Simple Optimizations for Applicative Array Programs for Graphics Processors

Bradford Larsen
Tufts University
blarsen@cs.tufts.edu

This work was supported in part by the NASA Space Grant Graduate Fellowship and NSF grants IIS-0082577 and OCI-0749125

GPUs are powerful, but difficult to program

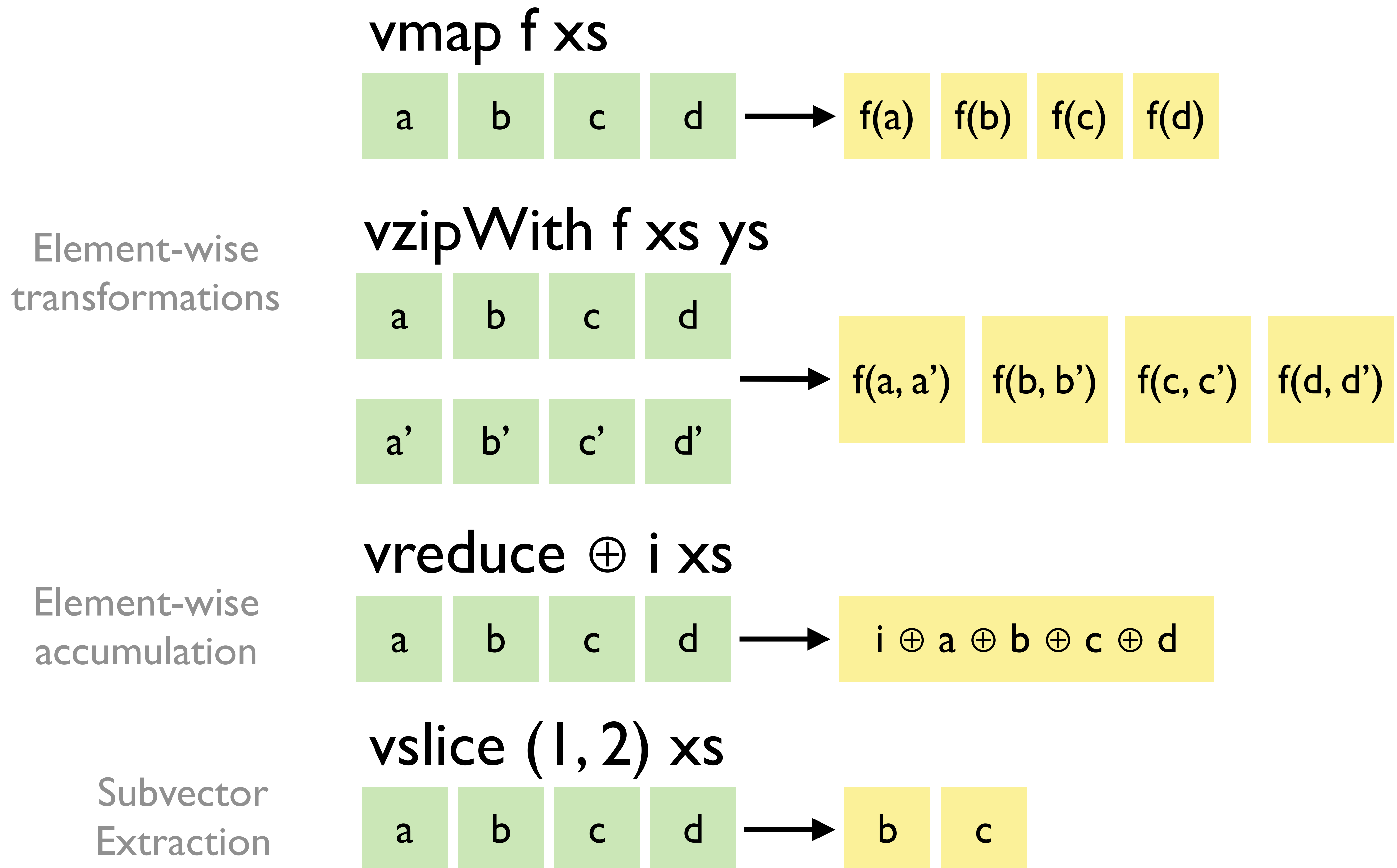
- 1 TFLOP/s on modern GPUs; several times greater than CPUs
- Lots of code for simple operations:

```
float sum = 0;
for (int i = 0; i < n; i += 1)
    sum += arr[i];
```

  in C takes ~150 lines of CUDA

- GPU code is data-parallel: you must decompose the problem's data

Applicative array programming allows easy GPU use



The Barracuda language supports these primitives on the GPU

- Applicative: no side effects
- Compositional: primitives can be freely nested
- Deeply embedded within Haskell
- Functions on vectors, matrices, and scalars are the unit of compilation

Barracuda code resembles Haskell code on lists

Haskell lists

```
rmse :: [Float] -> [Float] -> Float
rmse x y = sqrt (sumDiff / fromIntegral (length x))
  where
    sumDiff = sum (map (^2) (zipWith (-) x y))
```

Barracuda

```
rmse :: VExp Float -> VExp Float -> SExp Float
rmse x y = sqrt (sumDiff / fromIntegral (vlength x))
  where
    sumDiff = vsum (vmap (^2) (vzipWith (-) x y))
```

Barracuda code resembles Haskell code on lists

Haskell lists

```
rmse :: [Float] -> [Float] -> Float
rmse x y = sqrt (sumDiff / fromIntegral (length x))
  where
    sumDiff = sum (map (^2) (zipWith (-) x y))
```

Barracuda

```
rmse :: VExp Float -> VExp Float -> SExp Float
rmse x y = sqrt (sumDiff / fromIntegral (vlength x))
  where
    sumDiff = vsum (vmap (^2) (vzipWith (-) x y))
```

Barracuda code works on GPU vectors, not lists

Barracuda code resembles Haskell code on lists

Haskell lists

```
rmse :: [Float] -> [Float] -> Float
rmse x y = sqrt (sumDiff / fromIntegral (length x))
  where
    sumDiff = sum (map (^2) (zipWith (-) x y))
```

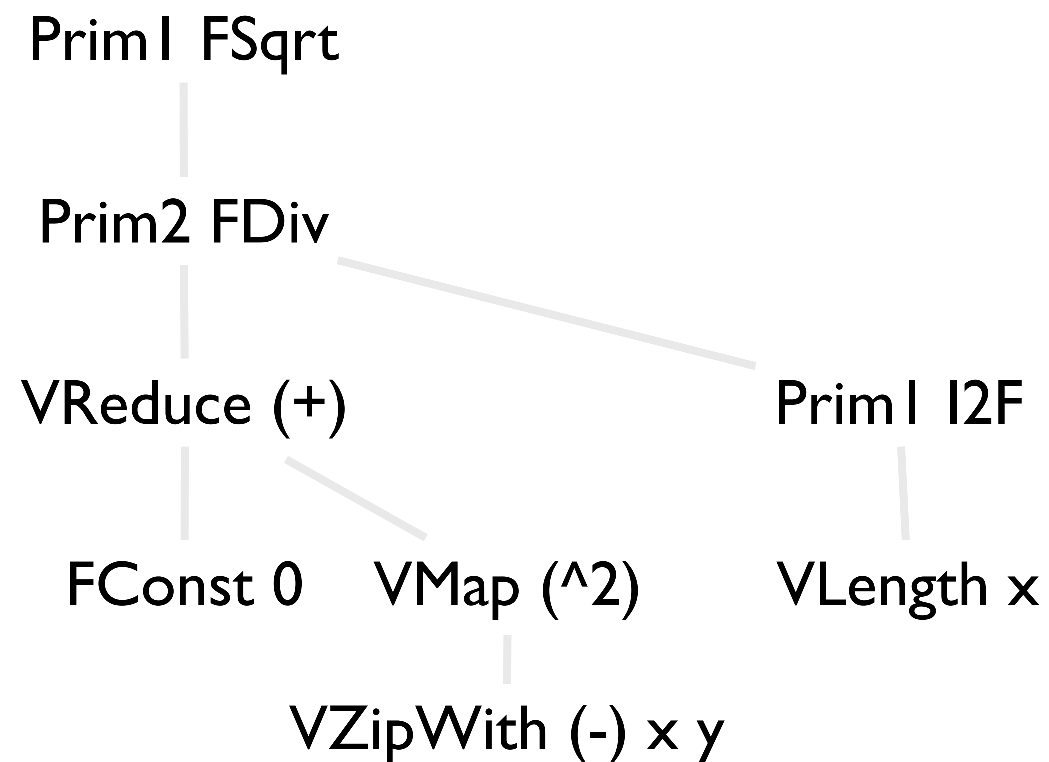
Barracuda

```
rmse :: VExp Float -> VExp Float -> SExp Float
rmse x y = sqrt (sumDiff / fromIntegral (vlength x))
  where
    sumDiff = vsum (vmap (^2) (vzipWith (-) x y))
```

Barracuda functions are named differently

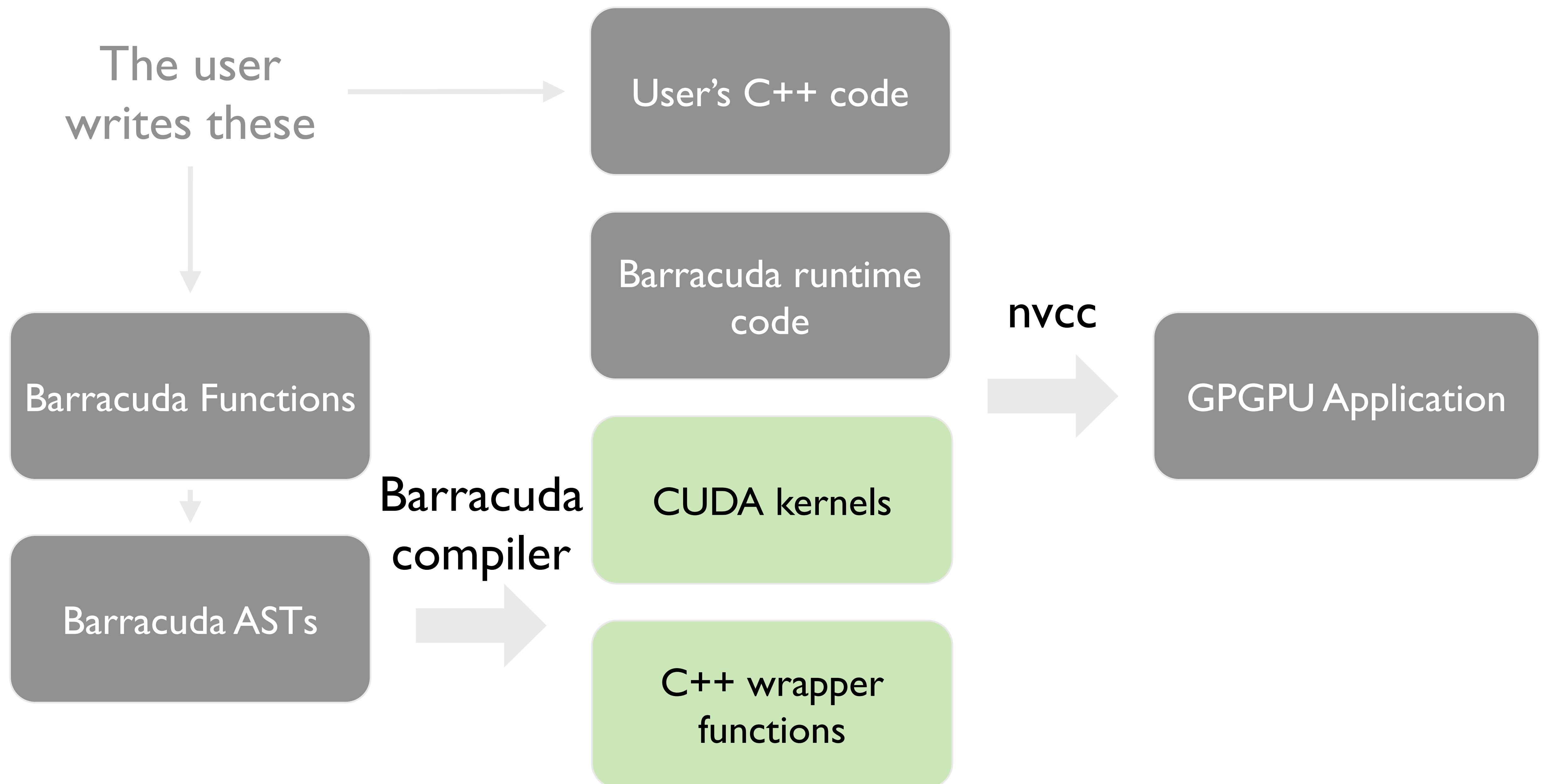
Barracuda functions construct abstract syntax trees

```
rmse:: VExp Float -> VExp Float -> SExp Float
rmse x y = sqrt (sumDiff / fromIntegral (vlength x))
  where sumDiff = vsum (vmap (^2) (vzipWith (-) x y))
```

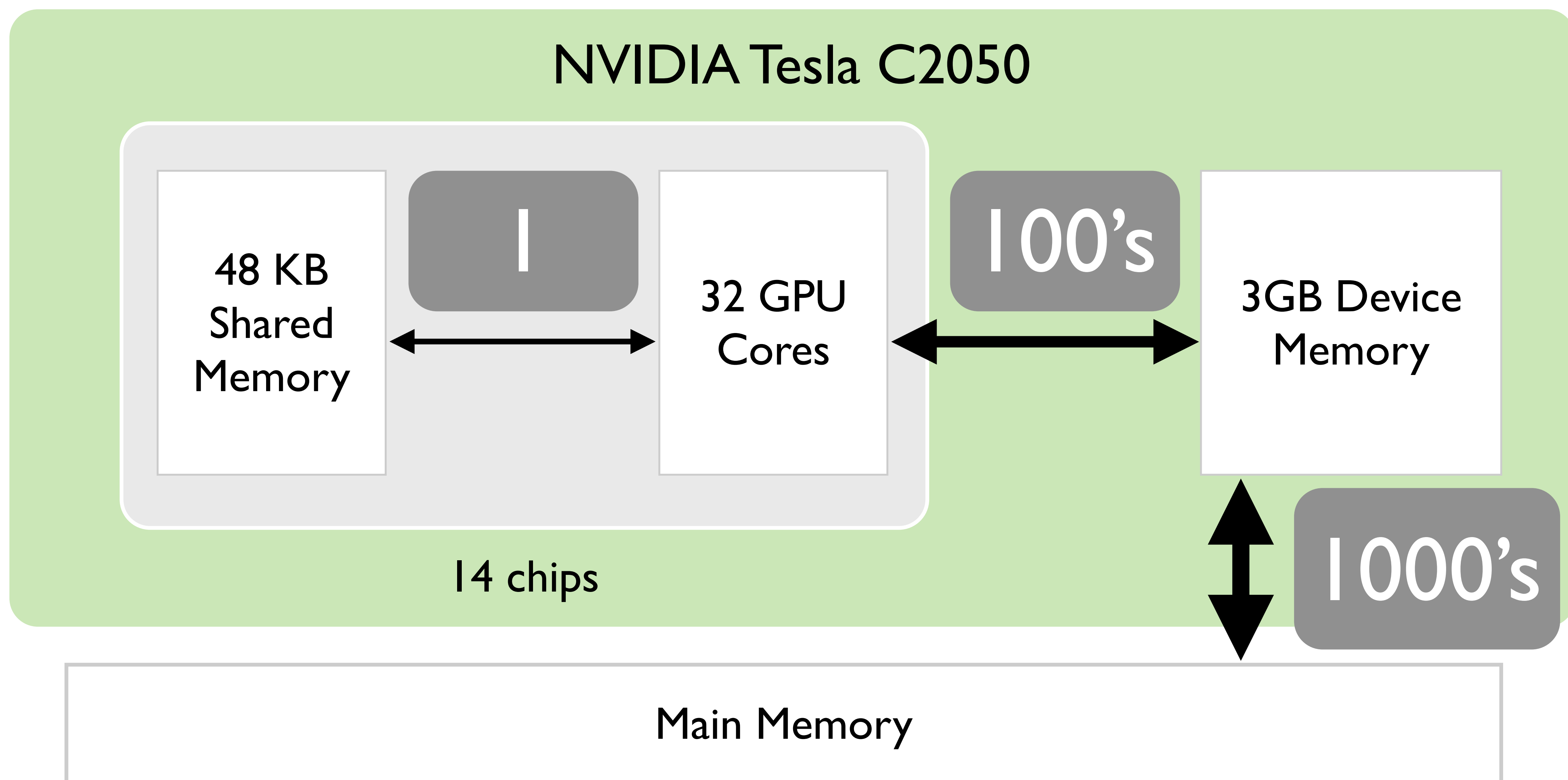


I.e., Barracuda is *deeply embedded* within Haskell

Barracuda ASTs are compiled into optimized CUDA code



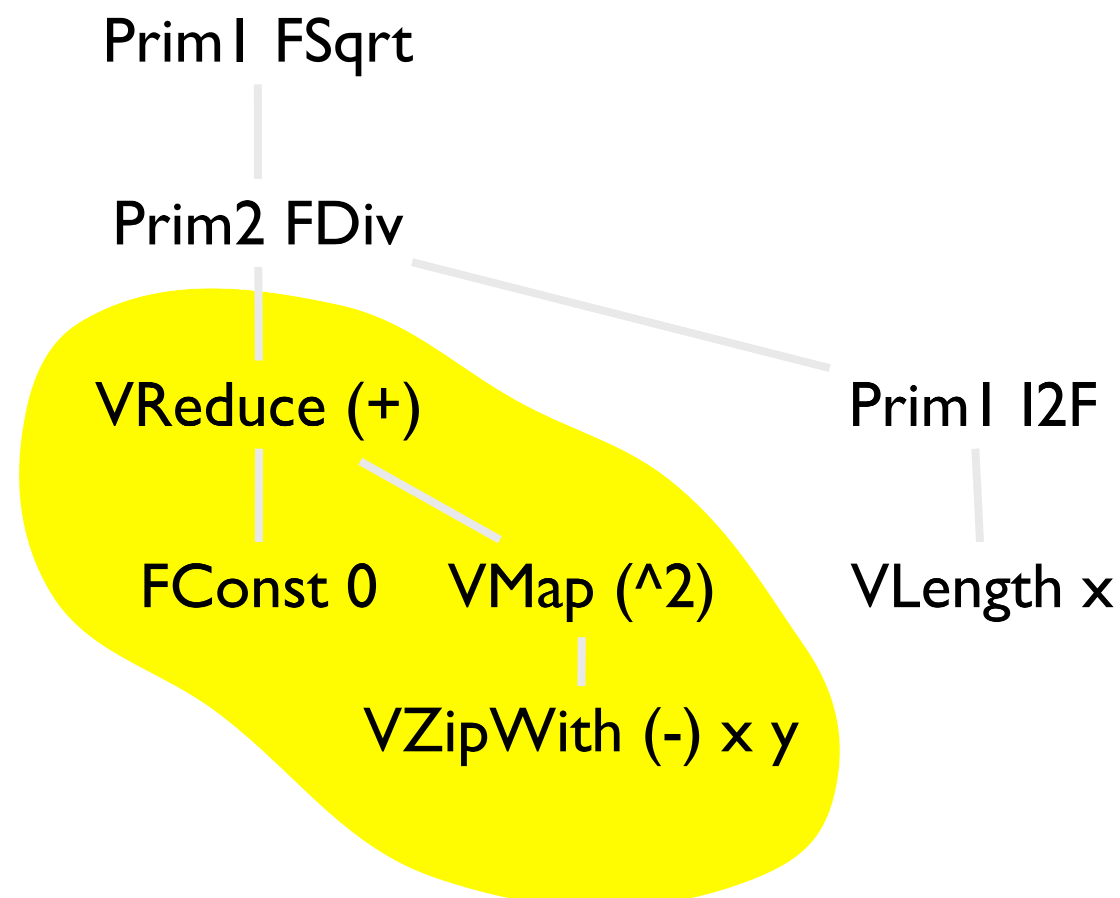
Efficient GPU code exploits the memory hierarchy



Nested array expressions are potentially troublesome

```
rmse:: VExp Float -> VExp Float -> SExp Float
rmse x y = sqrt (sumDiff / fromIntegral (vlength x))
  where sumDiff = vsum (vmap (^2) (vzipWith (-) x y))
```

Naive compilation uses temporaries, multiple passes over data



CUDA computes on elements, not arrays

CUDA code is data-parallel: *kernels* describe what happens at one location.

Array indexing laws allow for fusion:

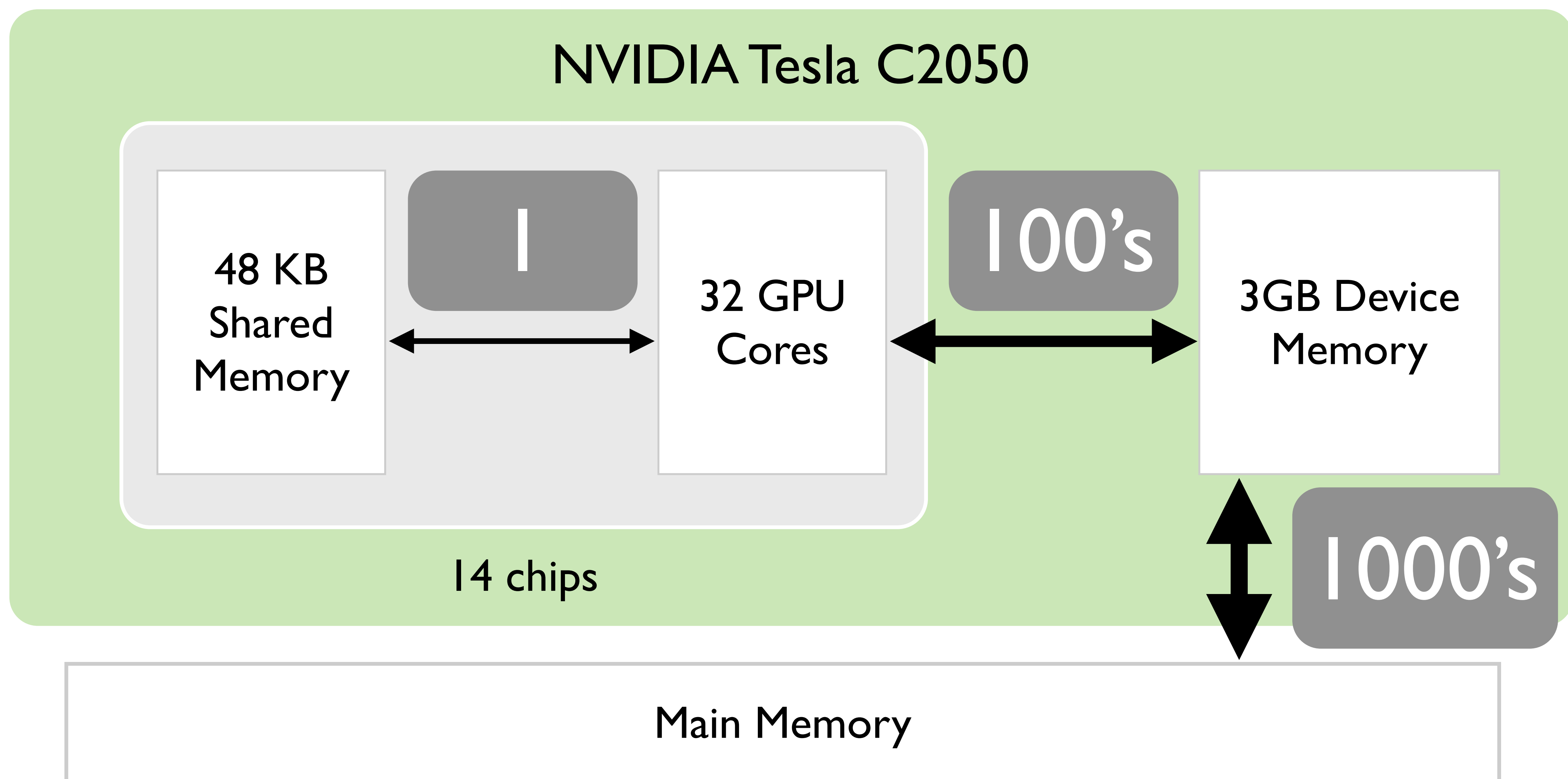
```
(vmap f xs)!i             =   f (xs!i)

(vzipWith f xs ys)!i      =   f (xs!i) (ys!i)

(vslice (b, e) xs)!i      =   xs!(e - b + i)
```

Barracuda always applies the array indexing laws

Array fusion comes naturally during codegen, e.g.:

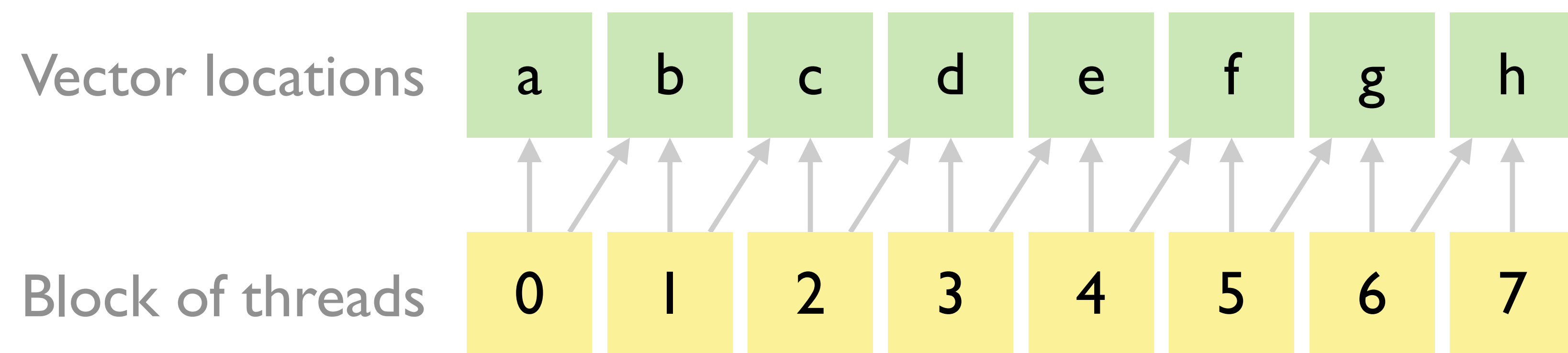
```
(vmap f (vmap g xs))!i                    →  f (g (xs!i))
vmap f (vzipWith g xs ys)!i               →  f (g (xs!i) (ys!i))
(vslice (b, e) (vmap f xs))!i             →  f (xs!(e - b + i))
(vmap f (vslice (b, e) xs))!i             →  f (xs!(e - b + i))
(vslice (b, e) (vslice (b’, e’) xs))!i    →  xs!(e - b + e’ - b’ + i)
```

Efficient GPU code exploits the memory hierarchy



Stencil operations involve redundant reads

A data-parallel CUDA *kernel* is run by many *threads* on the 14 GPU chips.

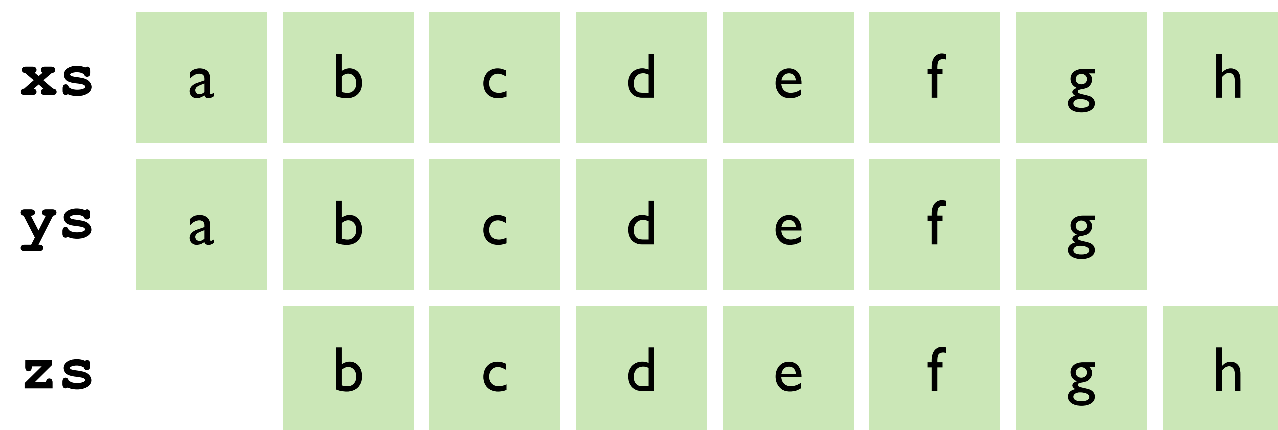


Stencil operations involve array elements in a neighborhood, resulting in several threads reading the same elements.

Barracuda automatically uses shared memory when useful

When multiple array subexpressions *overlap*, there is read redundancy, e.g.:

```
as = vzipWith (-) zs ys
ys = vslice (0, 6) xs
zs = vslice (1, 7) xs
```



Elements b–g are read twice in the computation of **as**

Use of shared memory is only useful when

- array elements are read at least two times;
- it is known at compile-time that elements are read multiple times; and
- there are enough elements to amortize the added indexing costs.

Shared memory optimization examples

```
as = vzipWith (-) zs ys
ys = slice (0, 1022) xs
zs = slice (1, 1023) xs
```

There are enough elements

```
as = vzipWith (-) zs ys
ys = slice (0, 511) xs
zs = slice (512, 1023) xs
```

No elements are read multiple times

```
as = vzipWith (-) zs ys
ys = slice (0, 511) xs
```

No elements are read multiple times

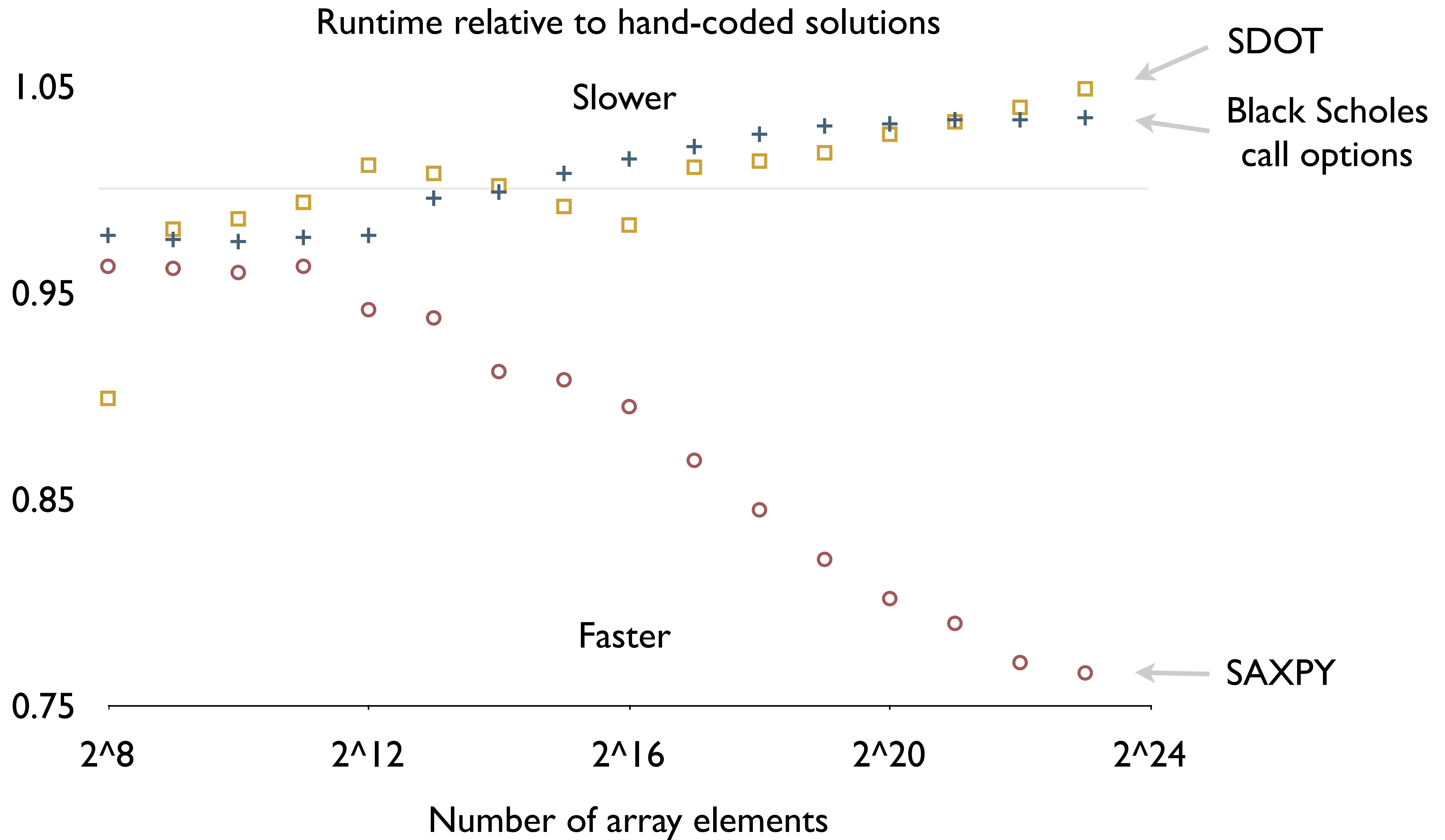
```
as = vzipWith (-) zs ys
ys = slice (0, 1022) xs
zs = slice (1, vlength xs) xs
```

Slices use only constant and vector length expressions; there are enough elements

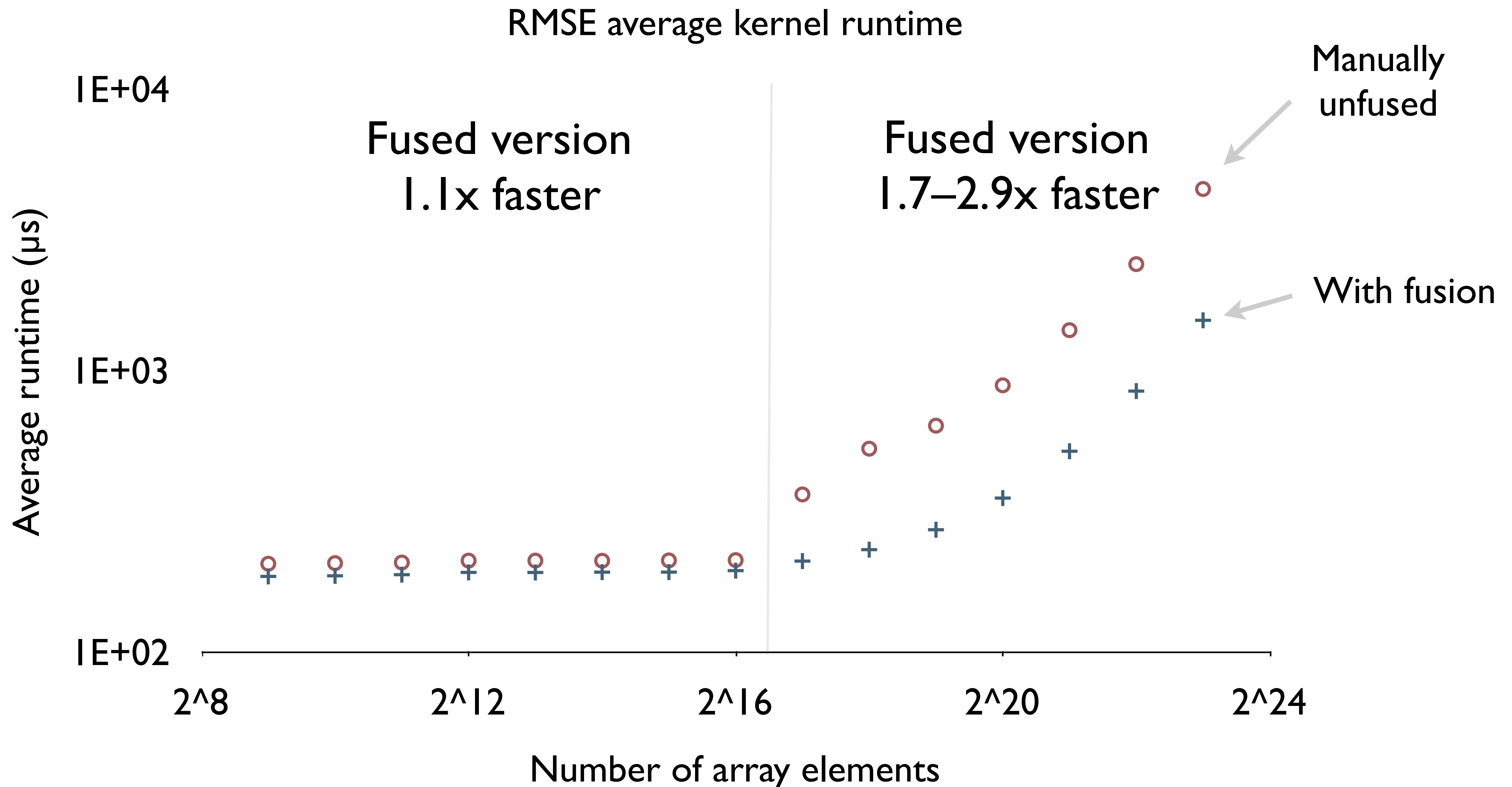
A mix of existing and new benchmarks was used

- BLAS operations, Black-Scholes seen in Lee et al. (2009) and Mainland and Morrisett (2010)
- Weighted moving average, RMSE, forward difference used to show impact of optimizations
- Test system: 512MB NVIDIA GeForce 8800GT, CUDA 3.2

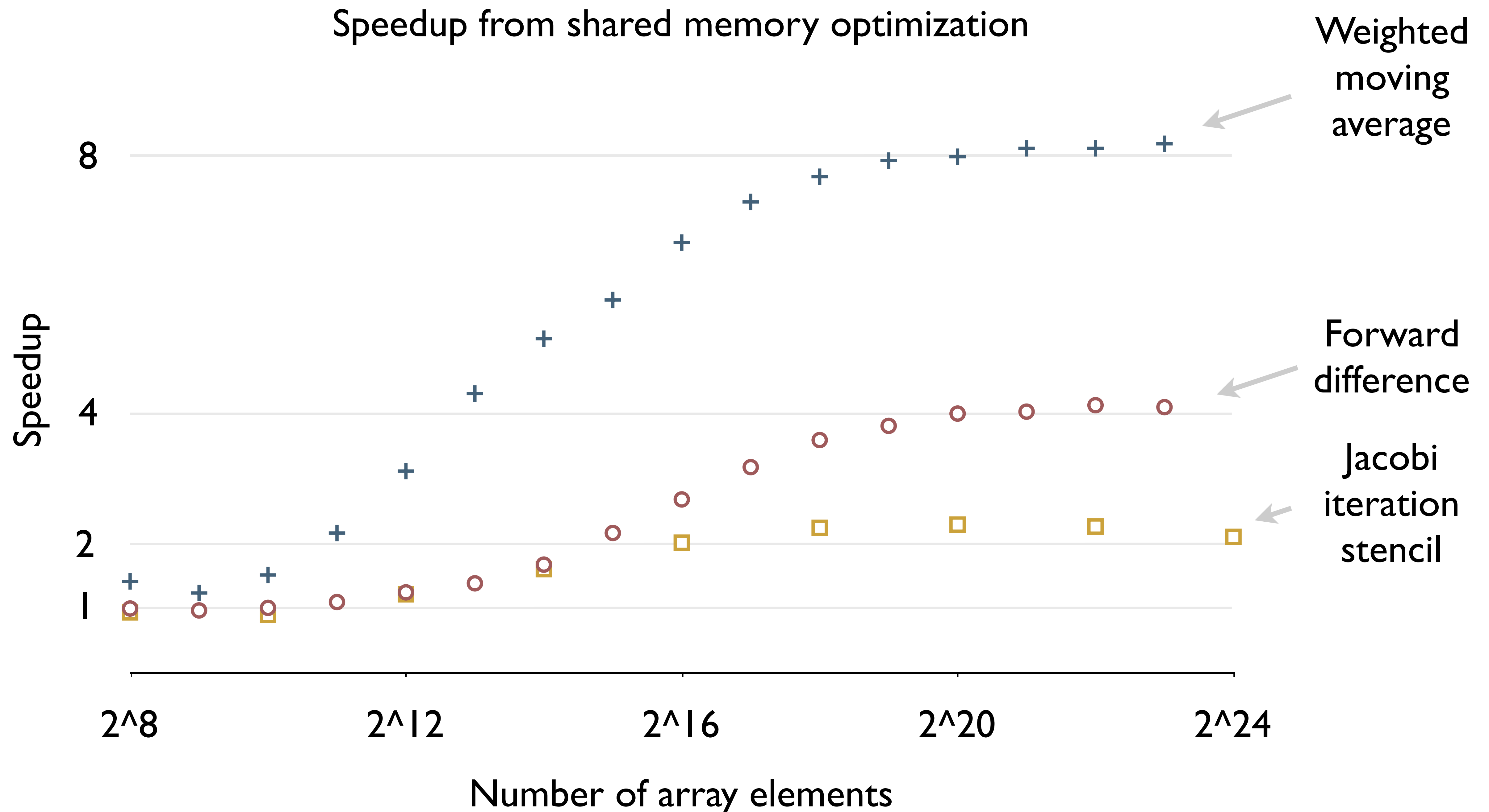
Barracuda performance is good



Array fusion is essential for good performance



Use of shared memory greatly improves performance



Speedups are enabled by careful use of declarative programming

Barracuda gets speedups through better use of GPU memory.

Computation is moved into fast memory through array fusion and shared memory optimization.

These optimizations are easy to implement because the source language is applicative and has few primitives.