Compiling NESL for GPUs

John Reppy

University of Chicago

August 2014

Credits

This work is a collaboration with

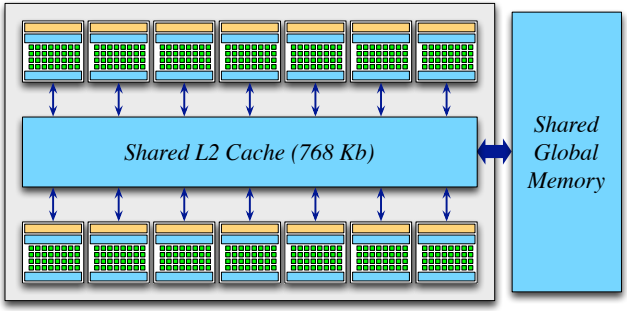
- Lars Bergstrom (Mozilla)
- Nora Sandler (U. of Chicago)

We also had support from NVIDIA.

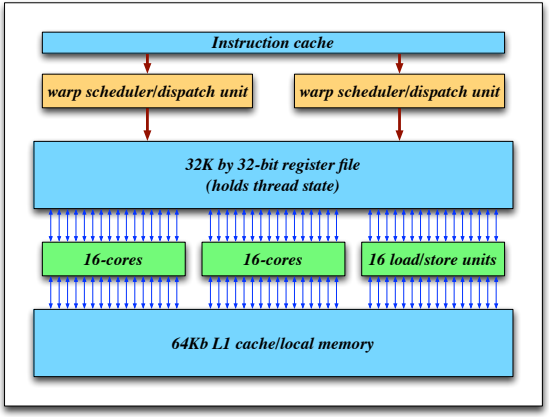
GPUs

GPUs provide super-computer levels of parallelism.

For example, NVIDIA's Fermi architecture has 14 Streaming Multiprocessors (SMs), each with 32 ALUs (1.5 TFlops peak performance).



Fermi SM



More recent designs (Kepler & Maxwell) have even more compute power.

GPU programming model

- **S**ingle-**I**nstruction, **M**ultiple-**T**hread execution model.
- Each warp (32 threads) executes the same instruction.
- SM-local barrier synchronization (fast); global atomics (slow).
- Predication used to handle divergent control flow (conditionals/loops).
- Explicit memory hierarchy:
  - per-thread memory in registers on SM
  - per-SM shared memory and cache
  - global memory (backed by shared L2 cache)
  - host memory

The “high-level” GPU programming languages (CUDA and OpenCL) expose these properties!

Programming becomes harder

C code for dot product (map-reduce):

```
float dotp (int n, float *a, float *b) {
     float sum = 0.0f;
     for (int i = 0;  i < n;  i++)
          sum += a[i] * b[i];
     return sum;
}
```

CUDA code for dot product:

```
__global__ void dotp (int n, const float *a, const float *b, float *results)
{
   __shared__ float chache[ThreadsPerBlock] ;
   float temp;
   const unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
   const unsigned int idx = threadIdx.x ;

   while (tid < n) {
        temp += a[tid] * b[tid] ;
        tid += blockDim.x * gridDim.x ;
   }
   chache[idx] = temp ;

   __synchthreads () ;

   int i = blockDim.x / 2 ;
   while (i != 0) {
        if (chacheindex < i)
            chache[chacheindex] += chache [chacheindex + i] ;
        __synchthreads () ;
        i /= 2;
   }
   if (chacheindex == 0)
        results[blockIdx.x] = chache [0];
}
```

```
// CPU side code
  cudaMalloc ((void **)&V1_D , N*sizeof(float)) ;
  cudaMalloc ((void **)&V2_D , N*sizeof(float)) ;
  cudaMalloc ((void **)&V3_D , blockPerGrid*sizeof(float)) ;

  cudaMemcpy (V1_D , V1_H , N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy (V2_D , V2_H , N*sizeof(float), cudaMemcpyHostToDevice);

  dotp <<<blockPerGrid, ThreadPerBlock>>> (N, V1_D, V2_D, V3_D);

  V3_H = new float [blockPerGrid]  ;
  cudaMemcpy (V3_H, V3_D, N*sizeof(float), cudaMemcpyDeviceToHost);

  float sum = 0 ;
  for (int i = 0 ; i<blockPerGrid ; i++)
      sum += V3_H[i] ;

  delete V3_H;
```

Better programming models for GPUs

- Domain-specific languages can be harnessed to both lift the level of programming and provide portable parallelism.
- Higher-level, but more restricted, programming models can be mapped to efficient parallel codes.

This talk is about the second approach.

NESL

- NESL is a first-order functional language for parallel programming over sequences designed by Guy Blelloch [CACM '96].
- Provides parallel for-each operation

```
{ x+y :  x in xs; y in ys }
```

- Provides other parallel operations on sequences, such as reductions, prefix-scans, and permutations.

```
function dotp (xs, ys) =
   sum ({ x*y :  x in xs; y in ys })
```

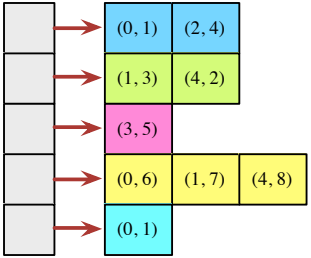
- Supports Nested Data Parallelism (NDP) — components of a parallel computation may themselves be parallel.

NDP example: sparse-matrix times vector

$$\begin{bmatrix} \mathbf{1} & 0 & \mathbf{4} & 0 & 0 \\ 0 & \mathbf{3} & 0 & 0 & \mathbf{2} \\ 0 & 0 & 0 & \mathbf{5} & 0 \\ \mathbf{6} & \mathbf{7} & 0 & 0 & \mathbf{8} \\ 0 & 0 & \mathbf{9} & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

Want to avoid computing products where matrix entries are 0.

Represent matrix as sequence of sequences of pairs



NDP example: sparse-matrix times vector

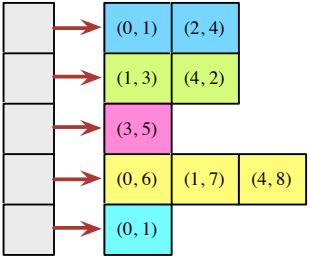
In NESL, this algorithm has a compact expression:

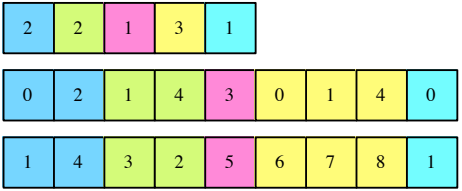
```
function svxv (sv, v) = sum ({ x * v[i] : (i, x) in sv })

function smxv (sm, v) = { svxv (sv, v) : sv in sm }
```

Notice that the `smxv` is a map of map-reduce subcomputations; *i.e.*, nested data parallelism.

NDP example: sparse-matrix times vector

Naive parallel decomposition will be unbalanced because of irregularity in sub-problem sizes.





Flattening transformation converts NDP to flat DP (including AoS to SoA)

NESL on GPUs

- NESL was designed for wide-vector machines (SIMD)
- Good fit for GPU computation
- First try [ICFP '12]:
  - Implement Blelloch's VCODE VM on GPUs using CUDA and Thrust.
  - Added map fusion.
  - Outperforms CPU on NDP benchmarks.
  - As fast as hand-written CUDA in some cases (Quickhull), but usually slower (worst case: 100 times slower on Barnes-Hut).

Areas for improvement

There are a number of areas for improvement.

- Better fusion:
  - Fuse generators, scans, and reductions with maps.
  - "Horizontal fusion," (fuse independent maps over the same index space).
- Better segment descriptor management.
- Better memory management.

It proved difficult/impossible to support these improvements.

Nessie

New NESL compiler built from scratch.

- Designed to support better fusion, *etc.*.
- Backend transforms flattened code to CUDA in several steps.
- Testbed for future optimization experiments:
  - Vectorization avoidance (works for SIMT but not SIMD) [Keller et al '12]
  - Piecewise execution [Prins '96; Madsen and Filinski '13]
  - Blocking (*i.e.*, multiple elements per CUDA thread)

Nessie compiler

- Front-end produces monomorphic, direct-style IR.
- Flattening eliminates NDP and produces Flan, which is a flat-vector language.
- Shape analysis is used to tag vectors with size information (symbolic in some cases).
- Backend transforms flattened code to CUDA in several steps.

Generating CUDA from Flan

To get from Flan to CUDA takes a number of transformation steps.

- Translate Flan to FuseAST, which makes maps, reductions, *etc.* explicit.
- Fuse map compositions (“vertical fusion”).
- Compute the PDG for the FuseAST program [Ferrante et al 1987]
- For each group of computational nodes in a control region, we compute a schedule based on data dependencies and synchronization requirements.
- Using the schedules, we translate the program into λ_{cu}, which makes the CPU/GPU distinction explicit.
- CUDA code is generated from the λ_{cu} (plus some library code).

Example

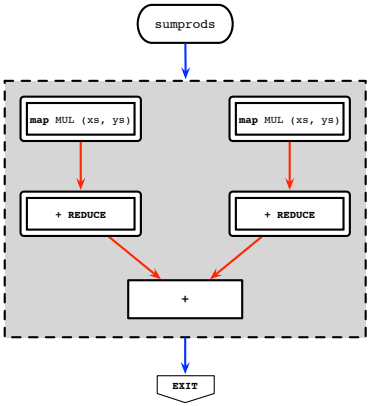
Consider the following NESL function:

```
function sumprods (xs, ys, zs) = let
  s1 = sum ({x * y : x in xs; y in ys});
  s2 = sum ({x * z : x in xs; z in zs})
in
  s1 + s2 ;
```

Example

Consider the following NESL function:

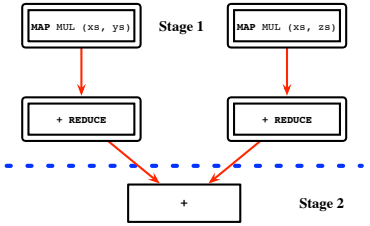
```
function sumprods (xs, ys, zs) = let
  s1 = sum ({x * y : x in xs; y in ys});
  s2 = sum ({x * z : x in xs; z in zs})
in
  s1 + s2 ;
```



Example

Consider the following NESL function:

```
function sumprods (xs, ys, zs) = let
  s1 = sum ({x * y : x in xs; y in ys});
  s2 = sum ({x * z : x in xs; z in zs})
in
  s1 + s2 ;
```



Stage 1 is translated to a CUDA kernel that produces two scalar results

Example

The λ_{cu} representation:

```
task task1 () (xs : [int], ys : [int], zs : [int]) =
  let (t1 : [int], t2 : [int]) =
        map (kernel (x, y, z) => (x*y, x*z)) (xs, ys, zs)
  let s1 = reduce t1
  let s2 = reduce t2
  in
    (s1, s2)

function sumprods (xs : [int], ys : [int], zs : [int]) =
  let (s1, s2) = run task1 () (xs, ys, zs)
  in
    s1+s2
```

Status and future work

- Generating running code for a number of the simpler examples (*e.g.*, dot product).
- Optimized reduce, scan, *etc.*, operations [NVIDIA].
- Early performance measurements are promising (1.3 speedup on dot product over VCODE version).
- Lots of work to do, particularly for segmented operations.
- Want to develop a proper calculus of heterogeneous computation (λ_{cu} is a first step).
- Lots of optimizations to explore!