Automatic Parallelisation: Code Generation for Domain Specific Languages on GPUs

Luke Cartey

Programming Tools Group, Department of Computer Science

14th December 2011

Background

Building **domain** focused languages for scientific GPU applications

- Why DSLs for GPUs?
  - No specialised expertise required.
  - Reduction in training and development time.
  - High-quality parallelization.
- Why scientific domains?
  - Large data-sets
  - Extensive search spaces
  - Repeat iterations

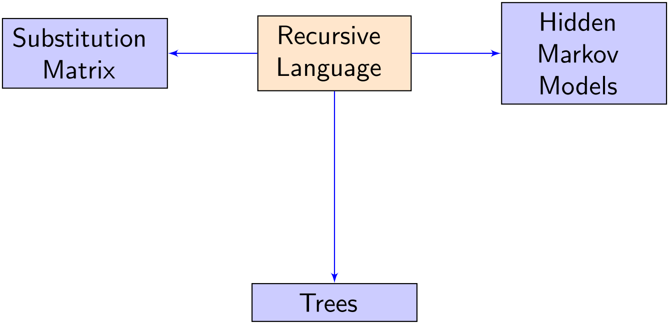
Examples in bioinformatics

Typical applications include:

- Sequence alignment
- Gene-finding
- Structure Prediction.

Using tools such as Hidden Markov Models, Stochastic Context Free Grammars, Substitution Matrices, Trees, etc.

DSL Framework



Language example - edit distance

$$d(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ d(i-1,j-1) & \text{if } s[i] = t[j] \\ min\begin{pmatrix} d(i-1,j), \\ d(i,j-1), \\ d(i-1,j-1) \end{pmatrix} + 1 & \text{otherwise} \end{cases}$$

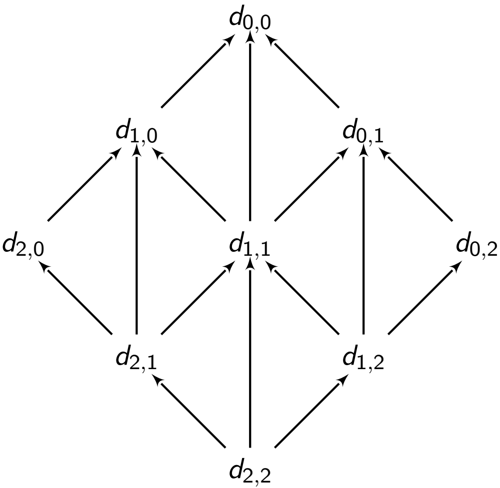
Language example - edit distance

$$d(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ d(i-1,j-1) & \text{if } s[i] = t[j] \\ min \begin{pmatrix} d(i-1,j), \\ d(i,j-1), \\ d(i-1,j-1) \end{pmatrix} + 1 & \text{otherwise} \end{cases}$$

```
int d(seq[en] s, index[s] i, seq[en] t, index[t] j) =
   if i == 0 then
      j
   else if j == 0 then
      i
   else if s[i - 1] == t[j - 1] then
      d(i - 1, j - 1)
   else
      (d(i - 1, j) min d(i, j - 1) min d(i - 1, j - 1)) + 1
```

Schedule

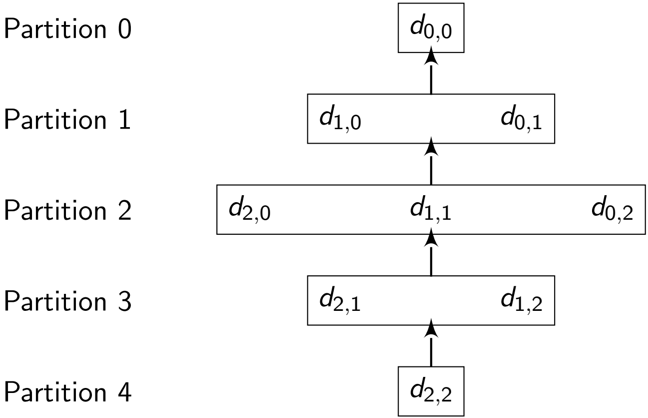
- Dependencies dictate parallelism.



Schedule

Linear function that describes partitions of **independent** values.

e.g $S_d(i, j) = i + j$



Schedule function criteria

Find $S_d(i,j) = a_1 i + a_2 j$ where $d(i,j) = \ldots d(i_r, j_r) \ldots$

The dependence condition is: $S_d(i,j) > S_d(i_r, j_r)$

Example:

$$S_d(i,j) > S_d(i-1,j)$$

$$\equiv$$

$$a_1 i + a_2 j - (a_1(i-1) + a_2 j) > 0$$

$$\equiv$$

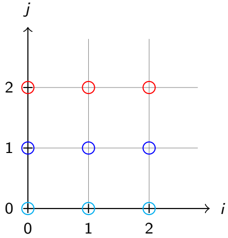
$$a_1 > 0$$

Use criteria to guide the search for a valid schedule.

Polyhedral model

Given a schedule + original function, we can generate code using the **polyhedral model**.

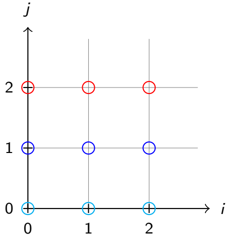
Domain of recursion: $\{0 \le i \le n, 0 \le j \le m\}$
Schedule: $S_d(i,j) = i + j$

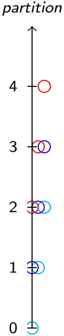


Polyhedral model

Given a schedule + original function, we can generate code using the **polyhedral model**.

Domain of recursion: $\{0 \leq i \leq n, 0 \leq j \leq m\}$
Schedule: $S_d(i, j) = i + j$

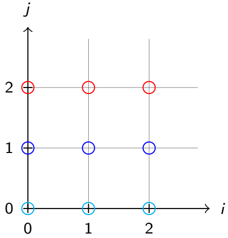


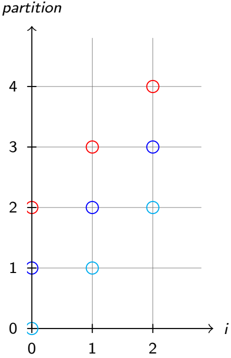


Polyhedral model

Given a schedule + original function, we can generate code using the **polyhedral model**.

Domain of recursion: $\{0 \leq i \leq n, 0 \leq j \leq m\}$
Schedule: $S_d(i, j) = i + j$





Edit Distance Program Synthesis

We use ClooG, a notable polyhedral code generator to produce a set of nested loops that iterate over the transformed domain.

```
for (p=0;p<=m+n;p++) {
  for (i=max(0,p-m);i<=min(n,p);i++) {
    S1(i,p-i);
  }
}
```

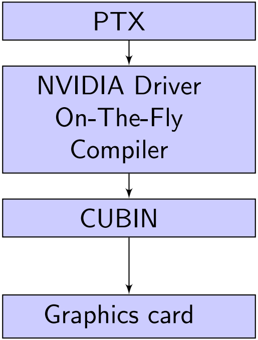
C. Bastoul: Code Generation in the Polyhedral Model is Easier Than You Think. PACT 2004

GPU Program Synthesis

Block size: *tn* threads

```
parfor threads t in 0..tn {
  for (p=0;p<=m+n;p++) {
    for (i=t+max(0,p-m);i<=min(n,p);i+=tn) {
      j = p - i;
      darr[i,j] = d(i,j);
    }
    sync;
  }
}
```

Generating PTX



Parallel **T**hread e**X**ecution
A low-level, 3-address pseudo-assembly language for CUDA cards.

Why?

- Easy for end user - only requires drivers.
- Optimising compiler - register allocation, block optimisation, etc.
- (Mostly) abstracts underlying GPU architecture.
- Cross-platform (Windows, Linux, Mac).

PTX: features and problems

- Strongly typed, verbose/explicit e.g `out[boffset] = 3`

```
ld.param.s64 %out, [%output];
mul24.lo.s32 %s1,4,%boffset;
cvt.s64.s32 %sw, %s1;
add.s64 %out,%out,%sw;
st.global.s32 [%out], 3;
```

- Efficient on code paths used by `nvcc`.
- Debugging less useful than C for Cuda.

PTX: features and problems

- Strongly typed, verbose/explicit e.g `out[boffset] = 3`

```
ld.param.s64 %out, [%output];
mul24.lo.s32 %s1,4,%boffset;
cvt.s64.s32 %sw, %s1;
add.s64 %out,%out,%sw;
st.global.s32 [%out], 3;
```

- Efficient on code paths used by `nvcc`.
- Debugging less useful than C for Cuda.

Development problems not user problems!

Intermediate Representation

Abstraction layer to aid generation of PTX:

- Describes an "abstract" many-core program.
- Support some basic constructs (loops, conditionals, etc).
- Potential for cross-platform GPU targets.

On-the-fly code generation

1. High-level code parsed, schedule analysis performed.
2. Polyhedral model used to generate IR.
3. Generate PTX from IR, and load the module.

Wrapped into a runtime framework using JCuda & JastAdd to support a wide-range of optimisations.

Optimisation examples

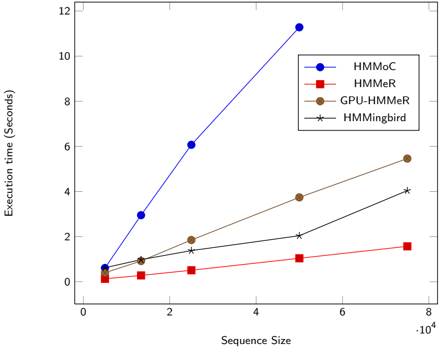
Block size

- PTX compiler returns number of registers etc.
- We can use this to suggest a block size (optimising occupancy).
- Factor in DSL model structures.

“Sliding Window”

- Parallel analysis may return “dependency” window.
- Can be used to optimise caching facilities.

Preliminary performance results



Performance for the forward algorithm on a profile HMM model of 10 positions, with varying numbers of sequences.

Conclusion

- Framework with automatic schedule identification.
- Polyhedral model ideal for implementing GPU schedules.
- PTX is a good low-level target.
- DSLs provide high-level automatic optimisation opportunities.

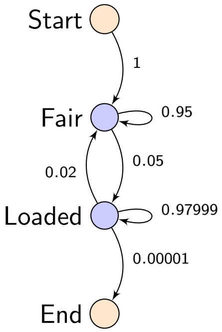
Possible extensions: Tiling large problems, complex recursions (mutual recursion).

DSL Extensions - Hidden Markov Models

```
define alphabet dice [1,2,3,4,5,6]

define hmm casino {
   alphabet dice;
   startstate start;
   state fair emits fairemission;
   state loaded emits loadedemission;
   endstate end;
   start -> fair 1;
   fair -> fair 0.94999;
   fair -> loaded 0.05;
   fair -> end 0.00001;
   loaded -> loaded 0.97999;
   loaded -> fair 0.02;
   loaded -> end 0.00001;
   emission fairemission =
      [1/6, 1/6, 1/6, 1/6, 1/6, 1/6];
   emission loadedemission =
      [0.1, 0.1, 0.1, 0.1, 0.1, 0.5];
}

prob forward(hmm h, state[h] s, seq[*] x, index[x] i) =
   if i == 0 then
      if s.isstart then 1.0 else 0.0
   else
      sum(t in s.transitionsto :
            forward(t.start, i - 1) * t.prob)
      * (if s.isend then 1.0 else s.emission{x[i-1]})
```



High-level optimisations

We can automatically:

- Optimise the model.
  - Eliminate unused elements (to/from transitions).
  - Model transformations - user accessible.
- The selection of memory location and layout.
  - Emissions - constant or texture cache
  - Transition table - texture cache
  - Sliding window size.