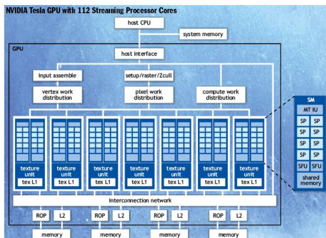
Polyhedral code generation for GPUs

Piotr Leśnicki
Albert Cohen, Cédric Bastoul
DaeGon Kim, Louis-Noël Pouchet

Goal

```
for(i=0; i<=N; i++)
  for(j=i; j<=M; j++)
       A[i][j] += B[j][i]*v[i];
```



Goal

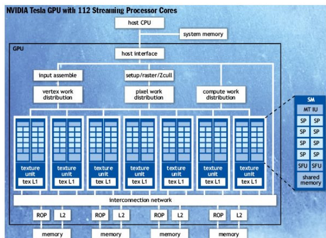
```
for(i=0; i<=N; i++)
  for(j=i; j<=M; j++)
       A[i][j] += B[j][i]*v[i];
```

programs:

- regular programs: “Static COntrol Parts”
- loop nests with affine bounds
- they have an algebraic representation

Goal

```
for(i=0; i<=N; i++)
  for(j=i; j<=M; j++)
       A[i][j] += B[j][i]*v[i];
```



processors:

- embarassingly parallel
- computation intensive, large bandwith
- easier programming model (CUDA)
- still difficult to get performance !

Issues to generate code

Legality:

- explicit parallelism
- memory copies: CPU $\leftrightarrow$ GPU

Performance:

- parallel cache (spatial locality)
- data access patterns
- parallel idioms (reductions)
- code transformations

Decomposing the problem

- program parallelization (e.g. tiling)
- loop transformations to get performance (e.g. unrolling)
- code generation to a specific AST
- memory transfers
- parallel idiom generation
- pretty printing

The polyhedral model

- well-studied representation for affine loop nests
- closed form data flow and schedule
- captures complex sequences of transformations

```
for(i= 1; i<=N, i++)
  for(j= 1; j<=N; j++)
 S:  ...= A[ i][ 2*j+1];
```

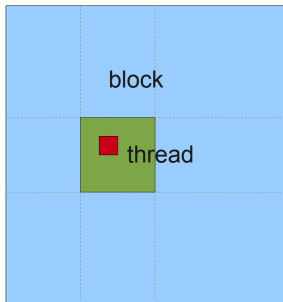
schedule of S:

$$\Theta^S.\vec{x}_S = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} . \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix}$$

access fonction of A:

$$F_{As}.\vec{x}_s = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 1 \end{pmatrix} . \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix}$$

Parallelisation



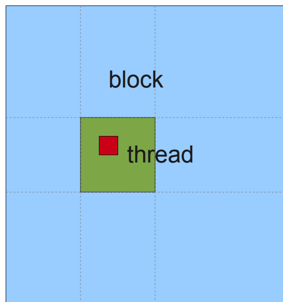

```
for ( ii=0; ii< floord(n/10); ii++)
 for( i=max( 10*ii,0);i<=min(10*ii+9, n);i++)
  S;
```

```
dimGrid =  floord(n/10);
dimBlock =  10;
 kernel<<<blockDim,threadDim>>>(...)
  ii = blockIdx;
  i = threadIdx +  10*ii;
 if(i>=0 && i<= n)
  S;
```

code legality

parallelism of gpu computations on hierarchical architecture

Parallelisation




```
for ( ii=0; ii< floord(n/10); ii++)
 for( i=max( 10*ii,0);i<=min(10*ii+9, n);i++)
  S;
```

↓

```
dimGrid = floord(n/10);
dimBlock = 10;
 kernel<<<blockDim,threadDim>>>(...)
  ii = blockIdx;
  i = threadIdx + 10*ii;
 if(i>=0 && i<= n)
  S;
```

code legality

parallelism of gpu computations on hierarchical architecture

Tiling in recent litterature:

- express tiling in the search space of affine transformation (permutable loops) :(PLDI'08)
- scalable code generation for tiling (PLDI'07)

Memory transfers

- generate data movements (DMA transfers)

code legality

gpu offshoring

- copies to parallel data cache

optimization

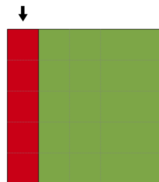
leverage temporal locality

- precise information: array access functions
- read/written data for each statement

Memory issues

issues

- global memory aligned accesses
- select data reused for caching
- beware of bank conflicts



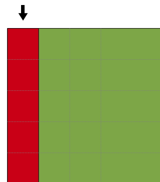
Memory issues

issues

- global memory aligned accesses
- select data reused for caching
- beware of bank conflicts

information:

- extract static access from array subscripts
- reuse: dim(data) $<$ dim(loop domain)
- access order: schedule and data layout



Memory issues

issues

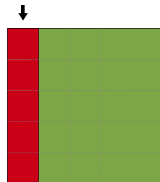
- global memory aligned accesses
- select data reused for caching
- beware of bank conflicts

information:

- extract static access from array subscripts
- reuse: dim(data) $<$ dim(loop domain)
- access order: schedule and data layout

one can think of method of increasing complexity

- can move whole array/tile (current method)
- refine precision of moved data
- change data layout during the copy for alignement / bank conflicts



Parallel idioms

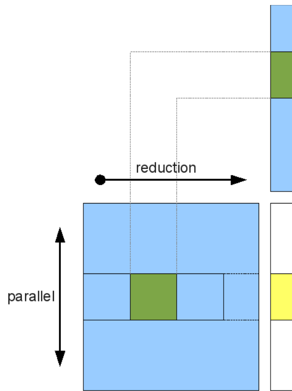
optimization

parallel algorithms: non affine case

Detect reductions

- commutative/associative operation
- accumulation
- dim(written items) $<$ dim(read items)

Simple pattern matching approach



Parallel idioms (2)

optimization

parallel algorithms: non affine case

```
for (i=0; i<=n; i++)
  v += a[i];
```

Generate "tree" pattern

- reorder operations
- parallel reduction in steps with synchronizations
- local / global

```
kernel<<<blockDim,threadDim>>>( ... ) {
 float v_shared[BLOCL_SIZE]
 i = threadIdx + blockIdx*blockDim;

  for(step=blockDim/2; step>=0; step/=2) {
   v_shared [threadIdx] += a[i + step];
    __syncthreads();
  }

 v_temp[blockIdx] = v_shared[0];
}
```

```
  kernel_on_blocks<<<...>>>()
   sum results from blocks
```

Gemver benchmark

4 statements:

1	$A = A + u1v1' + u2v2'$	2D parallel tiling [parallel]
2	$x = \beta(A'y)$	2D tiling [parallel + reduction] + fusion with (1)
3	$x = x + z$	1D tiling
4	$w = \alpha(Ax)$	2D tiling [parallel+ reduction]

Results: we're still lagging behind: for calculation alone

- on saxpy: 10% speedup
- on gemver: 8 times slowdown

Future Work

- iterative optimization (LeTSee)
- more complex data movement policies
- data layout optimizations
- staged code generation (polyhedra $\leftrightarrow$ AST)
- semantic scan (/reduction) recognition (Xavier Redon)
- parallelization by slicing approaches

Conclusion

Problem decomposion thanks to the polyhedral model

- for programs for which precise information is available
- reuse of mature tools
  - thanks to CLooG and PLuTo ! soon LeTSee
- specific parts for GPUs
  - dealing with data
  - parallel idioms