Optimizing Linear Algebraic Operations for Improved Data-Locality

8th Wigner GPU Day

21 June 2018.

Dániel Berényi

Wigner Research Centre for Physics

András Leitereg, Gábor Lehel

Eötvös Lóránd University

Wigner Research Centre for Physics, Budapest

- GPU Laboratory
- Developer support

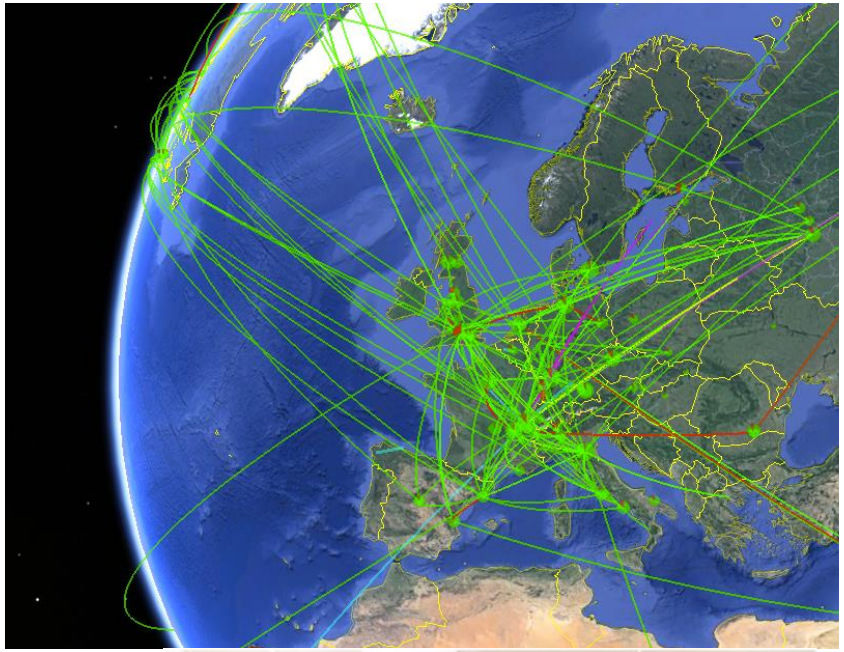
What we face day to day:

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 code written by non-experts, that could utilize the hardware better

Hardware hierarchies



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

Memory hierarchies

Speed

Storage

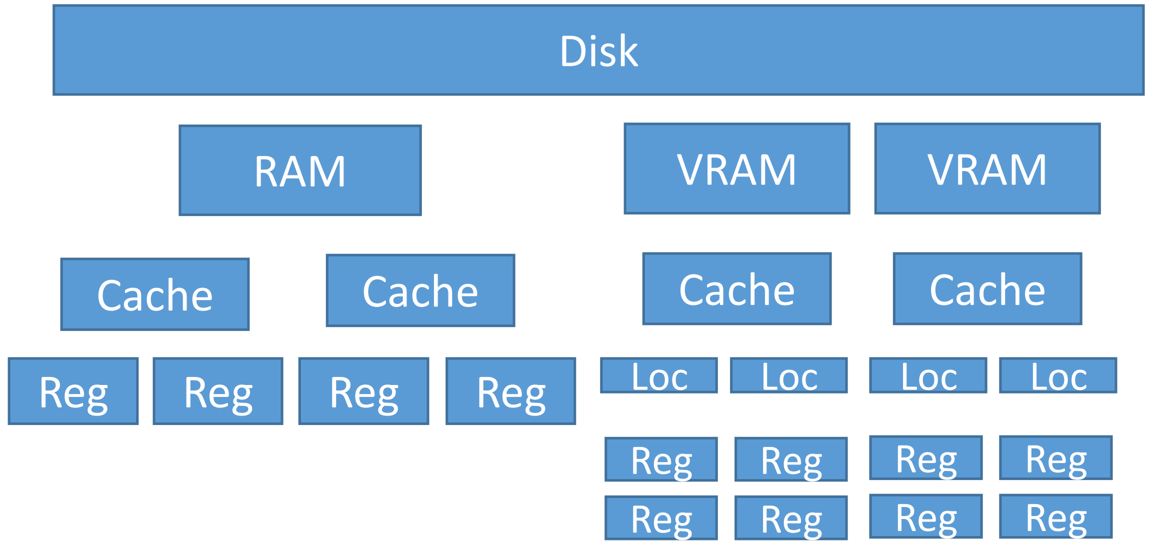
Device memory

Caches

Shared memory

Registers

Size



Specific example: linear algebra

The heart of simulations, neural networks, modeling and much more…
It 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. general tensor contractions?

Boost

Armadillo

Eigen

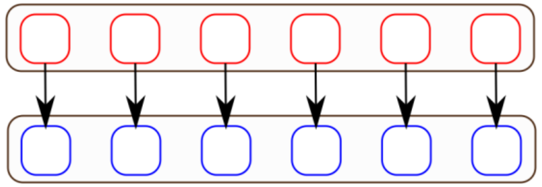
Specific example: linear algebra

Can we get more flexible, yet well optimizable primitives?

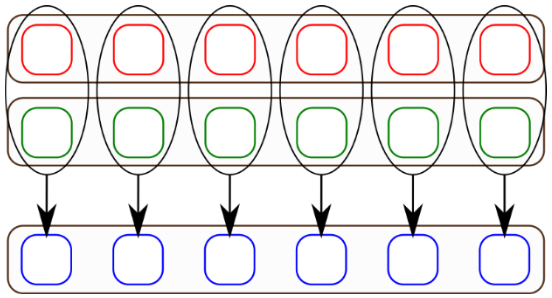
- That cover existing features of linear algebra and more
- Have primitives that are expressive, yet composable
- Automatic tools can be constructed to optimize them

Higher order function primitives

On arrays we may consider the usual primitives:

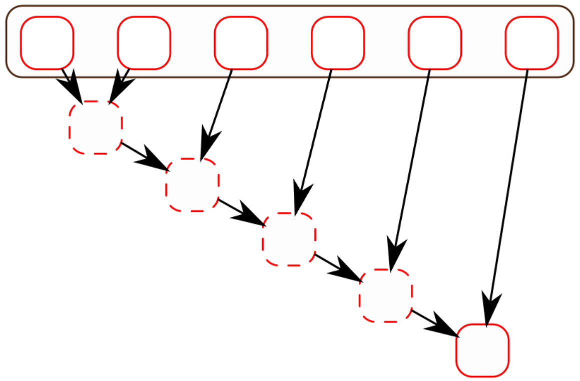


`map :: (a → b) → f a → f b`



`zip :: (a → b → c) → f a → f b → f c`

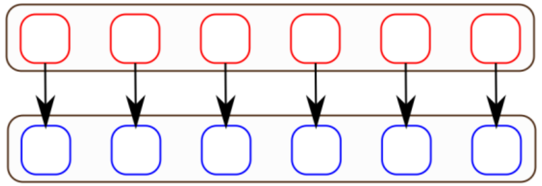
`reduce :: (a → a → a) → f a → a`



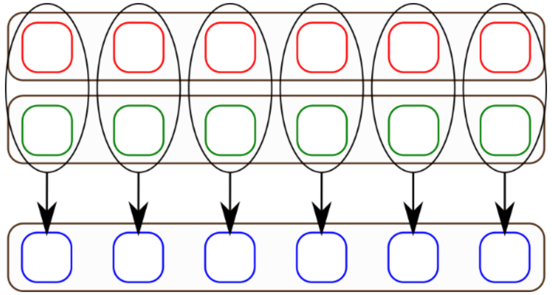
And lets have functions (lambdas) and composition

Higher order function primitives

What happens when we try to compose them?



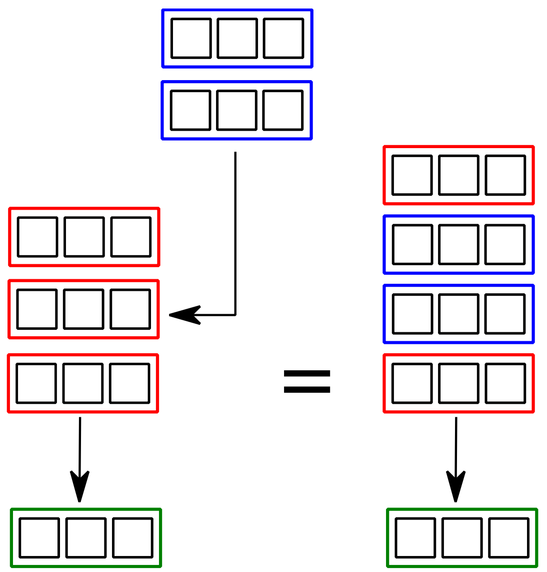
```
map f ∘ map g = map (f ∘ g)
```



```
map f ∘ zip g = ???
```

Well, it seems like we are not closed…

Higher order function primitives



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

nzip is closed under compositions

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

We can also compose arbitrary nzips before the reduce:

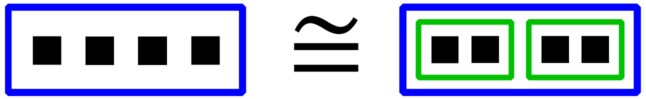
`reducezip ::`

$(b \to b \to b) \to (a_1 \to a_2 \to \ldots \to b) \to (f\ a_1) \to (f\ a_2) \to \ldots \to b$

Higher order function primitives

How can we optimize them?

- Fusion rules (like the composition before)
- Subdivision rules



```
map f A ≅ map (\b → map f b) (subdiv A)
```

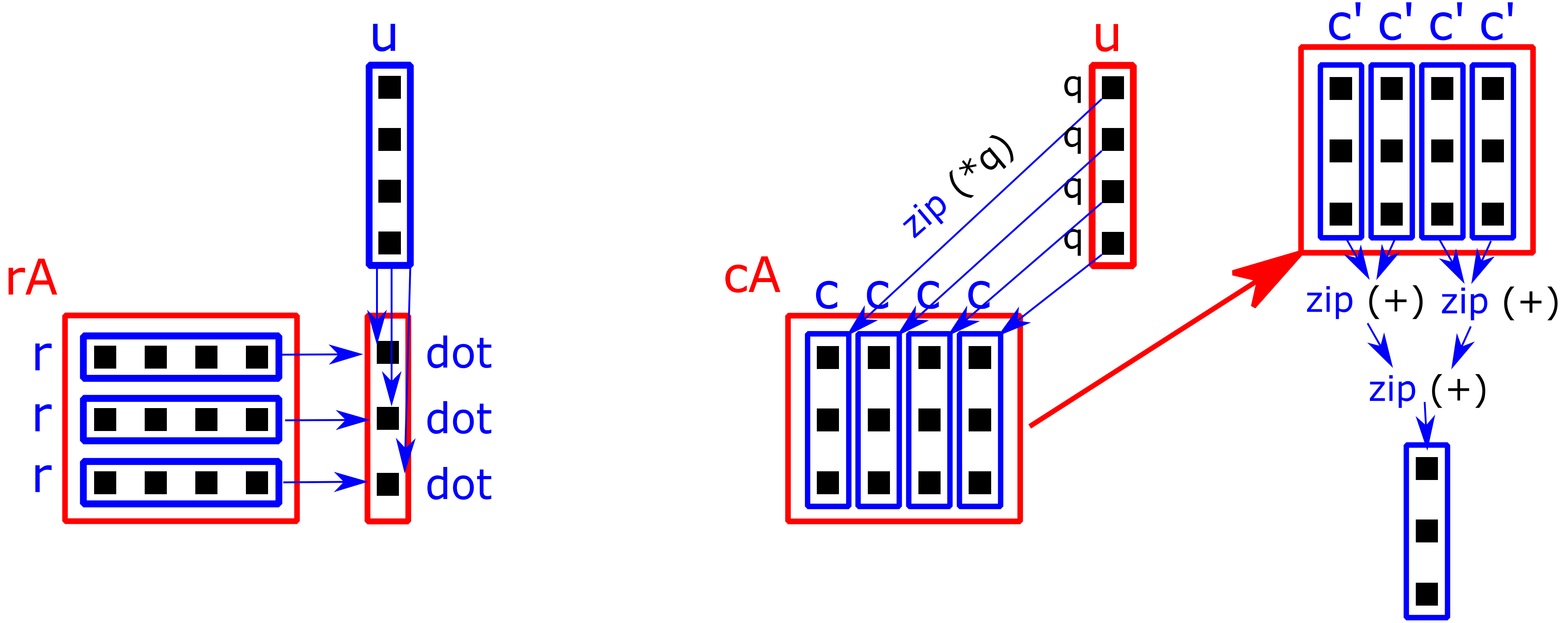
- Exchange rules, like the following:

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

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

Higher order function primitives

Important example: matrix-vector product

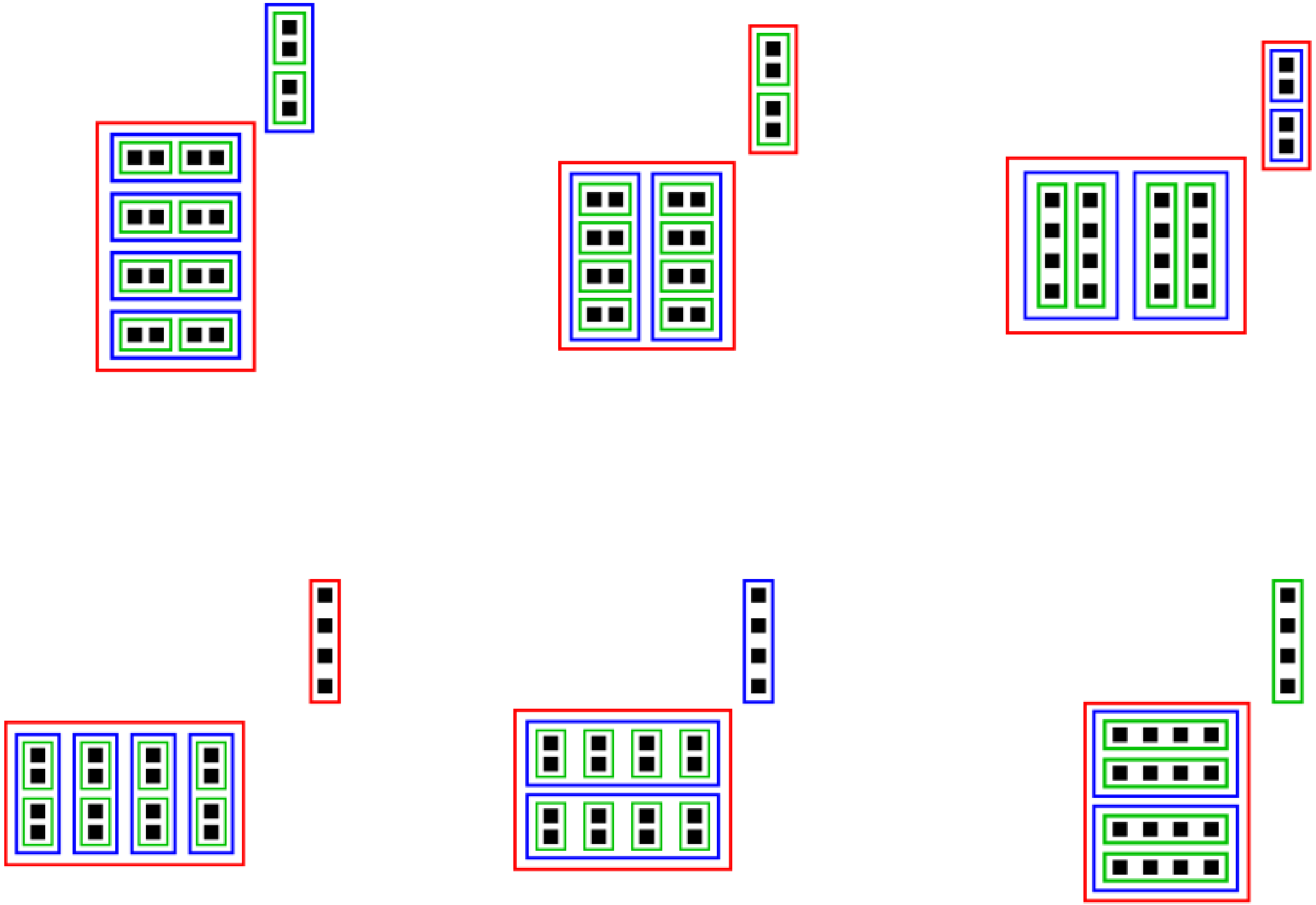


zip (\r -> redozip (+) (*) r u) rA

redozip (zip (+)) (\c q -> zip (*q) c) cA u)

Same result, but different performance!

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?

naive →

HoF ordering			Time [ms]
mapA	reducezip	mapB	450
reducezip	mapA	mapB	1410
mapA	mapB	reducezip	4670
mapB	mapA	reducezip	6050
reducezip	mapB	mapA	13 800
mapB	reducezip	mapA	15 600

What have we gained?

- If a naive algorithm is given (higher-order function expression)
- We can <u>*automatically*</u> generate 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

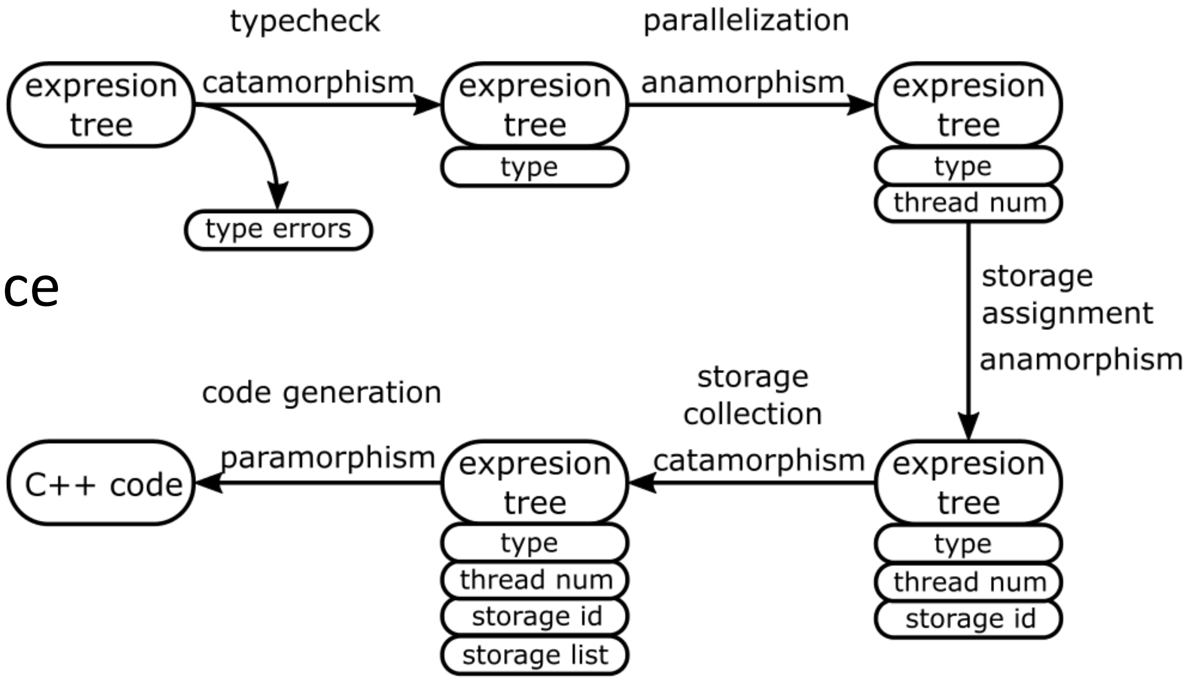
↓

180 ms

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

What is in the background?

We built a compiler in Haskell
using only structured recursion schemes
Optimization is based on pattern-find-and-replace



We have constructed and *proven* the optimization patterns
for the higher-order functions shown earlier.

ComputeCpp™

We generate C++ code for CPUs and GPUs (using SYCL and ComputeCPP)

Future

- We investigated only 1 level of the hierarchy, but it is self-similar
- A cost model based heuristic would scale better than the brute-force n! evaluation
- The operations should be extended to include sliding-window computations (like convolution)

More about the project

The LambdaGen project

https://github.com/leanil/LambdaGen

https://github.com/leanil/DataView

Related publication:

D. Berényi, A. Leitereg, G. Lehel

Towards scalable pattern-based optimization for dense linear algebra

Will appear in: Concurrency and Computation: Practice and Experience

arXiv 1805.04319

This research was supported by:

NKFIH No. K120660 and K123815

A. L. is supported by the UNKP-17-2 New National Excellence Program of the Ministry of Human Capacities

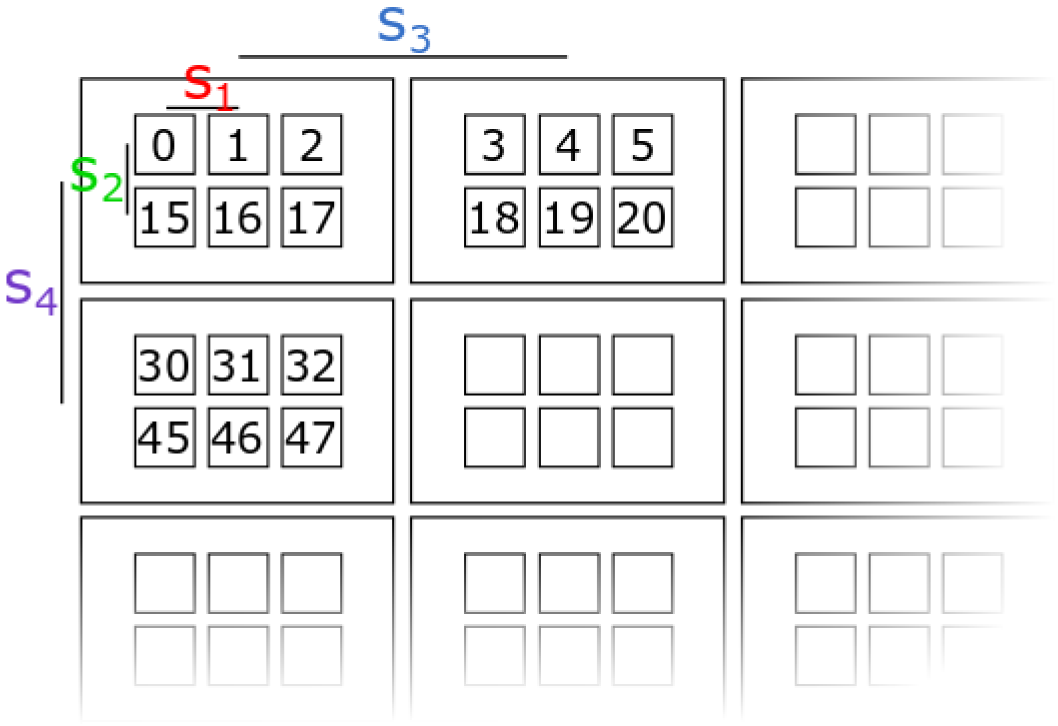
EMBERI ERŐFORRÁSOK MINISZTÉRIUMA

Backup slides

Multidimensional tensors

- We can nest 1 dimensional arrays,
  but can they represent multidimensional <u>and</u> subdivided tensors?
- We can add strides at type level
- We created a C++ View class to handle multi dimensional and strided data

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



The LambdaGen EDSL

```
reduce
    (lam x (lam y (add x y)))
    (zip
        (lam x (lam y (mul x y)))
        u
        v)
```

The generated code

```
auto evaluator(std::map<std::string, double*> bigVectors){
    View<double> s2147482884;
    View<double,Pair<3,1>> s483997720;
    Zip(
        [&](const auto& x){return
            [&](const auto& y){return
                [&](auto& result){result=x*y;};};},
        View<double,Pair<3,1>>(bigVectors.at("u")),
        View<double,Pair<3,1>>(bigVectors.at("v")),
        s483997720);
    Reduce(
        [&](const auto& x){return
            [&](const auto& y){return
                [&](auto& result){result=x+y;};};},
        s483997720,
        s2147482884);
    return s2147482884;
}
```