Simple optimizations speed array programs on graphics processors

Bradford Larsen, Department of Computer Science, Tufts University

Graphics processors are fast, but difficult to program effectively

Modern graphics processors (GPUs) are extremely fast, computing at over 1 TFLOPS, and are flexible enough to be used for general-purpose computing.

CUDA is too low-level for easy GPU programming. For example, array summation requires ~150 lines of parallel CUDA code.

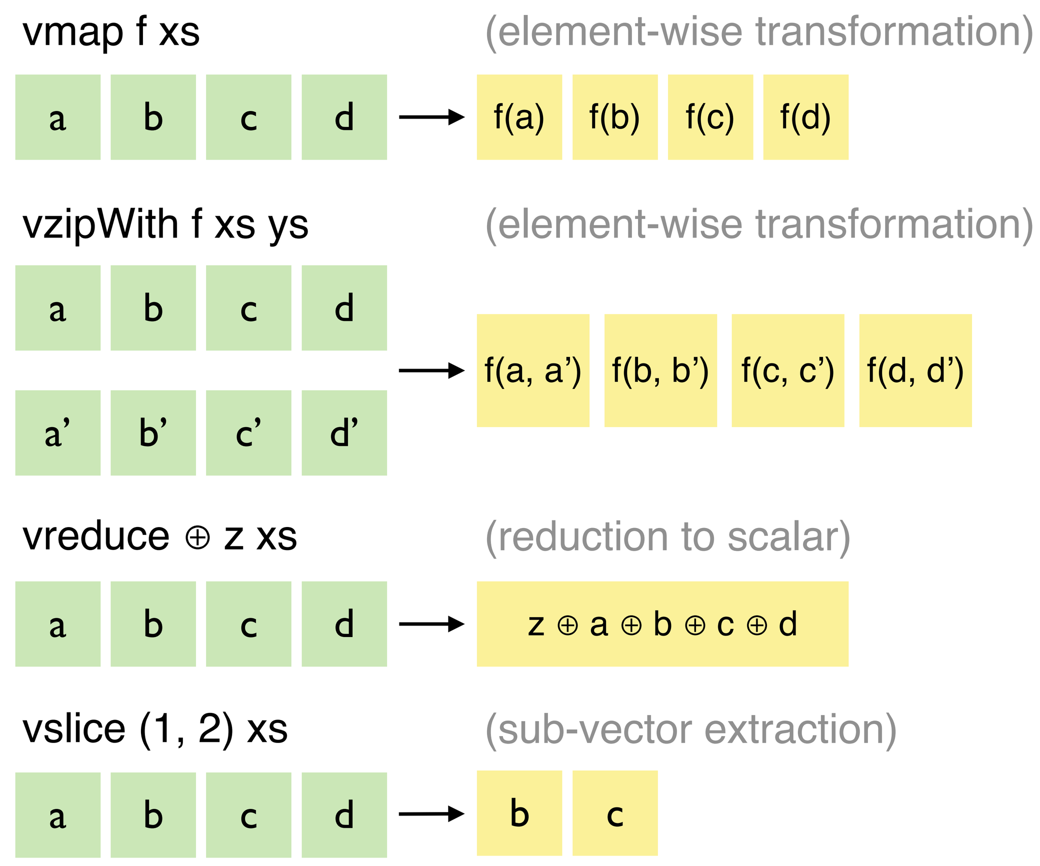
We use Barracuda, a prototype for an array-based language that is compiled into optimized CUDA code.

Barracuda emphasizes *collective array operations*, which describe how an array is transformed as a whole rather than element-by-element in a loop.

Barracuda is *applicative*, or purely functional: programs have no side effects, such as input, output, or assignment.

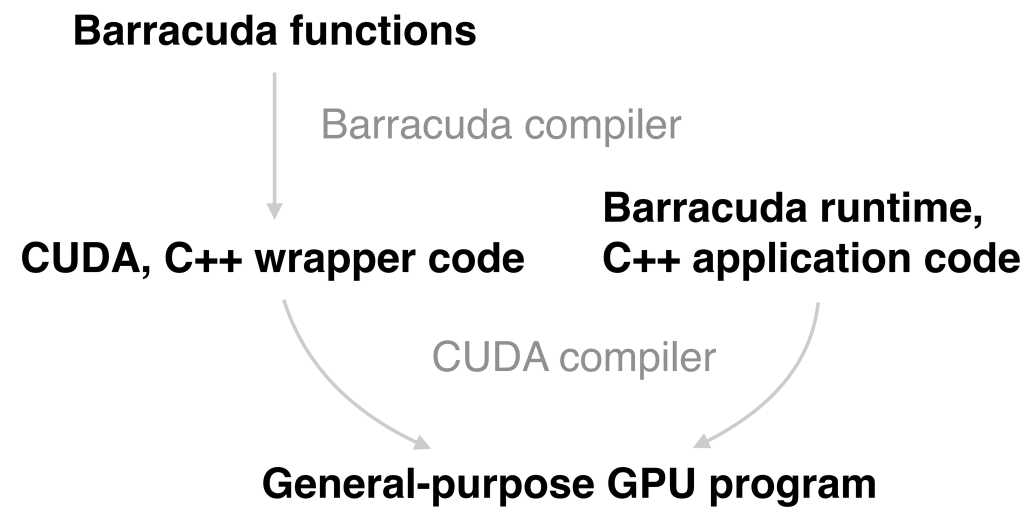
Array programs in Barracuda are concise and implicitly parallel

Barracuda provides the following array operations:



These primitives have efficient GPU implementations.

Array programs are compiled into optimized CUDA procedures



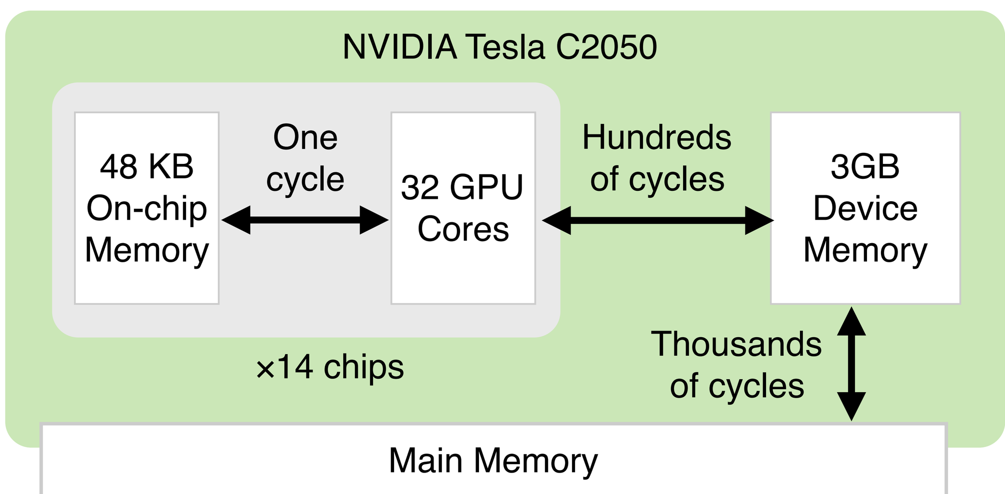
Root-mean squared error in Barracuda:

```
rmse :: VExp Float -> VExp Float -> SExp Float
rmse xs ys = sqrt (vsum diffs / len)
where diffs = vmap (^2) (vzipWith (-) xs ys)
      len = fromIntegral (vlength xs)
```

Generated CUDA procedure declaration:

```
void rmse(gpu_float_vec &xs, gpu_float_vec &ys,
          float &result);
```

Efficient GPU code exploits the memory hierarchy



The GPU interacts with a complicated memory hierarchy. On-chip memory is quite small but can be accessed as quickly as a register; the larger memories are much slower to access.

Fast GPU code will minimize overall memory traffic and will favor the faster memory; this is where much time is spent when optimizing CUDA code by hand.

Nested array expressions are common and potential trouble

Deeply nested expressions are common:

```
rmse xs ys = sqrt (vsum diffs / len)
where
  diffs = vmap (^2) (vzipWith (-) xs ys)
  len = fromIntegral (vlength xs)
```

Naive compilation would use temporary vectors and would iterate many times over the data.

Array fusion avoids trouble

Applying array indexing laws during code generation gives a simple & dependable array fusion scheme:

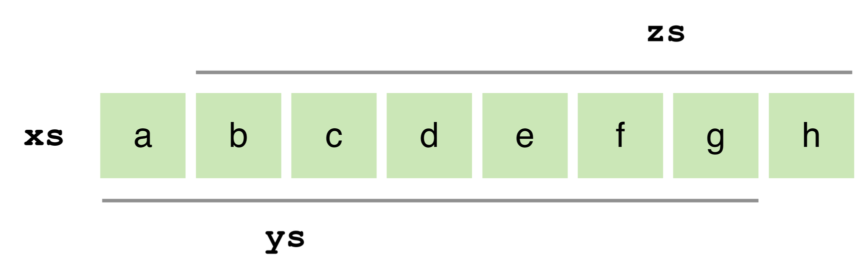
$$(\text{vmap } f\ xs)!i \quad = \quad f\ (xs!i)$$
$$(\text{vzipWith } f\ xs\ ys)!i \quad = \quad f\ (xs!i)\ (ys!i)$$
$$(\text{vslice } (b, e)\ xs)!i \quad = \quad xs!(e - b + i)$$

The `rmse` function is compiled using no temporaries and only a single pass over the data.

Barracuda's compiler uses fast on-chip GPU memory

A CUDA *kernel* specifies sequential code to run in parallel by hundreds of *threads*, each responsible for a single element of the result array. When array expressions *alias*, array elements will be read by multiple threads, e.g.:

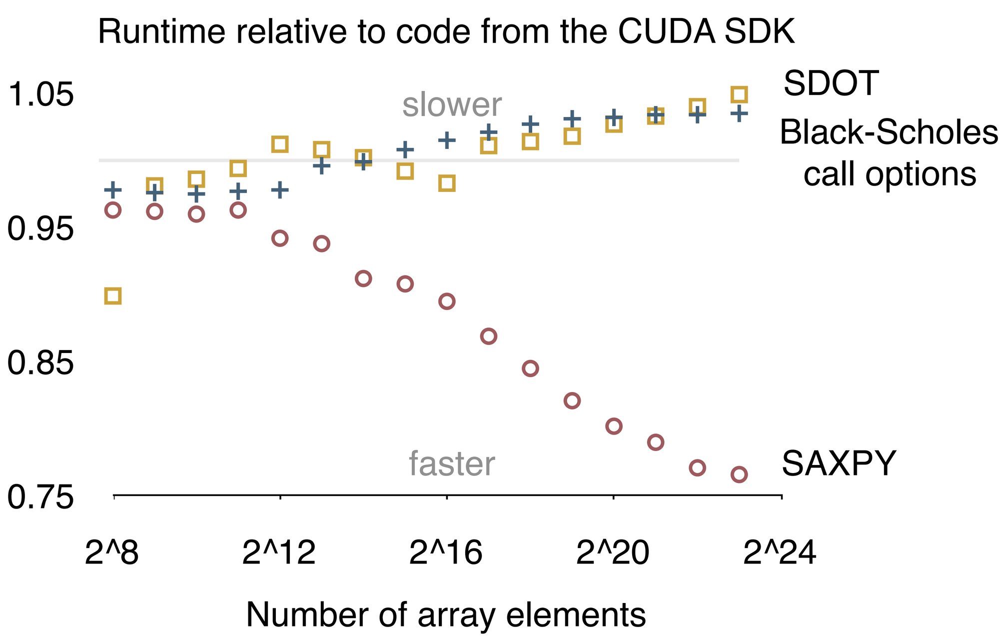
```
as = vzipWith (-) zs ys
ys = vslice (0, 6) xs
zs = vslice (1, 7) xs
```



Here elements b–g are read twice in the computation of `as`.

If read redundancy is known statically, the Barracuda compiler generates code that exploits fast on-chip memory to avoid repeatedly accessing slower memory.

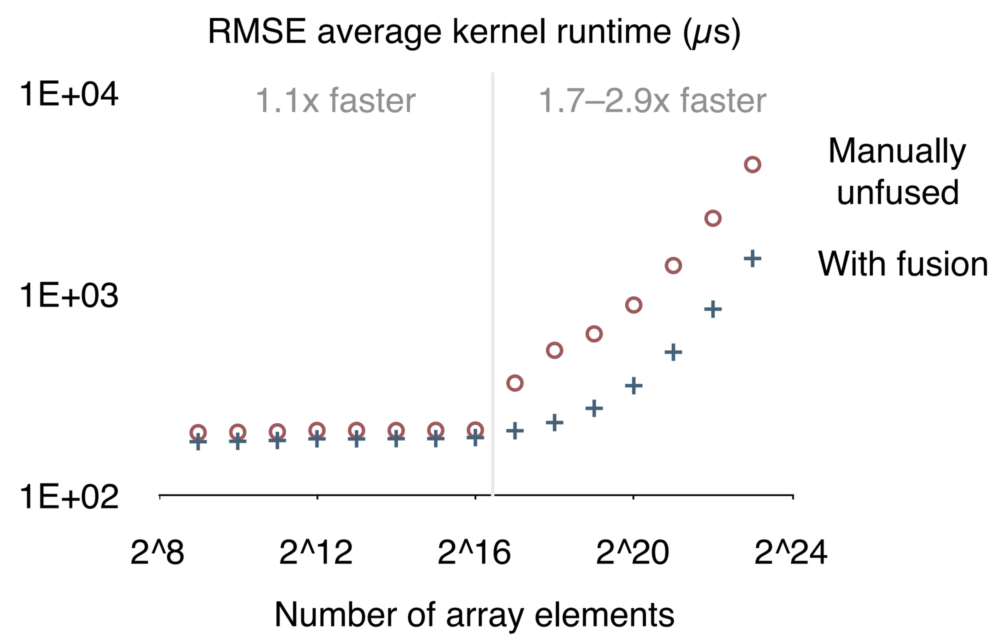
Generated code is competitive with handwritten code



New and existing benchmarks were run to evaluate the effectiveness of the optimizations. The test system had a 512 MB NVIDIA GeForce 8800 GT GPU and CUDA 3.2.

Surprisingly, Barracuda's generated SAXPY code is faster than the cuBLAS implementation.

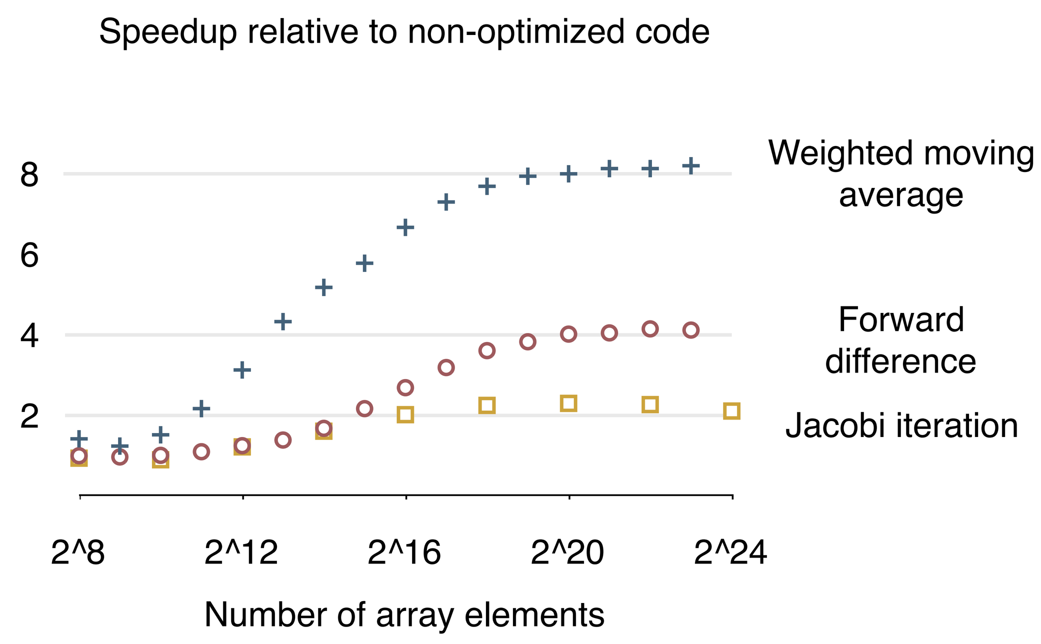
Array fusion is essential for good performance



Array fusion is always performed by the Barracuda compiler; the test kernel was manually unfused to measure the impact of the simple fusion scheme.

In the root-mean-squared error benchmark, fusion results in a 2.9 times speedup on large inputs.

Using on-chip memory greatly speeds up stencil computations



Three stencil kernels were compiled with and without on-chip memory optimization. Using on-chip memory gives dramatic speedups—8× for the weighted moving average.

Speedups are enabled by careful use of declarative programming!

This work was funded by the NASA Space Grant Graduate Fellowship and NSF grants OCI-0749125 and IIS-0082577.