Nessie: A NESL to CUDA Compiler

John Reppy

University of Chicago

October 2018

GPUs

- GPU architectures are optimized for arithmetically intensive computation.
- GPUs provide super-computer levels of parallelism at commodity prices.
- For example, the Tesla V100 provides 15.7 TFlops peak single-precision performance and 7.8 TFlops of peak double-precision performance.

NVidia GPUs have two (or three) levels of parallelism:

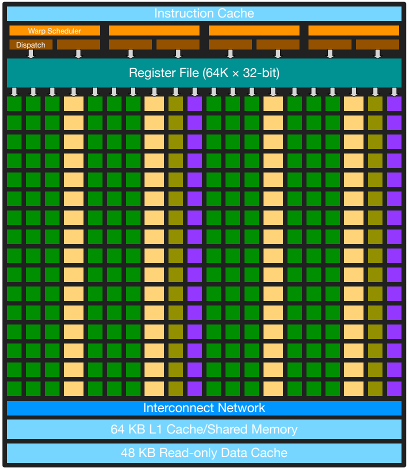
- A multicore processor that supports Single-Instruction Multiple-Thread (SIMT) parallelism.
- Multiple multicore processors on a single chip.
- Multiple GPGPU boards per system.

GPUs

For example, Nvidia's Kepler GK110 Streaming Multiprocessor (SMX).

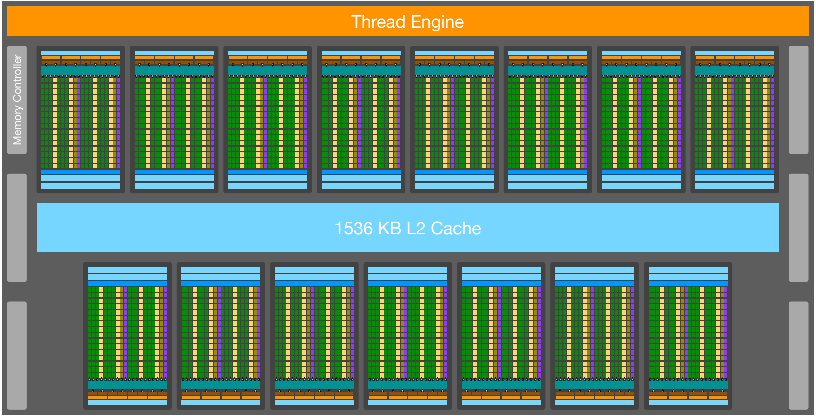
- 192 single-precision cores
- 64 double-precision cores
- 32 load/store units
- 32 special function units
- 4 $\times$ 32-lane warps in parallel

Lots of parallel compute, but not very much memory



GPUs (*continued ...*)

NVIDIA's Tesla K40 architecture has 15 GK110 SMXs (2880 Cuda cores).



Optimized for processing data in bulk!

GPU programming model

The design of GPU hardware is manifest in the widely used GPU programming languages (*e.g.*, Cuda and OpenCL).

Thread hierarchy

- Threads (grouped into warps for SIMT execution)
- Blocks (mapped to the same SMX)
- Grid (multiple blocks running the same kernel)

Synchronization

- Block-level barriers
- Atomic memory operations
- Task synchronization

Explicit memory hierarchy

- Disjoint memory spaces
- Per-thread memory maps to registers
- Per-block shared memory
- Global memory
- Host memory
- Also texture and constant memory

Programming becomes harder!

C code for dot product (map-reduce):

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

Also need 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;
```

CUDA device code for dot product:

```
__global__ void dotp (int n, const float *a, const float *b, float *results)
{
   __shared__ float cache[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 ;
   }
   cache[idx] = temp ;

   __synchthreads () ;

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

NESL

- NESL is a first-order functional language for parallel programming over sequences designed by Guy Blelloch [Blelloch ’96].
- Provides parallel for-each operation (with optional filter)

```
{ x + y : x in xs; y in ys }
{ x / y : x in xs; y in ys | (y /= 0) }
```

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

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

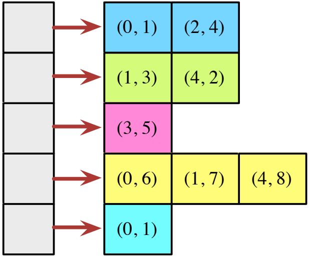
- Supports Nested Data Parallelism (NDP) — components of a parallel computation may themselves be parallel.

NDP example: sparse matrix times dense 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.

Sparse representation tracks non-zero entries using sequence of sequences of index-value pairs:



NDP example: sparse-matrix times vector

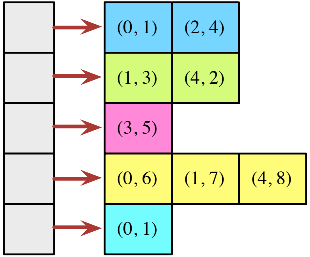
In NESL, this algorithm has a compact expression:

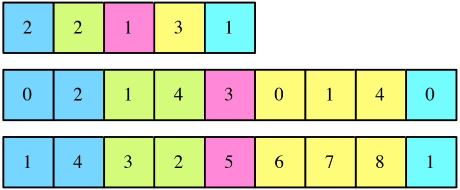
```
function svxv (sv, v) = sum ({ x * v[i] : (i, x) in sv })

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

Notice that the `smxv` function 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)

Flattening

Flattening (*a.k.a. vectorization*) is a global program transformation that converts irregular nested data parallel code into regular flat data parallel code.

- Lifts scalar operations to work on sequences of values
- Flattens nested sequences paired with segment descriptors
- Conditionals are encoded as data
- Residual program contains vector operations plus sequential control flow and recursion/iteration.

Flattening function calls

$$\{ f(e) \;\texttt{:}\; x \;\texttt{in}\; xs \}$$

$$\Longrightarrow$$

$$\begin{array}{l}
\texttt{if}\ \#xs = 0 \\
\quad \texttt{then}\ [\,] \\
\quad \texttt{else let}\ es = \{ e \;\texttt{:}\; x \;\texttt{in}\; xs \} \\
\qquad \texttt{in}\ f^{\uparrow}(es)
\end{array}$$

Lifting functions

If we have

$$\texttt{function}\, f\,(x) = e$$

then $f^{\uparrow}$ is defined to be

$$\texttt{function}\, f^{\uparrow}\,(xs) = \{\, e\ \texttt{:}\ x\ \texttt{in}\ xs\,\}$$

Flattening conditionals

$$\{\, \texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{:}\ x\ \texttt{in}\ xs \,\}$$

$$\Longrightarrow$$

$$\begin{array}{l}
\texttt{let}\ fs = \{\, b\ \texttt{:}\ x\ \texttt{in}\ xs \,\} \\
\texttt{let}\ (xs_1, xs_2) = \textbf{PARTITION}(xs, fs) \\
\texttt{let}\ vs_1 = \{\, e_1\ \texttt{:}\ x\ \texttt{in}\ xs_1 \,\} \\
\texttt{let}\ vs_2 = \{\, e_2\ \texttt{:}\ x\ \texttt{in}\ xs_2 \,\} \\
\texttt{in}\ \textbf{COMBINE}(vs_1, vs_2, fs)
\end{array}$$

Flattening example: factorial

```
function fact (n) = if (n <= 0) then 1 else n * fact (n - 1)
function fact↑ (ns) = { fact(n) : n in ns }
```

$$\Longrightarrow$$

```
function fact↑ (ns) =
  let fs = (ns <=↑ dist(0, #ns));
      (ns1, ns2) = PARTITION(ns, fs);
      vs1 = dist(1, #ns1);
      vs2 = if (#ns2 = 0)
            then []
            else let
              es = (ns2 -↑ dist(1, #ns2));
              rs = fact↑ (es);
              in (ns2 *↑ rs);
      in COMBINE(vs1, vs2, fs)
```

NESL on GPUs

- NESL was designed for bulk-data processing on wide-vector machines (SIMD)
- Potentially a good fit for GPU computation
- First try [Bergstrom & Reppy '12] demonstrated feasibility of NESL on GPUs, but was signficantly slower than hand-tuned CUDA code for some benchmarks (worst case: over 50 times slower on Barnes-Hut [Burtscher & Pingali '11]).

Areas for improvement

We identified 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 in the context of the VCODE interpreter.

Nessie

New NESL compiler built from scratch.

- Front-end produces monomorphic, direct-style IR.
- Flattening eliminates NDP and produces Flan, which is a flat-vector language with VCODE-like operators.
- Shape analysis is used to tag vectors with size information (symbolic in some cases).
- Flan is converted to λ_{cu}, which is where fusion and other optimizations occur.

λ_{cu} — An IR for GPU programs *(continued ...)*

λ_{cu} is a three-level language:

- CPU expressions – direct-style extended λ-calculus with kernel dispatch
- Kernels – sequences of second-order array combinators (SOAC)
- GPU anonymous functions – first-order functions that are the arguments to the SOACs.

λ_{cu} — An IR for GPU programs *(continued ...)*

CPU expressions

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{\mathit{kern}}\ \mathit{dcl} \\
\mathit{dcl} & ::= & \texttt{function}\ \mathrm{f}\ \texttt{(}\ \mathit{params}\ \texttt{)}\ \mathit{blk}\ \mathit{dcl} \\
 & ::= & \texttt{let}\ \mathit{params}\ \texttt{=}\ \mathit{exp}\ \mathit{dcl} \\
 & ::= & \mathit{exp} \\
\mathit{params} & ::= & \overline{x_i : \chi_i} \\
\mathit{blk} & ::= & \{\ \overline{\mathit{bind}}\ \mathit{exp}\ \} \\
\mathit{bind} & ::= & \texttt{let}\ \mathit{params}\ \texttt{=}\ \mathit{exp} \\
\mathit{exp} & ::= & \mathit{blk} \\
 & | & \texttt{run}\ \mathrm{K}\ \mathit{args} \\
 & | & \mathrm{f}\ \mathit{args} \\
 & | & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{blk}\ \texttt{else}\ \mathit{blk} \\
 & | & \mathit{exp}\ \odot\ \mathit{exp} \\
 & | & \ldots
\end{array}
$$

Kernel expressions

$$
\begin{array}{rcl}
\mathit{kern} & ::= & \texttt{kernel}\ \mathrm{K}\ \mathit{xs}\ \{\ \overline{\mathit{bind}}\ \texttt{return}\ \mathit{ys}\ \} \\
\mathit{bind} & ::= & \texttt{let}\ \mathit{xs}\ \texttt{=}\ \mathit{SOAC}\ \overline{\mathit{arg}} \\
\mathit{arg} & ::= & \Lambda \\
 & | & \mathit{rop}^{*} \\
 & | & \mathit{shape}
\end{array}
$$

GPU expressions

$$
\begin{array}{rcl}
\Lambda & ::= & \{\ \mathit{xs}\ \texttt{=>}\ \mathit{exp}\ \texttt{using}\ \mathit{ys}\ \} \\
\mathit{exp} & ::= & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
 & | & \texttt{let}\ \mathrm{x}\ \texttt{=}\ \mathit{exp}\ \texttt{in}\ \mathit{exp} \\
 & | & \mathit{exp}\ \odot\ \mathit{exp} \\
 & | & \mathrm{xs}\,\texttt{[}\,\mathrm{i}\,\texttt{]} \\
 & | & \mathrm{x} \\
 & | & \ldots
\end{array}
$$

Second-Order Array Combinators

Like Futhark [Henriksen et al. '14], Nova [Collins et al. '14], and other systems, we use Second-Order Array Combinators (SOACs) to represent the iteration structure of our operations on sequences.

$$
\begin{array}{rcl}
\texttt{ONCE} & : & (\mathbf{unit} \Rightarrow \tau) \rightarrow \tau \\
\texttt{MAP} & : & (\mathbf{int} \Rightarrow \tau)\ \mathbf{int} \rightarrow \tau^{\uparrow} \\
\texttt{PERMUTE} & : & (\mathbf{int} \Rightarrow \tau)\ (\mathbf{int} \Rightarrow \mathbf{int})\ \mathbf{int} \rightarrow \tau^{\uparrow} \\
\texttt{REDUCE} & : & (\mathbf{int} \Rightarrow \tau)\ \mathbf{rop}_{\tau}\ \mathbf{int} \rightarrow \tau \\
\texttt{SCAN} & : & (\mathbf{int} \Rightarrow \tau)\ \mathbf{rop}_{\tau}\ \mathbf{int} \rightarrow \tau^{\uparrow} \\
\texttt{FILTER} & : & (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{int} \rightarrow \tau^{\uparrow} \\
\texttt{PARTITION} & : & (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{int} \rightarrow \tau^{\uparrow} \times \tau^{\uparrow} \\
\texttt{SEG\_PERMUTE} & : & (\mathbf{int} \Rightarrow \tau)\ (\mathbf{int} \Rightarrow \mathbf{int})\ \mathbf{sd} \rightarrow \tau^{\uparrow} \\
\texttt{SEG\_REDUCE} & : & (\mathbf{int} \Rightarrow \tau)\ \mathbf{rop}_{\tau}\ \mathbf{sd} \rightarrow \tau^{\uparrow} \\
\texttt{SEG\_SCAN} & : & (\mathbf{int} \Rightarrow \tau)\ \mathbf{rop}_{\tau}\ \mathbf{sd} \rightarrow \tau^{\uparrow} \\
\texttt{SEG\_FILTER} & : & (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{sd} \rightarrow \tau^{\uparrow} \times \mathbf{sd} \\
\texttt{SEG\_PARTITION} & : & (\mathbf{int} \Rightarrow \tau)\ (\tau \Rightarrow \mathbf{bool})\ \mathbf{sd} \rightarrow \tau^{\uparrow} \times \mathbf{sd} \times \tau^{\uparrow} \times \mathbf{sd}
\end{array}
$$

Second-Order Array Combinators *(continued ...)*

There is a key difference between our combinators and previous work: combinators use a pull thunk, which is parameterized by the array indices, to get their inputs

Example: iota

```
{ i : i in [0 : n-1] }
```

$$\Longrightarrow$$

```
kernel K (n : int) -> [int] {
  let res = MAP { i => i } n
  return res
}
```

Second-Order Array Combinators *(continued ...)*

There is a key difference between our combinators and previous work: combinators use a pull thunk, which is parameterized by the array indices, to get their inputs

Example: the element-wise product of two sequences

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

$$\Longrightarrow$$

```
kernel K (xs : [float], ys : [float]) -> [float] {
  let res = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  return res
}
```

Second-Order Array Combinators *(continued ...)*

There is a key difference between our combinators and previous work: combinators use a pull thunk, which is parameterized by the array indices, to get their inputs

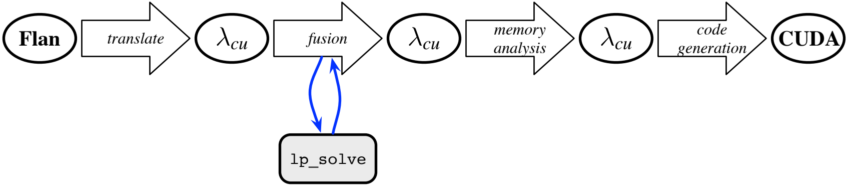
Example: summing a sequence

```
sum (xs)
```

$$\Longrightarrow$$

```
kernel K (xs : [float]) -> float {
  let res =
    REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}
```

Nessie backend



- Designed to support better fusion, *etc.*.
- Backend transforms flattened code to CUDA in several steps.
  - ILP-based fusion [Megiddo and Sarkar '99; Robinson et al '14].
  - Memory analysis based on Uniqueness types [de Vries et al '07].
  - Add explicit memory management based on analysis.

Simple map-reduce fusion

The λ_{cu} code for the `dotp` example is

```
kernel prod (xs : [float], ys : [float]) -> [float] {
  let res = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  return res
}
kernel sum (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}

function dots (xs : [float], ys : [float]) -> [float] {
  let t1 : [float] = run prod (xs, ys)
  let t2 : float = run sum (t)
  return t2
}
```

Simple map-reduce fusion

Step 1: Fuse the two kernels into a combined kernel.

```
kernel prod (xs : [float], ys : [float]) -> [float] {
  let res = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  return res
}
kernel sum (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}

function dots (xs : [float], ys : [float]) -> [float] {
  let t1 : [float] = run prod (xs, ys)
  let t2 : float = run sum (t)
  return t2
}
```

Simple map-reduce fusion

Step 1: Fuse the two kernels into a combined kernel.

```
kernel F (xs : [float], ys : [float]) -> float {
  let ts = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  let res = REDUCE { i => ts[i] using ts } (FADD) (#ts)
  return res
}

function dots (xs : [float], ys : [float]) -> [float] {
  let t2 : float = run F (xs, ys)
  return t2
}
```

Simple map-reduce fusion

Step 2: Fuse the **MAP** operation into the **REDUCE**'s pull operation

```
kernel F (xs : [float], ys : [float]) -> float {
  let ts = MAP { i => xs[i] * ys[i] using xs, ys } (#xs)
  let res = REDUCE { i => ts[i] using ts } (FADD) (#ts)
  return res
}

function dots (xs : [float], ys : [float]) -> [float] {
  let t2 : float = run F (xs, ys)
  return t2
}
```

Simple map-reduce fusion

Step 2: Fuse the **MAP** operation into the **REDUCE**'s pull operation

```
kernel F (xs : [float], ys : [float]) -> float {
  let res = REDUCE { i => xs[i] * ys[i] using xs, ys } (FADD) (#xs)
  return res
}

function dots (xs : [float], ys : [float]) -> [float] {
  let t2 : float = run F (xs, ys)
  return t2
}
```

Fancier fusion

Consider the following Nesl function (adapted from [Robinson et al ’14]):

```
function norm2 (xs) : [float] -> ([float], [float]) =
  let sum1 = sum(xs);
      gts = { x : x in xs | (x > 0) };
      sum2 = sum(gts);
  in
    ({ x / sum1 : x in xs }, { x / sum2 : x in xs })
```

Fancier fusion *(continued ...)*

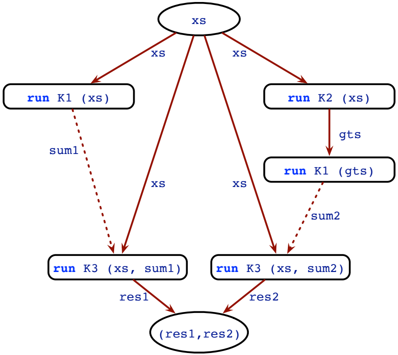
Translating to λ_{cu} produces the following code:

```
kernel K1 (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}
kernel K2 (xs : [float]) -> [float] {
  let res = FILTER { i => xs[i] using xs } { x => x > 0 } (#xs)
  return res
}
kernel K3 (xs : [float], s : float) -> [float] {
  let res = MAP { i => xs[i] / s using xs } (#xs)
  return res
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let sum1 : float = run K1 (xs)
  let its : [float] = run K2 (xs)
  let sum2 = run K1 (its)
  let res1 : [float] = run K3 (xs, sum1)
  let res2 : [float] = run K3 (xs, sum2)
  return (res1, res2)
}
```

Fancier fusion *(continued ...)*

```
kernel K1 (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}
kernel K2 (xs : [float]) -> [float] {
  let res = FILTER { i => xs[i] using xs } { x => x > 0 } (#xs)
  return res
}
kernel K3 (xs : [float], s : float) -> [float] {
  let res = MAP { i => xs[i] / s using xs } (#xs)
  return res
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let sum1 : float = run K1 (xs)
  let its : [float] = run K2 (xs)
  let sum2 = run K1 (its)
  let res1 : [float] = run K3 (xs, sum1)
  let res2 : [float] = run K3 (xs, sum2)
  return (res1, res2)
}
```

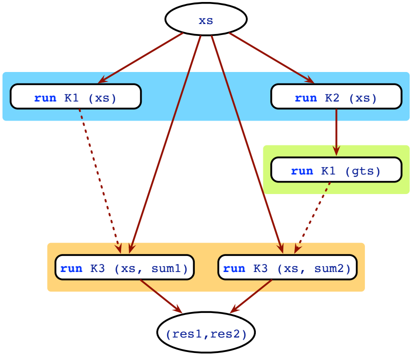
PDG control region



Fancier fusion *(continued ...)*

```
kernel K1 (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}
kernel K2 (xs : [float]) -> [float] {
  let res = FILTER { i => xs[i] using xs } { x => x > 0 } (#xs)
  return res
}
kernel K3 (xs : [float], s : float) -> [float] {
  let res = MAP { i => xs[i] / s using xs } (#xs)
  return res
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let sum1 : float = run K1 (xs)
  let its : [float] = run K2 (xs)
  let sum2 = run K1 (its)
  let res1 : [float] = run K3 (xs, sum1)
  let res2 : [float] = run K3 (xs, sum2)
  return (res1, res2)
}
```

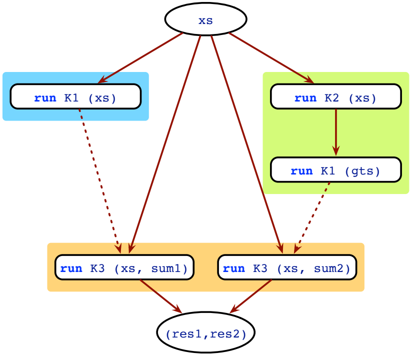
One possible schedule



Fancier fusion *(continued ...)*

```
kernel K1 (xs : [float]) -> float {
  let res = REDUCE { i => xs[i] using xs } (FADD) (#xs)
  return res
}
kernel K2 (xs : [float]) -> [float] {
  let res = FILTER { i => xs[i] using xs } { x => x > 0 } (#xs)
  return res
}
kernel K3 (xs : [float], s : float) -> [float] {
  let res = MAP { i => xs[i] / s using xs } (#xs)
  return res
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let sum1 : float = run K1 (xs)
  let its : [float] = run K2 (xs)
  let sum2 = run K1 (its)
  let res1 : [float] = run K3 (xs, sum1)
  let res2 : [float] = run K3 (xs, sum2)
  return (res1, res2)
}
```

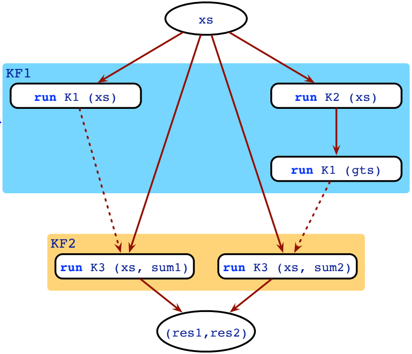
Another possible schedule



Fancier fusion *(continued ...)*

```
kernel F1 (xs : [float]) -> (float, float) {
  let (sum1, sum2) =
        REDUCE { i => let x = xs[i] in (x, if x > 0 then x else 0) using xs }
               (FADD, FADD)
               (#xs)
  return (sum1, sum2)
}
kernel F2 (xs : [float], sum1 : float, sum2 : float) -> [float] {
  let (res1, res2) =
        MAP { i => let x = xs[i] in (x / sum1, x / sum2) using xs, sum1, sum2 }
             (#xs)
  return (res1, res2)
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let (sum1 : float, sum2) = run F1 (xs)
  let (res1 : [float], res2 : [float]) = run F2 (xs, sum1, sum2)
  return (res1, res2)
}
```

Using ILP produces the following schedule:

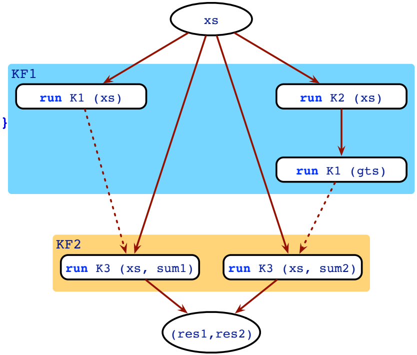


Fancier fusion *(continued ...)*

```
kernel F1 (xs : [float]) -> (float, float) {
  let (sum1, sum2) =
        REDUCE { i => let x = xs[i] in (x, if x > 0 then x else 0) using xs }
               (FADD, FADD)
               (#xs)
  return (sum1, sum2)
}
kernel F2 (xs : [float], sum1 : float, sum2 : float) -> [float] {
  let (res1, res2) =
        MAP { i => let x = xs[i] in (x / sum1, x / sum2) using xs, sum1, sum2 }
             (#xs)
  return (res1, res2)
}

function norm2 (xs : [float]) -> ([float], [float]) {
  let (sum1 : float, sum2) = run F1 (xs)
  let (res1 : [float], res2 : [float]) = run F2 (xs, sum1, sum2)
  return (res1, res2)
}
```

Notice how we fused the **FILTER** into the **REDUCE**!

Using ILP produces the following schedule:

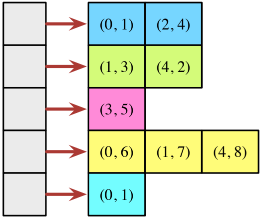


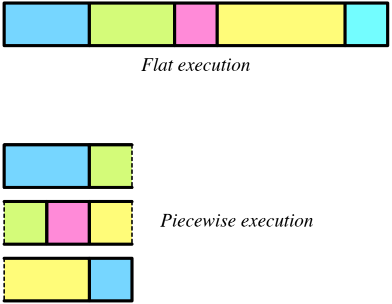
Streaming and piecewise execution

- λ_{cu} processes vectors as atomic objects, which can exceed the memory resources of a GPU.
- We could partition kernel execution into smaller pieces (either statically or dynamically) to improve scalability enable multi-GPU parallelism.
- Palmer *et al.* describe a post-flattening piecewise execution strategy and there was some follow-on work by Pfannestiel about scheduling piecewise execution for threaded execution.

Streaming and piecewise execution

- λ_{cu} processes vectors as atomic objects, which can exceed the memory resources of a GPU.
- We could partition kernel execution into smaller pieces (either statically or dynamically) to improve scalability enable multi-GPU parallelism.
- Palmer *et al.* describe a post-flattening piecewise execution strategy and there was some follow-on work by Pfannestiel about scheduling piecewise execution for threaded execution.





Streaming and piecewise execution *(continued ...)*

- Connections to Keller and Chakravarty's Distributed Types and Palmer *et al.*'s Piecewise execution of NDP programs.
- Not all operations can be executed in piecewise fashion (*e.g.*, permutations).
- The execution model for Madsen and Filinski's Streaming NESL also requires piecewise execution of kernels.