How to Compile StreamMD

Ian Buck, Mattan Erez, Francois Labonte, Ujval Kapasi

May 2002
Goals

- **Reduce compilation down to bare minimum steps**
  - No optimizations
  - Single node
  - Suggestions for multi-node

- **Assumptions**
  - Compile only to the SVM
  - SVM to HW compilation doable
  - Kernels are executed as is
    - No inter-cluster communication
    - All conditionals predicated
Simplified StreamMD Code

- Three basic kernel calls
  - simple kernels
  - self-product reduction
  - basic scalar reductions

```c
main() {
  ...
  streamSelfProduct (forcePairs, force);
  streamSelfProduct(posPairs, pos);
  MolcInteractions(pos, reduce force);  // Self-product reduction
  ...
  PosUpdate(pos, force, pos2);        // Simple kernel
  KineticEnergy (veloc, reduce &kinetic);  // Basic reductions
}
```
Outline

- Running a kernel
- Reductions of scalars
- Stream operators (self-product)
- Reductions of streams
- Multi-node issues
- Next steps
Running a Kernel

- **Load the kernel code**
  - Code is first loaded into SRF
  - SRF space is reserved for \( \mu \)code

- **All stream operations are blocking**
  - Kernels run to completion
  - No dependency analysis

- **Need to strip kernel execution**
  - Determine strip size by input and output record sizes
  - Allocate SRF for input and output streams
  - Loop until input stream is consumed
    - Load next input strip
    - Execute kernel
    - Write output stream
Running a Kernel (2)

Brook:

```c
stream Molcl  pos;
stream Force  force;
stream Molcl  pos2;

// Kernel definition
Kernel PosUpdate(Molcls pos,
                 Forces force,
                 out Molcls pos2)
{
    pos2 = f(pos, force);
}

// call kernel
PosUpdate(pos, force, pos2)
```

SVM :

```c
// Load kernel code into SRF, then micro-code
StreamMemUnitDesc[0] = {PosUpdate}
StreamSRFDesc[0] = {PosUpdate}
StreamLoad(0);
KernelCodeLoad(0, 0);

// calculate number of strips and strip size
n      = GetLength(pos);
s_size = SRF Size /
         _ (sizeof(Molcl)*2+sizeof(Force);
num_strips = n / strip_size;

// reserve SRF space
StreamSRFDesc[0] = {128 Molcl records}
StreamSRFDesc[1] = {128 Force records}
StreamSRFDesc[2] = {128 Molcl records} // out

// run the stripped kernel
for (j=0; j<num_strips; j++) {
    // Load description of streams to mem unit
    StreamMemUnitDesc[0] = {pos, j*s_size, (j+1)*s_size}
    StreamMemUnitDesc[1] = {force, j*s_size, (j+1)*s_size}
    StreamMemUnitDesc[2] = {pos2, j*s_size, (j+1)*s_size}

    // run kernel on strip
    StreamLoad(0);
    StreamLoad(1);
    KernelCall(0, 0, 1, 2);
    StreamStore(2);
}
```
Scalar Reductions

- Scalar reduction implies persistent storage
  - Insert persistent reduction variable in SVM kernel
  - Reduce across strips and pass intermediate reduction results

Brook:

```c
kernel void
KineticEnergy (veloc, reduce &kenergy) {
    kintetic += 0.5*veloc.m*veloc.vsquared;
}
```

SVM:

```c
kernel void
KineticEnergy(veloc, &kenergy, old_kenergy) {
    persistent float tmp_k = old_kenergy;
    loop_stream(veloc) {
        tmp_k += 0.5*veloc.m*veloc.vsquared;
    }
    kenergy = tmp_k
}
```

// stripped call
...
float kenergy = 0.0;
...
while strips
...
    KernelCall(“Kinetic Energy”, “veloc”,
        kenergy, kenergy);
...
Self-Product

- Actually generates unique pairs of indices
  - Insert a kernel that generates the indices
  - No optimizations of curbing $n^2$ temps by combining with the future reduction

Brook:

```
streamSelfProduct(posPairs, pos);
```

SVM:

```
kern void
posPairsSelfProduct(base_idx,
    outm ints idx[2]) {
    base = base_idx;
    for (int j=base+1; j<length; j++) {
        idx[0] << base;
        idx[1] << j;
    }
}
```

there are more load-balanced ways of generating the pairs
Stream Reductions

Brook:

MolclInteractions(molclPair p[2],
    reduce forcePair f[2])
{
    force = forceCalc(p[0], p[1]);
    f[0] += force;
    f[1] += -force;
}

SVM:

// change reduction to out and add out idx
MolclInteractions(molclPairs p[2],
    out forceIdxFPairs f[2],
    out IdxPairs idx[2])
{
    force = forceCalc(p[0], p[1]);
    f[0].force = force;  f[0].idx = idx[0];
    f[1].force = -force; f[1].idx = idx[1];
}
// flatten the idx stream since we always add
streamFlatten(f);
// sort the update forces by idx to allow reduction
SortByIdx(forceIdxFs f, out forceIdxFs f_sorted);
// reduce by summing all forces with a certain idx
ReduceByIdx(forceIdxFs f_sorted, outm f_reduced)
{
    persistent forceIdx tmp_f;
    while(tmp_f.idx == f_sorted.idx) {
        tmp_f.force += f_sorted.force;
    }
    f_reduced << tmp_f;
}

In this case we can use a ScatterAdd(f.idx, f.force) instead of the sort and reduce
Multi-Node Issues

- **Partitioning**
  - Each processor gets $N/P$ molecules (randomly)
  - Molecule data is gathered according to the pair indices
    - Not a bad solution for non-gridded if pairs are generated with load-balancing

- **Reductions**
  - Each processor gets $N/P$ elements of the reduction
  - Sort and reduce locally on each processor
  - Barrier
  - Combine results
    - Tree combine - $O(n \log(p))$ ($n$ – number of elements reduced to)
    - Each processor responsible for $n/P$ elements - $O(n)$
    - Different communication patterns for each option
  - Use a combining tree for a scalar reduction
Next Steps

- **Build a simple compiler to do the above for StreaMD**
  - define a suitable IR
  - evaluate infrastructure options (SUIF, GCC, scratch)
  - interface with the meta-compiler
  - produce SVM code (or StreamC/macrocode)

- **Analyze StreamFlow in a similar way**
  - basic steps + update the compiler

- **Optimization passes**
  - identify and rank critical optimizations
  - implement

- **Code Generation**
  - need to handle the kernels as well