pattern for maps:

```
map (\x →
   map (\y → f x y) Y ) X

=

map (\y →
   map (\x → f x y) X ) Y
```

Higher order function primitives

How we can optimize them?

Exchange rule pattern for map/reducezip pair:

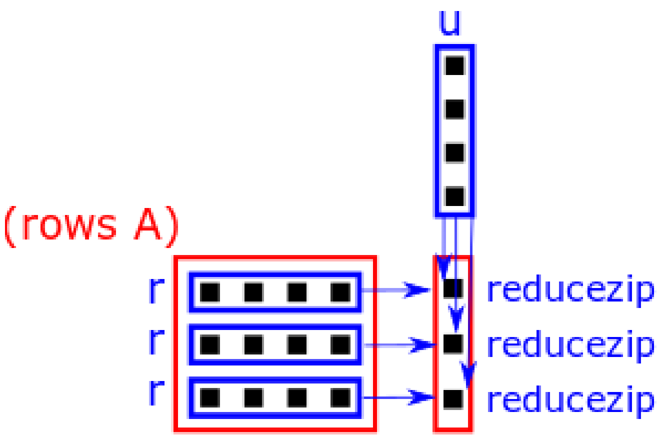
```
map (\r →
   reducezip (+) (∗) r u) A
=
reducezip (zip (+)) (\c v →
   map (\e → e∗v) c) (flip A) V
```

Can be understood in terms of the Naperian functor transpose property

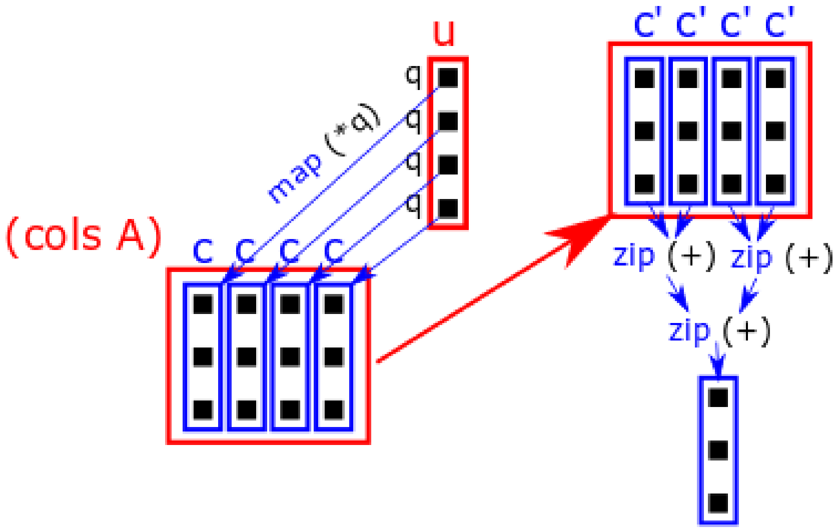
$$f\ (g\ a) \simeq g\ (f\ a)$$

Higher order function primitives

map - reducezip exchange rule in action:
matrix-vector product

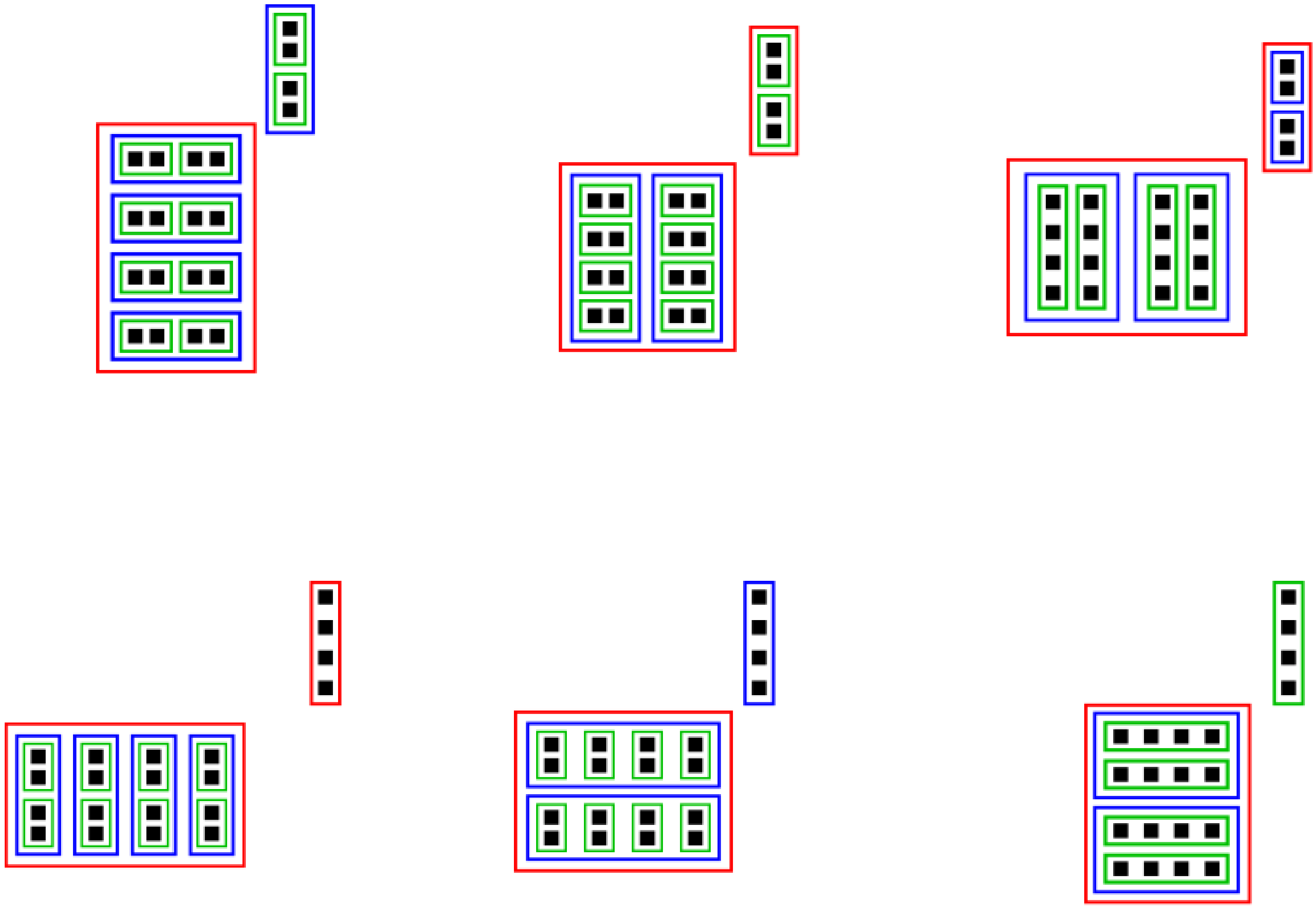


map (\r -> reducezip (+) (*) r u) (rows A)



reducezip (zip (+)) (\c q -> map (\e ->e*q) c) (cols A) u

6 Rearrangements of the matrix-vector multiplication at 1 level of subdivision



Rearrangements of the matrix-matrix multiplication

```
map (\r_A →
    map (\c_B →
        reducezip (+) (*) r_A c_B) B) A
```

What is the performance difference if we reorder?

HoF ordering			Time [s]
mapA	reducezip	mapB	0.45
reducezip	mapA	mapB	1.41
mapA	mapB	reducezip	4.67
mapB	mapA	reducezip	6.05
reducezip	mapB	mapA	13.8
mapB	reducezip	mapA	15.6

naive → (mapA, mapB, reducezip)

What have we gained?

- If a naive algorithm
  (higher-order function expression) is given
- We can generate automatically different subdivisions and reorderings
- Even if we don't know the hardware details, we can benchmark them, and select the best candidates

≈5 sec

↓

n! candidates

↓

180 ms

Suitable for computations running for
CPU/GPU months/years!

What next?

- This was just one level of the hierarchy
- But the hierarchy is self similar, the same higher-order functions can be used on all levels
- It would be nice if we could fit stencil / sliding window problems into a similar closed system

More about us:

gpu.wigner.mta.hu

https://github.com/Wigner-GPU-Lab

Join us on the Wigner GPU Day

gpuday.com   21-22. June 2018, Budapest

More on this and related projects:

https://github.com/leanil/DataView

https://github.com/leanil/LambdaGen