Petalisp

A Common Lisp Library for Data Parallel Programming

Marco Heisig

16.04.2018

Chair for System Simulation
FAU Erlangen-Nürnberg

Petalisp

The library **Petalisp**[1] is a new approach to data parallel computing.

The Goal: Elegant High Performance Computing

- Programs that are beautiful *and* fast
- A programming model that is safe and productive

Drawbacks:

- Limited to operations on structured data
- Significant run-time overhead

[1] https://github.com/marcoheisig/Petalisp

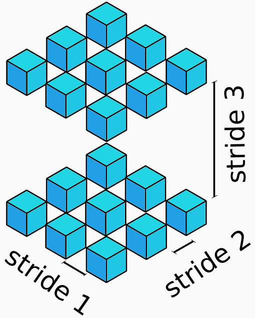
Table of contents

Using Petalisp

Strided Arrays

A **strided array** in n dimensions is a function from elements of the cartesian product of n ranges to a set of Common Lisp objects.

A **range** with the lower bound x_L, the step size s and the upper bound x_U, with $x_L, s, x_U \in \mathbb{Z}$, is the set of integers $\{\, x \in \mathbb{Z} \mid x_L \leq x \leq x_U \;\wedge\; (\exists k \in \mathbb{Z})\, [x = x_L + ks] \,\}$.

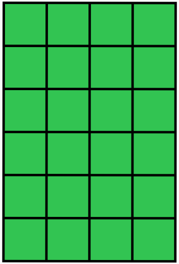


Objects of type `cl:simple-array` are a special case of strided arrays.

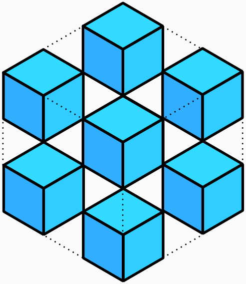
API 1/5 — First Class Index Spaces

To introduce parallelism, Petalisp always operates on index spaces, not on individual array elements.

Notation: `(σ (START [STEP] END) ...)`



`(σ (0 5) (0 3))`



`(σ (1 2 3) (1 2 3) (1 2 3))`

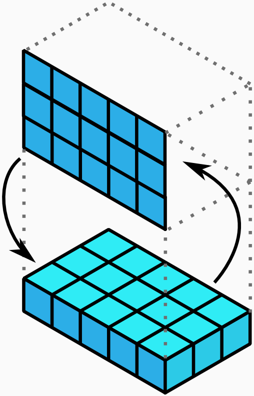
Implementation detail: Petalisp can compute the union, difference and intersection of arbitrary index spaces.

API 2/5 — Transformations

A transformation is an affine-linear mapping from indices to indices.

Notation: `(τ (INDEX ...)  (EXPRESSION ...))`

```
(τ (i j)
   (0 j (- i 4)))
```



```
(τ (0 j k)
   ((+ k 4) j))
```

Implementation detail: Petalisp can compute the inverse and composition of arbitrary transformations.

API 3/5 — Moving Data

The -> operator allows to select, transform or broadcast data.

```
(-> 0 (σ (0 9) (0 9)))  ; a 10 × 10 array of zeros

(-> #(2 3) (σ (0 0)))   ; the first element only

(-> A (τ (i j) (j i)))  ; transposing A
```

Admittedly, -> is a mediocre function name. Better suggestions are most welcome!

API 4/5 — Combining Data

The `fuse` and `fuse*` operator combine multiple arrays into one.

For `fuse`, the arguments must be non-overlapping. For `fuse*`, the value of the rightmost array takes precedence on overlap.

```
(defvar B (-> #(2) (τ (i) ((1+ i)))))

(fuse  #(1)   B) ; equivalent to (-> #(1 2))

(fuse  #(1 3) B) ; an error!

(fuse* #(1 3) B) ; equivalent to (-> #(1 2))
```

API 5/5 — Parallel Operations

The final piece: Application of Common Lisp functions to strided arrays.

- The function α is basically `cl:map` for strided arrays.
- The function β is basically `cl:reduce` applied to the *last* dimension of a strided array.

```
(α #'+ 2 3)       ; adding two numbers
(α #'+ A B C)     ; adding three arrays element-wise

(β #'+ #(2 3))    ; adding two numbers
(defvar B #2A((1 2 3) (4 5 6)))
(β #'+ B)         ; summing the rows of B
```

Remark: No guarantees are made about when and how often the functions passed to α and β are invoked.

API Summary

All core functions at a glance:

- Index spaces, e.g. `(σ (a b))`
- Transformations, e.g. `(τ (x y) (y (- x)))`
- Data motion, e.g. `(-> A (σ (2 5)))`
- Data combination, e.g. `(fuse* A B C)`
- Parallel map, e.g. `(α #'* A B)`
- Parallel reduce, e.g. `(β #'+ A)`

This API is purely functional and declarative.

But how do we obtain values?

API 6/5 — Triggering Evaluation

Petalisp provides two functions to trigger evaluation.

The `compute` function converts strided arrays into regular Common Lisp arrays.

```
(compute (-> 0.0 (σ (0 1))))       => #(0.0 0.0)

(defvar A #(1 2 3))

(compute (β #'+ A))                => 6

(compute (-> A (τ (i) ((- i)))))  => #(3 2 1)
```

Remark: There is also a `schedule` function for asynchronous evaluation.

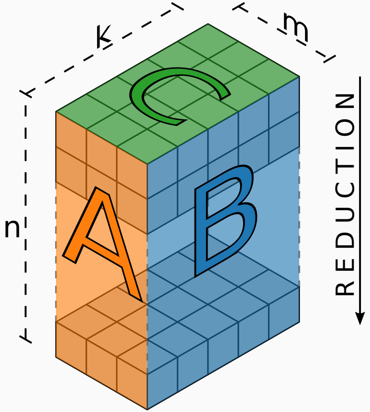
Example: Matrix Multiplication

The mathematical definition

$$C_{ij} = \sum_{p=1}^{n} A_{ip} B_{pj}$$

The corresponding Petalisp code

```
(β #'+
   (α #'*
      (-> A (τ (m n) (m 1 n)))
      (-> B (τ (n k) (1 k n)))))
```

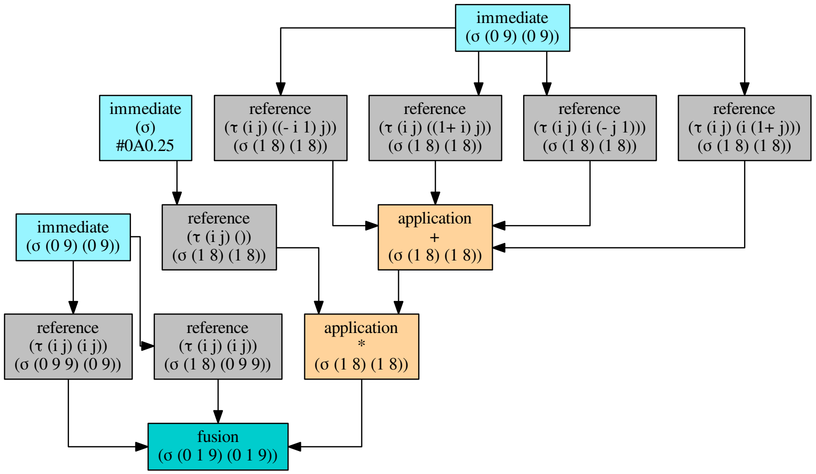


Implementation

Lazy Arrays are Data Flow Graphs

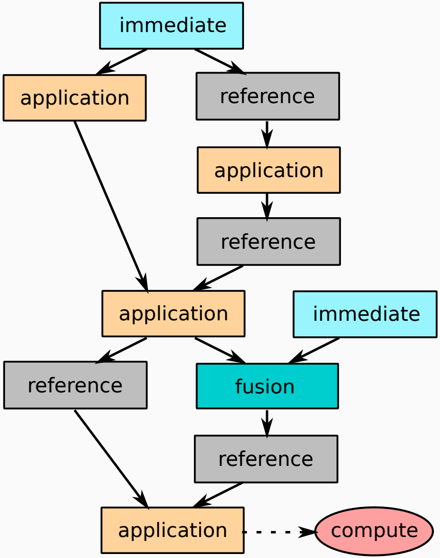
```
(jacobi u 1) => #<strided-array-fusion t (σ (0 9) (0 9))>
```

Internal representation:



From Arrays to Executable Code

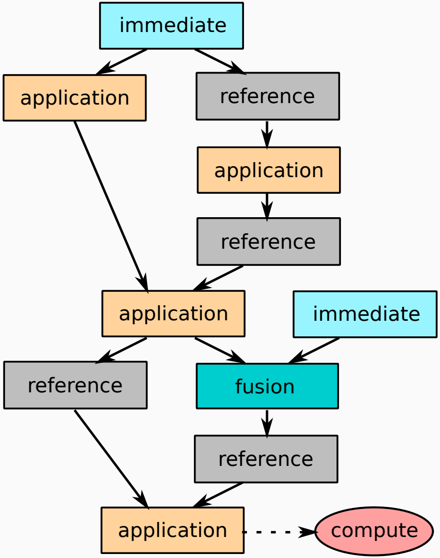
How do we execute a graph?



From Arrays to Executable Code

How do we execute a graph?

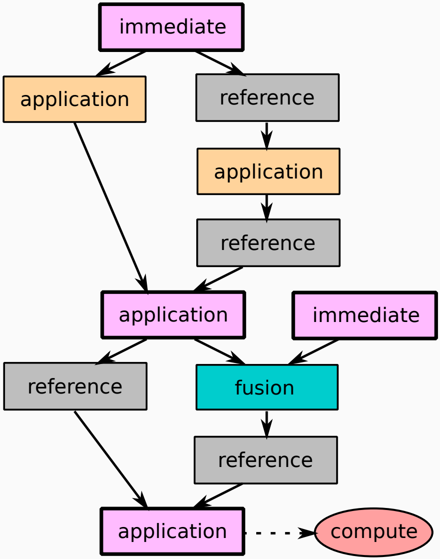
1. Determine critical nodes



From Arrays to Executable Code

How do we execute a graph?

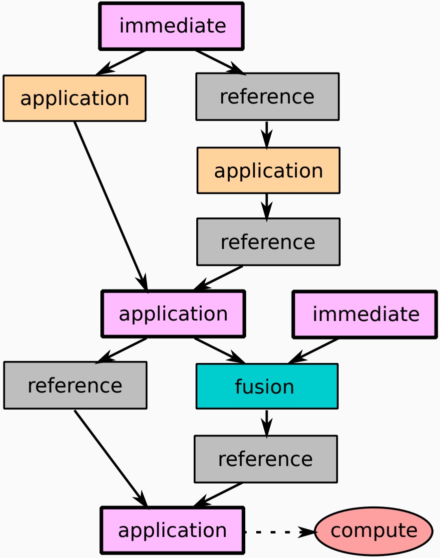
1. Determine critical nodes



From Arrays to Executable Code

How do we execute a graph?

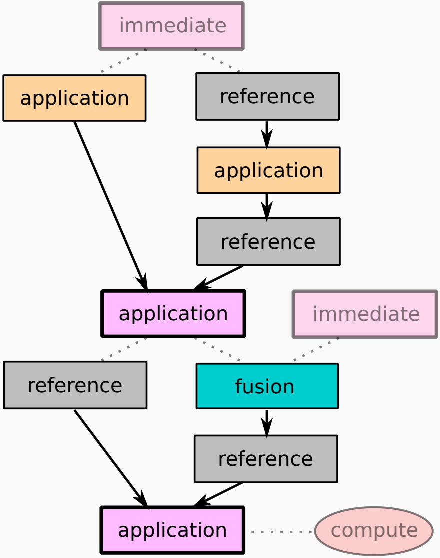
1. Determine critical nodes
2. Determine subtrees



From Arrays to Executable Code

How do we execute a graph?

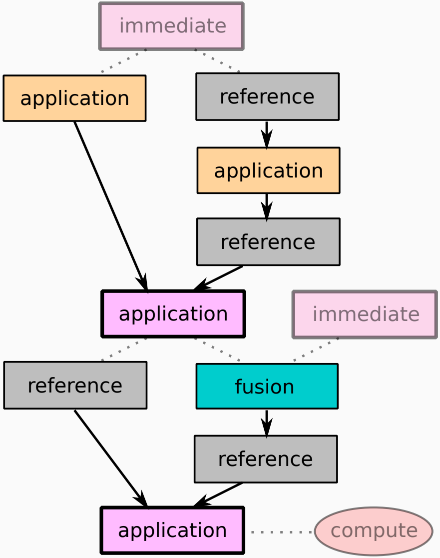
1. Determine critical nodes
2. Determine subtrees



From Arrays to Executable Code

How do we execute a graph?

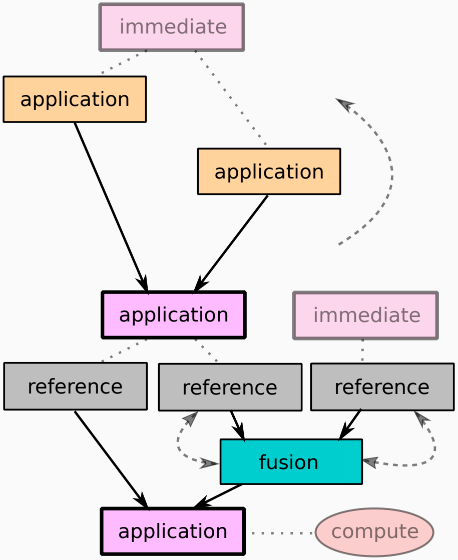
1. Determine critical nodes
2. Determine subtrees
3. Lift references



From Arrays to Executable Code

How do we execute a graph?

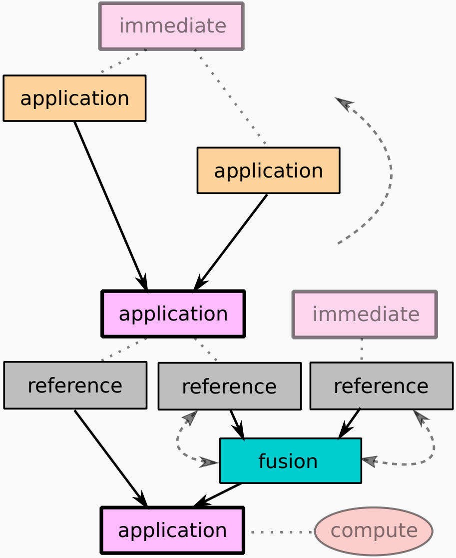
1. Determine critical nodes
2. Determine subtrees
3. Lift references



From Arrays to Executable Code

How do we execute a graph?

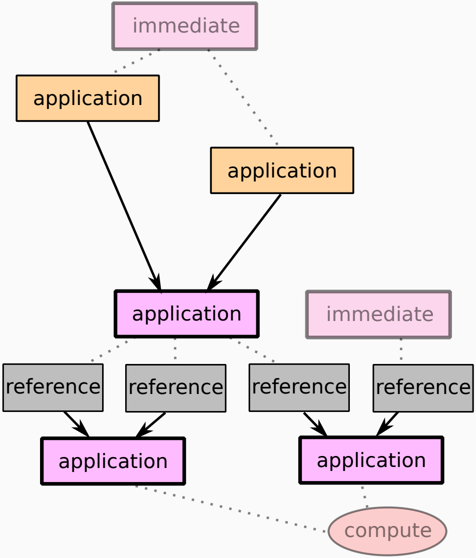
1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions



From Arrays to Executable Code

How do we execute a graph?

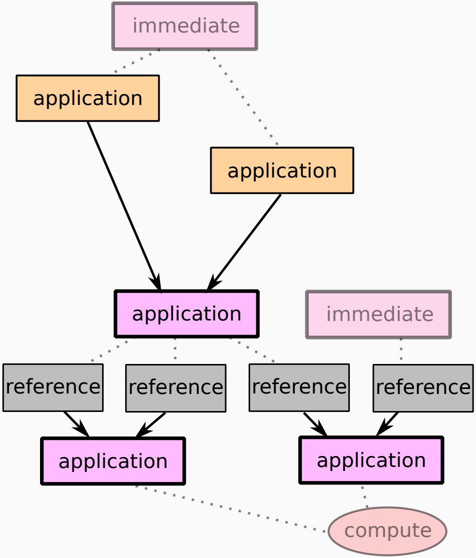
1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions



From Arrays to Executable Code

How do we execute a graph?

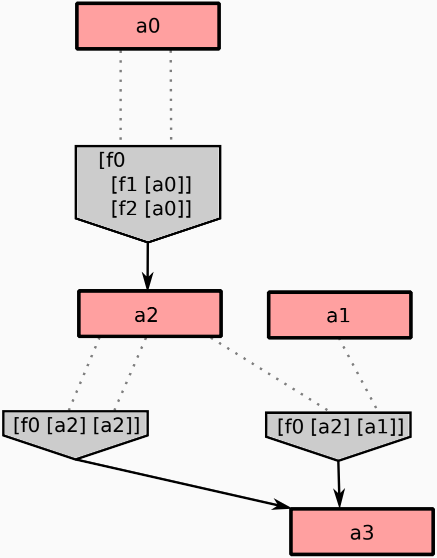
1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions
5. Construct kernels



From Arrays to Executable Code

How do we execute a graph?

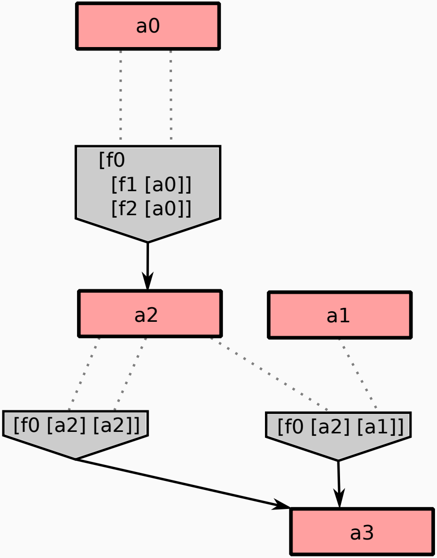
1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions
5. Construct kernels



From Arrays to Executable Code

How do we execute a graph?

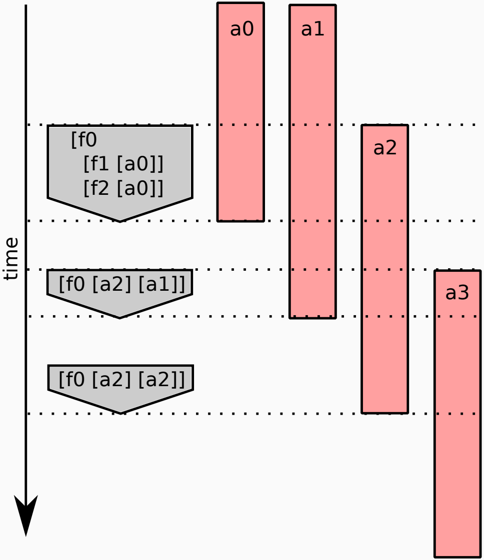
1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions
5. Construct kernels
6. Schedule & allocate



From Arrays to Executable Code

How do we execute a graph?

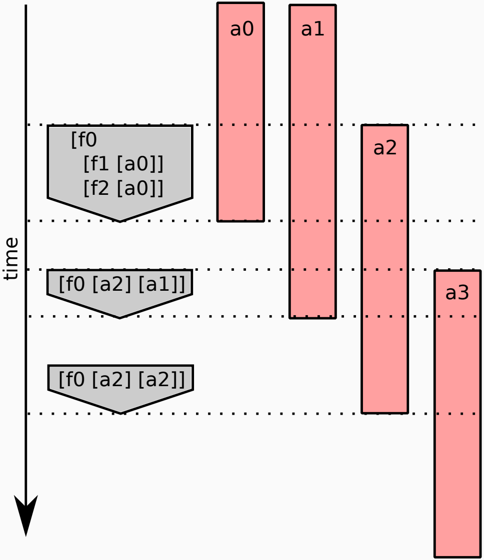
1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions
5. Construct kernels
6. Schedule & allocate



From Arrays to Executable Code

How do we execute a graph?

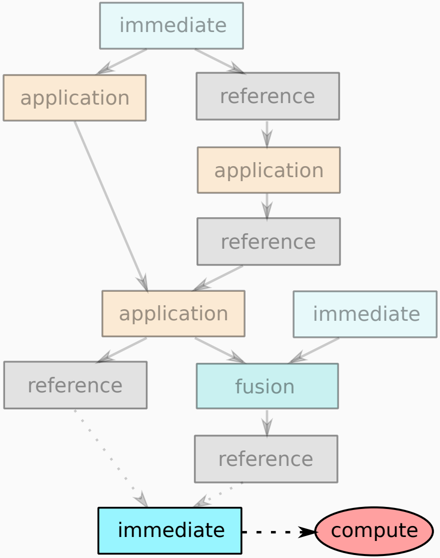
1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions
5. Construct kernels
6. Schedule & allocate
7. Compile & execute



From Arrays to Executable Code

How do we execute a graph?

1. Determine critical nodes
2. Determine subtrees
3. Lift references
4. Eliminate fusions
5. Construct kernels
6. Schedule & allocate
7. Compile & execute
8. **Done!**



Performance

On Performance

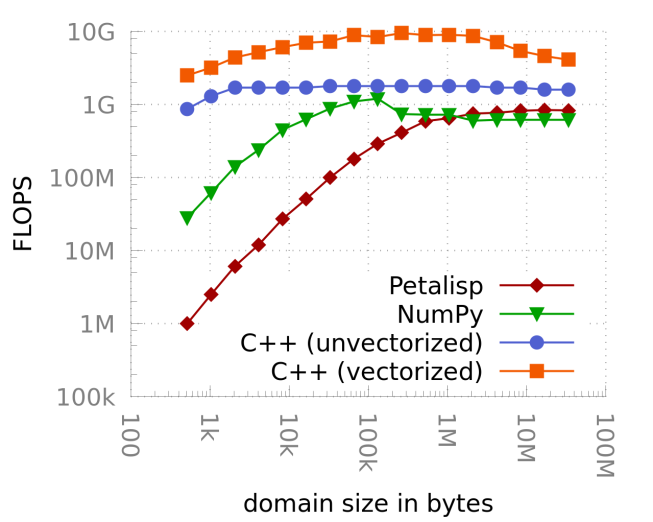
The long-term goal of Petalisp is to provide a programming model for Petascale (10^{15} operations per second) systems.

The constant, high overhead of analysis and JIT-compilation seems to be at odds with this goal.

However:

- Do not underestimate the power of memoization, hash-consing and CLOS wizardry.
- Scheduling can often be done asynchronously.
- Petalisp's analysis is independent of the problem size.

Jacobi's method: Python vs. C++ vs. Petalisp



Hardware: Intel Xeon E3-1275 CPU 3.6GHz

Conclusions

Conclusions

Main Result: Our compilation strategy is feasible, with just about 10 – 500 microseconds overhead when calling `compute`.

Benefits:

- Clean separation between notation and execution.
- Unprecedented potential for optimization.
- Already faster than NumPy.

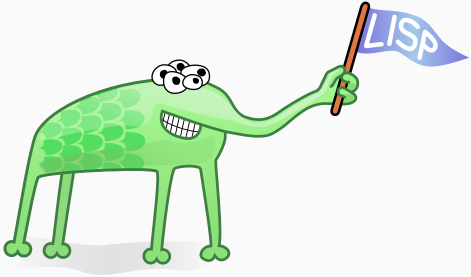
. . . all in just about 5000 lines of maintainable code.

Future Work

My preliminary roadmap for the next years:

- More applications (simulations, image processing, machine learning)
- API finalization
- Sophisticated Scheduling
- Better Shared-Memory Parallelization
- Auto-Vectorization
- Distributed Parallelization
- ...
- Make this a PhD thesis

Thank you!



Questions or remarks?