Petalisp

A Common Lisp Library for Data Parallel Programming

Marco Heisig
16.04.2018

Chair for System Simulation
FAU Erlangen-Nürnberg
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](https://github.com/marcoheisig/Petalisp)
Table of contents

1. Using Petalisp
2. Implementation
3. Performance
4. Conclusions
Using Petalisp
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
\[
\{ x \in \mathbb{Z} \mid x_L \leq x \leq x_U \land (\exists k \in \mathbb{Z}) [x = x_L + ks] \}.
\]

Objects of type `cl:simple-array` are a special case of strided arrays.
To introduce parallelism, Petalisp always operates on index spaces, not on individual array elements.

**Notation:** \((\sigma (\text{START} [\text{STEP}] \text{END}) \ldots)\)

**Implementation detail:** Petalisp can compute the union, difference and intersection of arbitrary index spaces.
A transformation is an affine-linear mapping from indices to indices.  

**Notation:** $\tau (\text{INDEX ... }) (\text{EXPRESSION ... })$

**Implementation detail:** Petalisp can compute the inverse and composition of arbitrary transformations.
The \( \rightarrow \) operator allows to select, transform or broadcast data.

\[
(-\rightarrow 0 (\sigma (0 \ 9) (0 \ 9))) \; ; \text{a } 10 \times 10 \text{ array of zeros}
\]

\[
(-\rightarrow #(2 \ 3) (\sigma (0 \ 0))) \; ; \text{the first element only}
\]

\[
(-\rightarrow A (\tau (i \ j) (j \ i))) \; ; \text{transposing } A
\]

Admittedly, \( \rightarrow \) is a mediocre function name. Better suggestions are most welcome!
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.

\[(\text{defvar } B \rightarrow #(2) (\tau (i) ((1+ i))))\]

\[(\text{fuse } #1B); \text{ equivalent to } \rightarrow #(1 2))\]

\[(\text{fuse } #13B); \text{ an error!}\]

\[(\text{fuse* } #13B); \text{ equivalent to } \rightarrow #(1 2))\]
The final piece: Application of Common Lisp functions to strided arrays.

- The function $\alpha$ is basically `cl:map` for strided arrays.
- The function $\beta$ is basically `cl:reduce` applied to the last dimension of a strided array.

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

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

Remark: No guarantees are made about when and how often the functions passed to $\alpha$ and $\beta$ are invoked.
All core functions at a glance:

- Index spaces, e.g. \((\sigma (a \ b))\)
- Transformations, e.g. \((\tau (x \ y) (y (- x)))\)
- Data motion, e.g. \((\rightarrow A (\sigma (2 \ 5)))\)
- Data combination, e.g. \((\text{fuse}\ A \ B \ C)\)
- Parallel map, e.g. \((\alpha \ #'* A \ B)\)
- Parallel reduce, e.g. \((\beta \ #'+ A)\)

This API is purely functional and declarative.

But how do we obtain values?
Petalisp provides two functions to trigger evaluation.

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

\[
\begin{align*}
\text{(compute } (-\rightarrow 0.0 (\sigma (0\ 1))))) & \quad \Rightarrow \#(0.0\ 0.0) \\
\text{(defvar A } \#(1\ 2\ 3))
\end{align*}
\]

\[
\begin{align*}
\text{(compute } (\beta \ #'+ A)) & \quad \Rightarrow 6 \\
\text{(compute } (-\rightarrow A (\tau (i) ((- i)))))) & \quad \Rightarrow \#(3\ 2\ 1)
\end{align*}
\]

**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

\[
\beta \ #'+ \\
\alpha \ #'* \\
\quad (\rightarrow A \ (\tau \ (m \ n) \ (m \ 1 \ n))) \\
\quad (\rightarrow B \ (\tau \ (n \ k) \ (1 \ k \ n))))
\]
Implementation
Lazy Arrays are Data Flow Graphs

\[(\text{jacobi } u \ 1) \Rightarrow \#<\text{strided-array-fusion } t \ (\sigma \ (0 \ 9) \ (0 \ 9))>\]

Internal representation:
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!
From Arrays to Executable Code

How do we execute a graph?

1. Determine critical nodes
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

1. Determine critical nodes
2. Determine subtrees
How do we execute a graph?

1. Determine critical nodes
2. Determine subtrees
How do we execute a graph?

1. Determine critical nodes
2. Determine subtrees
3. Lift references
How do we execute a graph?

1. Determine critical nodes
2. Determine subtrees
3. Lift references
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
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!
How do we execute a graph?

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

```
[f0 [a2] [a2]]
[f0 [a2] [a1]]
[f0 [a2] [a2]]
```
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
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
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
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.
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?