A polyhedral compilation framework for loops with dynamic data-dependent bounds

Jie Zhao and Albert Cohen

INRIA & École Normale Supérieure
45 rue d'Ulm, 75005 Paris

7th International Workshop on Polyhedral Compilation Techniques
(IMPACT 2017)
Stockholm, Sweden

January 23, 2017

Outline

1. Introduction
   - Motivation
   - Examples
2. Polyhedral compilation of dynamic counted loops
   - Schedule tree
   - Program analysis
   - Code generation
3. Experimental results
   - HOG descriptor
   - SpMV computations
   - Inspector-executor codes
4. Conclusion

Outline

1. Introduction
   - Motivation
   - Examples
2. Polyhedral compilation of dynamic counted loops
   - Schedule tree
   - Program analysis
   - Code generation
3. Experimental results
   - HOG descriptor
   - SpMV computations
   - Inspector-executor codes
4. Conclusion

Motivation

What are `dynamic counted loops`?

Motivation

What are `dynamic counted loops`?

Definition

`Counted loops` with `dynamic data-dependent bounds` are regular counted loops with numerical constant strides, whose lower and/or upper bound may not be an affine function of enclosing loop counters and loop-invariant parameters.

Motivation

What are `dynamic counted loops`?

Definition

`Counted loops` with `dynamic data-dependent bounds` are regular counted loops with numerical constant strides, whose lower and/or upper bound may not be an affine function of enclosing loop counters and loop-invariant parameters.

```
   for (i=0; i<N; i++) {
S0:  m = condition; // dynamically computed upper bound
     for (j=0; j<m; j++)
S1:    S;
   }
```

Motivation

Why are we interested in the class of loop nest kernels involving `dynamic counted loops`?

Motivation

Why are we interested in the class of loop nest kernels involving `dynamic counted loops`?

- `dynamic counted loops` are less expressive than general `while` loops.
- Less expressive/general control flow enables more aggressive optimizations.
- Building on the state of the art polyhedral optimization of `while` loops by Benabderrahmane et al. [BPCB10], but the authors' efficient code generation algorithm is not completely described.
- [BPCB10] is constrained by inductive dependences on exit conditions which limit affine transformations and parallelization.

An illustrative example

```
   for (i=0; i<N; i++) {
S0:   condition = ...;
      while (condition) {
S1:      condition = ...;
S2:      S;
      }
   }
```

A general while loop

```
   for (i=0; i<N; i++) {
S0:   m = condition;
      for (j=0; j<m; j++)
S1:      S;
   }
```

A dynamic counted loop

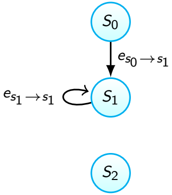
An illustrative example

```
   for (i=0; i<N; i++) {
S0:   condition = ...;
      while (condition) {
S1:      condition = ...;
S2:      S;
      }
   }
```

A general while loop

```
   for (i=0; i<N; i++) {
S0:   m = condition;
      for (j=0; j<m; j++)
S1:      S;
   }
```

A dynamic counted loop



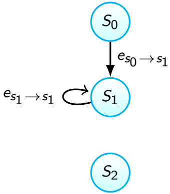
An illustrative example

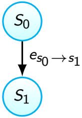
```
   for (i=0; i<N; i++) {
S0:   condition = ...;
      while (condition) {
S1:      condition = ...;
S2:      S;
      }
   }
```

A general while loop

```
   for (i=0; i<N; i++) {
S0:   m = condition;
      for (j=0; j<m; j++)
S1:      S;
   }
```

A dynamic counted loop





More examples

```
   for (i=0; i<N; i++) {
S0:  m = f(i);
S1:  n = g(i);
     for (j=0; j<m; j++)
       for (k=0; k<n; k++)
S2:      S(i, j, k);
   }
```

Histogram of Oriented Gradients (HOG)

```
   for (i=0; i<N; i++) {
S0:  m = f(i);
     for (j=0; j<m; j++)
S1:    S(i, j);
   }
```

Sparse matrix-vector CSR

```
   for (k=0; k<2*M-1; k++) {
S0:  m = f(k);
     for (i=0; i<m; i++) {
S1:    n = g(i);
       for (j=0; j<n; j++)
S2:      S(k, i, j);
     }
   }
```

Sparse matrix-vector DIA

```
   for (i=0; i<N; i++) {
S0:  m = f(i, K);
     for (jj=0; jj<m; jj++) {
S1:    n = g(i, jj, K);
       for (j=0; j<n; j++)
S2:      S(i, jj, j);
     }
   }
```

Sparse matrix-vector ELL

CSR: compressed sparse row DIA: diagonal ELL: ELLPack.

More examples

```
   for (i=0; i<N; i++) {
S0:  m = f(i);
S1:  n = g(i);
     for (j=0; j<m; j++)
       for (k=0; k<n; k++)
S2:      S(i, j, k);
   }
```

Histogram of Oriented Gradients (HOG)

```
   for (i=0; i<N; i++) {
S0:  m = f(i);
     for (j=0; j<m; j++)
S1:    S(i, j);
   }
```

Sparse matrix-vector CSR

```
   for (k=0; k<2*M-1; k++) {
S0:  m = f(k);
     for (i=0; i<m; i++) {
S1:    n = g(i);
       for (j=0; j<n; j++)
S2:      S(k, i, j);
     }
   }
```

Sparse matrix-vector DIA

```
   for (i=0; i<N; i++) {
S0:  m = f(i, K);
     for (jj=0; jj<m; jj++) {
S1:    n = g(i, jj, K);
       for (j=0; j<n; j++)
S2:      S(i, jj, j);
     }
   }
```

Sparse matrix-vector ELL

CSR: compressed sparse row DIA: diagonal ELL: ELLPack.

Outline

1. Introduction
   - Motivation
   - Examples
2. Polyhedral compilation of dynamic counted loops
   - Schedule tree
   - Program analysis
   - Code generation
3. Experimental results
   - HOG descriptor
   - SpMV computations
   - Inspector-executor codes
4. Conclusion

Schedule tree (in isl)

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \rightarrow (i); S_1(i,j) \rightarrow (i)$

sequence

$S_0(i)$

$\{S_1(i,j) : j < u\}$

$S_1(i,j) \rightarrow (j)$

Schedule tree (in isl)

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \to (i); S_1(i,j) \to (i)$

sequence

$S_0(i)$ $\quad$ $\{S_1(i,j) : j < u\}$

$S_1(i,j) \to (j)$

- Core node types
  - Band: multi-dimensional piecewise quasi-affine partial schedule
  - Filter: selects statement instances that are executed by descendants
  - Sequence/Set: children executed in given/arbitrary order

Schedule tree (in isl)

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \to (i); S_1(i,j) \to (i)$

sequence

$S_0(i)$ $\quad$ $\{S_1(i,j) : j < u\}$

$S_1(i,j) \to (j)$

- Core node types
  - Band: multi-dimensional piecewise quasi-affine partial schedule
  - Filter: selects statement instances that are executed by descendants
  - Sequence/Set: children executed in given/arbitrary order
- "External" node types
  - Domain: set of statement instances to be scheduled
  - Context: external constraints on symbolic constants

Schedule tree (in isl)

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \to (i); S_1(i,j) \to (i)$

sequence

$S_0(i)$ $\quad$ $\{S_1(i,j) : j < u\}$

$S_1(i,j) \to (j)$

- Core node types
  - Band: multi-dimensional piecewise quasi-affine partial schedule
  - Filter: selects statement instances that are executed by descendants
  - Sequence/Set: children executed in given/arbitrary order
- "External" node types
  - Domain: set of statement instances to be scheduled
  - Context: external constraints on symbolic constants
- Convenience node types
  - Mark: attach additional information to subtrees

Program analysis

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:      S(i, j);
   }
```

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:     S(i, j);
   }
```

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:     S(i, j+idx[i]);
   }
```

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:    S(i, j);
   }
```

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    S(i, j+idx[i]);
   }
```

- Modeling control dependences
  - Insert an exit predicate definition and check at the beginning of each iteration of a dynamically counted loop.
  - Delay the introduction of break instructions until code generation to keep the control flow in a manageable form for a polyhedral compiler.

Program analysis

- Preprocessing
  - Subtract (dynamic) lower bounds.
  - Synthesize static upper bounds (static analysis or dynamic inspector).

```
   for (i=0; i<N; i++) {
     for (j=idx[i]; j<idx[i+1]; j++)
S1:    S(i, j);
   }
```

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    S(i, j+idx[i]);
   }
```

- Modeling control dependences
  - Insert an exit predicate definition and check at the beginning of each iteration of a dynamically counted loop.
  - Delay the introduction of break instructions until code generation to keep the control flow in a manageable form for a polyhedral compiler.

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    if (j<m)
         S(i, j+idx[i]);
   }
```

Program analysis

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counter loop.

Program analysis

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counter loop.

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    if (j<m)
         S(i, j+idx[i]);
   }
```

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
     }
```

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counter loop.

```
   for (i=0; i<N; i++) {
S0:   m = idx[i+1] - idx[i];
      for (j=0; j<m; j++)
S1:      if (j<m)
            S(i, j+idx[i]);
   }
```

```
   for (i=0; i<N; i++)
      for (j=0; j<u; j++) {
S0:      m = idx[i+1] - idx[i];
S1:      if (j<m)
            S(i, j+idx[i]);
      }
```

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \to (i); S_1(i,j) \to (i)$

sequence

$S_0(i)$

$\{S_1(i,j) : j < u\}$

$S_1(i,j) \to (j)$

Program analysis

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counter loop.

```
   for (i=0; i<N; i++) {
S0:   m = idx[i+1] - idx[i];
      for (j=0; j<m; j++)
S1:      if (j<m)
            S(i, j+idx[i]);
   }
```

```
   for (i=0; i<N; i++)
      for (j=0; j<u; j++) {
S0:      m = idx[i+1] - idx[i];
S1:      if (j<m)
            S(i, j+idx[i]);
      }
```

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$\boxed{S_0(i) \to (i); S_1(i,j) \to (i)}$

sequence

$S_0(i)$ $\quad$ $\{S_1(i,j) : j < u\}$

$\boxed{S_1(i,j) \to (j)}$

Program analysis

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counter loop.

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    if (j<m)
          S(i, j+idx[i]);
   }
```

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
          S(i, j+idx[i]);
     }
```

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \to (i); S_1(i,j) \to (i)$

sequence

$S_0(i)$ $\quad$ $\{S_1(i,j) : j < u\}$

$S_1(i,j) \to (j)$

domain

context: $[u]$

$\{S_0(i,j); S_1(i,j)\}$

$S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)$

mark: "*dynamic_counted_loop*$([j], j < u)$"

sequence

$S_0(i,j)$ $\quad$ $S_1(i,j)$

Program analysis

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counter loop.

```
   for (i=0; i<N; i++) {
S0:   m = idx[i+1] - idx[i];
      for (j=0; j<m; j++)
S1:      if (j<m)
            S(i, j+idx[i]);
   }
```

```
   for (i=0; i<N; i++)
      for (j=0; j<u; j++) {
S0:      m = idx[i+1] - idx[i];
S1:      if (j<m)
            S(i, j+idx[i]);
      }
```

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \to (i); S_1(i,j) \to (i)$

sequence

$S_0(i)$ $\quad$ $\{S_1(i,j) : j < u\}$

$S_1(i,j) \to (j)$

domain

context: $[u]$

$\{S_0(i,j); S_1(i,j)\}$

$\boxed{S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)}$

mark: "*dynamic_counted_loop*$([j], j < u)$"

sequence

$S_0(i,j)$ $\quad$ $S_1(i,j)$

- Schedule generation
  - Apply any affine transformation, e.g., a variant of the Pluto algorithm.
  - Insert a mark node below each band node associated with a dynamically counter loop.

```
   for (i=0; i<N; i++) {
S0:  m = idx[i+1] - idx[i];
     for (j=0; j<m; j++)
S1:    if (j<m)
         S(i, j+idx[i]);
   }
```

```
   for (i=0; i<N; i++)
     for (j=0; j<u; j++) {
S0:    m = idx[i+1] - idx[i];
S1:    if (j<m)
         S(i, j+idx[i]);
     }
```

domain

context: $[u]$

$\{S_0(i); S_1(i,j)\}$

$S_0(i) \to (i); S_1(i,j) \to (i)$

sequence

$S_0(i)$ $\{S_1(i,j) : j < u\}$

$S_1(i,j) \to (j)$

domain

context: $[u]$

$\{S_0(i,j); S_1(i,j)\}$

$\boxed{S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)}$

mark: "*dynamic_counted_loop*$([j], j < u)$"

sequence

$S_0(i,j)$ $S_1(i,j)$

Code generation

Code generation

- About the general applicability of affine transformations
  - by default, resort to unoptimized exit-predicated execution with static upper bounds
  - simple yet optimized code generation template for tiling, strip mining, skewing, interchange.
  - the template does not apply to fusion and reversal.

Code generation

- About the general applicability of affine transformations
  - by default, resort to unoptimized exit-predicated execution with static upper bounds
  - simple yet optimized code generation template for tiling, strip mining, skewing, interchange.
  - the template does not apply to fusion and reversal.
- Code generation

Code generation

- About the general applicability of affine transformations
  - by default, resort to unoptimized exit-predicated execution with static upper bounds
  - simple yet optimized code generation template for tiling, strip mining, skewing, interchange.
  - the template does not apply to fusion and reversal.
- Code generation

domain

context: $[u]$

$\{S_0(i,j); S_1(i,j)\}$

$\boxed{S_0(i,j) \to (i); S_1(i,j) \to (i); S_0(i,j) \to (j); S_1(i,j) \to (j)}$

mark: "$dynamic\_counted\_loop([j], j < u)$"

sequence

$S_0(i,j)$ $S_1(i,j)$

Code generation

The code generation template

```
   for (i=0; i<N; i++)
     for (j=0; j<u1; j++) {
       for (k=0; k<u2; k++) {
         for (...) {
S0:        m = f(i);
S1:        n = g(i);
            ...
Sn:        if (j<m&&k<n&&...)
             S(i, j, k, ...);
           ...
         }
         if (k>=n)
           break;
       }
       if (j>=m)
         break;
     }
```

Code generation

The code generation template

```
   for (i=0; i<N; i++)
     for (j=0; j<u1; j++) {
       for (k=0; k<u2; k++) {
         for (...) {
S0:        m = f(i);
S1:        n = g(i);
            ...
Sn:        if (j<m&&k<n&&...)
             S(i, j, k, ...);
           ...
         }
         if (k>=n)
           break;
       }
       if (j>=m)
         break;
     }
```

Code generation

SpMV CSR code

```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:   m = idx[i+1] - idx[i];
S1:   if (j<m)
        y[i] += A[j]*x[col[j]];
   }
```

Code generation

SpMV CSR code

```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:   m = idx[i+1] - idx[i];
S1:   if (j<m)
        y[i] += A[j]*x[col[j]];
   }
```

```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:   m = idx[i+1] - idx[i];
S1:   if (j<m)
        y[i] += A[j]*x[col[j]];
      if (j>=m)
        break;
   }
```

Code generation

SpMV CSR code

```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:   m = idx[i+1] - idx[i];
S1:   if (j<m)
         y[i] += A[j]*x[col[j]];
   }
```

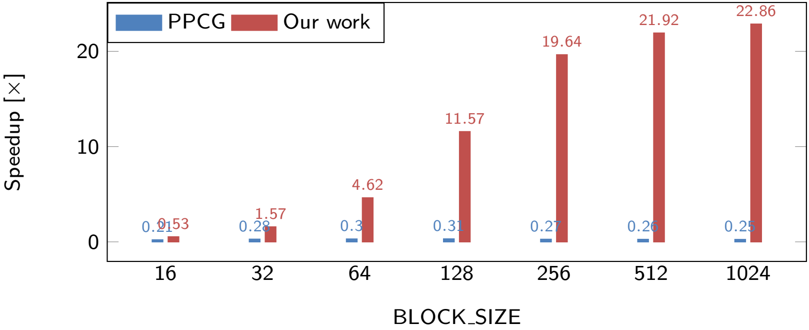
```
 for (i=0; i<N; i++)
   for (j=0; j<u; j++) {
S0:   m = idx[i+1] - idx[i];
S1:   if (j<m)
         y[i] += A[j]*x[col[j]];
      if (j>=m)
         break;
   }
```

```
   for (ii=32*b0; ii<N; ii+=8192)
     for (jj=32*b1; jj<u; jj+=8192) {
       for (i=t0; i<=min(31,N-ii); i+=32)
         for(j=t1; i<=min(31,u-jj); i+=32) {
S0:        m = idx[ii+i+1] - idx[ii+i];
S1:        if (jj+j<m)
             y[ii+i] += A[jj+j]*x[col[jj+j]];
           if (jj+j>=m)
             break;
         }
       if (jj>=m)
         break;
     }
```

Outline

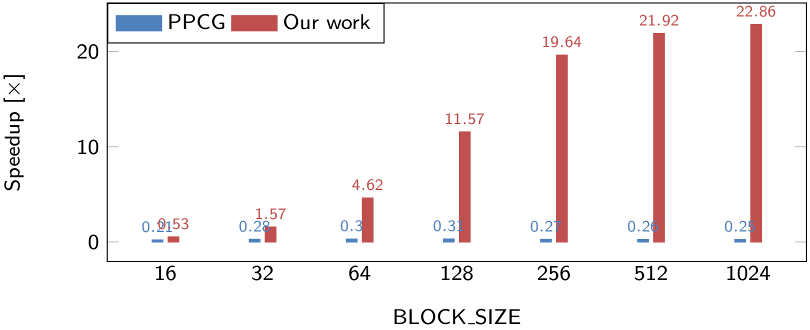
HOG descriptor

Performance of the HOG descriptor including data transfer time[1]



[1]Running on NVIDIA Quadro K4000 GPU with Intel Xeon E5-2630 CPU

Performance of the HOG descriptor including data transfer time[1]

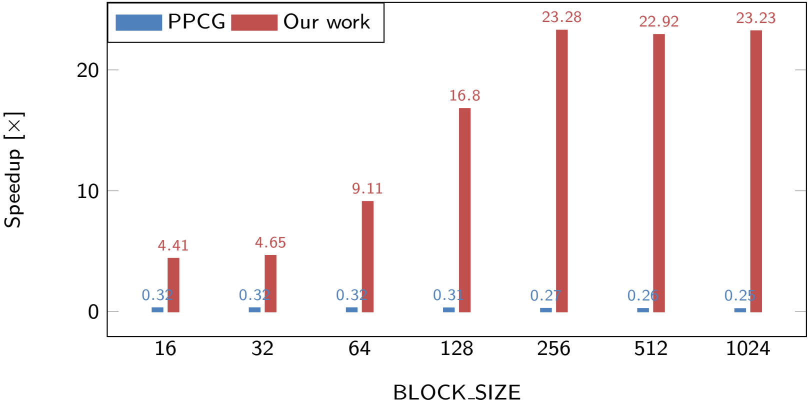


Our work enables the effective parallelization and optimization of HOG on the target platform, outperforming sequential execution and PPCG's inner parallelism version (total execution time including data transfers)

[1] Running on NVIDIA Quadro K4000 GPU with Intel Xeon E5-2630 CPU

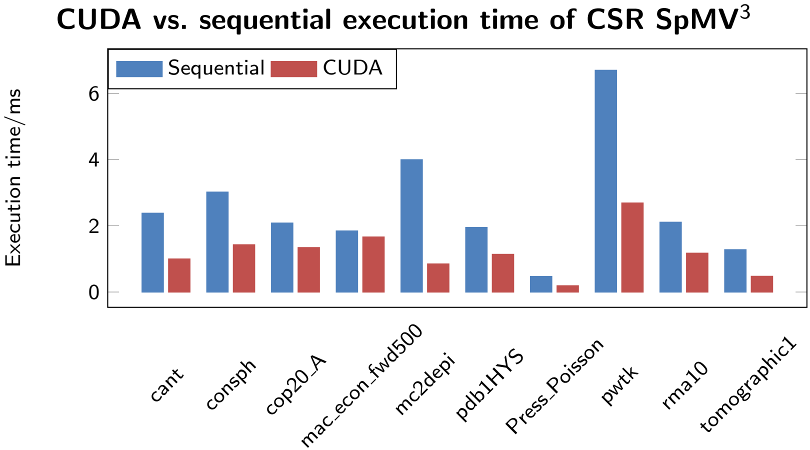
HOG descriptor

Performance of the HOG descriptor without data transfer time[2]



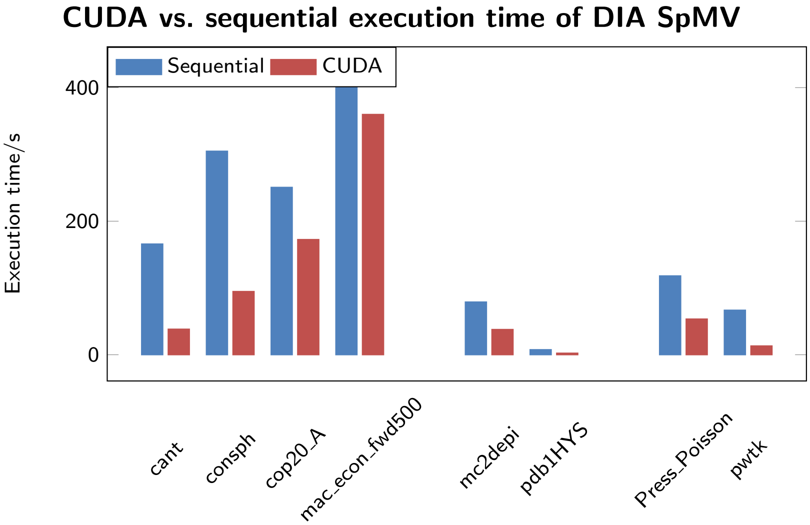
[2] The performance is the speedup w.r.t. the sequential execution time

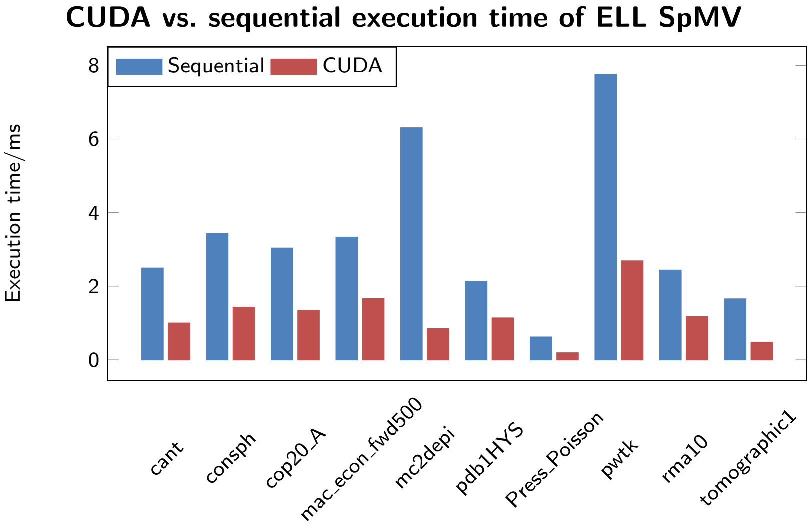
SpMV computations



[3] Based on the University of Florida sparse matrix collection [DH11]

SpMV computations




CUDA vs. sequential execution time of ELL SpMV
Sequential
CUDA
Execution time/ms
0
2
4
6
8
cant
consph
cop20_A
mac_econ_fwd500
mc2depi
pdb1HYS
Press_Poisson
pwtk
rma10
tomographic1

An inspector-executor implementation of CSR SpMV

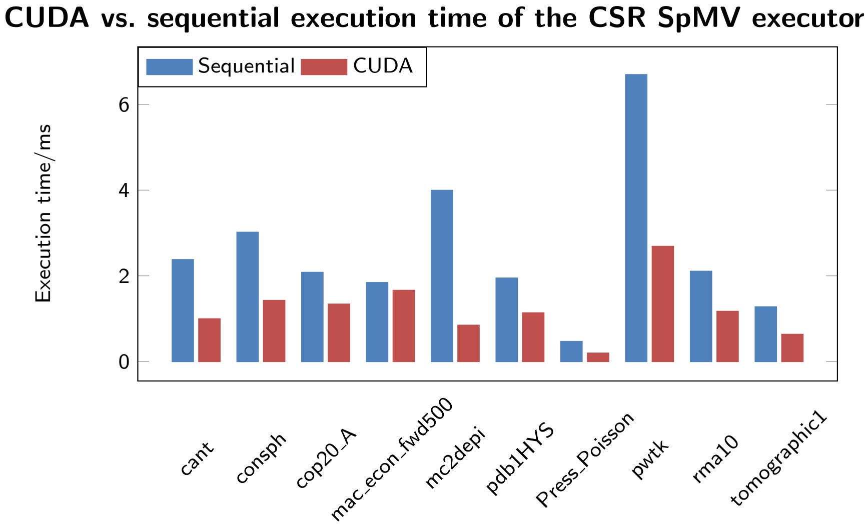
```
for (i=0; i<M; i++) {
  for (k=0; k<N; k++) {
    marked = false;
    for (j=idx[i]; j<idx[i+1]; j++)
      if (k == col[j])
        if (!marked) {
          marked = true;
          exp_idx[count] = k;
          count++;
        }
  }
  f_idx[i+1] = count;
}
```

inspector

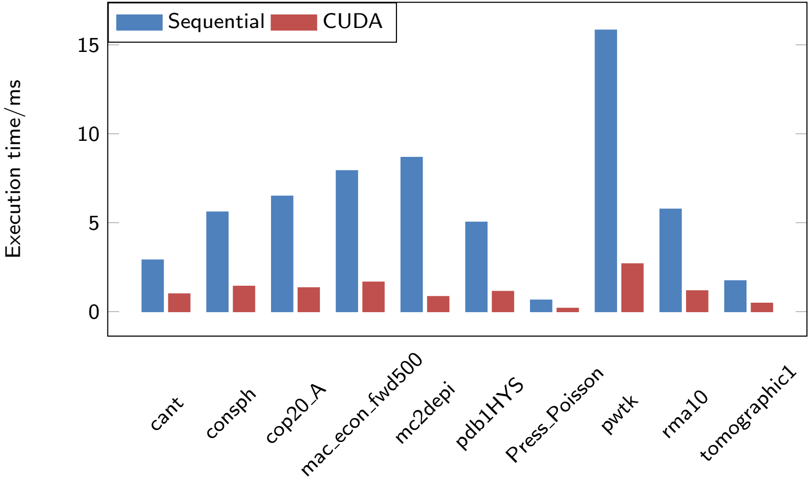
```
for (i=0; i<N; i++) {
  m = f_idx[i+1] - f_idx[i];
  for (j=0; j<m; j++)
    if (j<m)
      y[i] += val[j+f_idx[i]] * x[exp_idx[j+f_idx[i]]];
}
```

executor

Inspector-executor codes



CUDA vs. sequential execution time of the ELL SpMV executor



Outline

Conclusion

- Our work

- Future work

Conclusion

- Our work
  - models control dependences on data-dependent predicates by revisiting the work of Benabderrahmane et al. [BPCB10].
  - does not resort to more expressive first-order logic with non-interpreted functions/predicates, like [SCF03, SLC+16].
  - provides code generation templates for multiple scenarios, including the inspector-executor scheme [VHS15].
- Future work

Conclusion

- Our work
  - models control dependences on data-dependent predicates by revisiting the work of Benabderrahmane et al. [BPCB10].
  - does not resort to more expressive first-order logic with non-interpreted functions/predicates, like [SCF03, SLC$^+$16].
  - provides code generation templates for multiple scenarios, including the inspector-executor scheme [VHS15].
- Future work
  - fully automate and implement the framework in PPCG [VCJC$^+$13].
  - conduct further experiments on CPU and GPU platforms, comparing the performance with the CUSP library.

References

- Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul.
  The polyhedral model is more widely applicable than you think.
  In *Proceedings of 19th International Conference on Compiler Construction (CC)*, pages 283–303. Springer, 2010.
- Timothy A Davis and Yifan Hu.
  The university of florida sparse matrix collection.
  ACM Transactions on Mathematical Software (TOMS), 38(1):1:1–1:25, 2011.
- Michelle Mills Strout, Larry Carter, and Jeanne Ferrante.
  Compile-time composition of run-time data and iteration reorderings.
  ACM SIGPLAN Notices, 38(5):91–102, 2003.
- Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky.
  An approach for code generation in the sparse polyhedral framework.
  Parallel Computing, 53:32–57, 2016.
- Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor.
  Polyhedral parallel code generation for cuda.
  ACM Transactions on Architecture and Code Optimization (TACO), 9(4):54, 2013.
- Anand Venkat, Mary Hall, and Michelle Strout.
  Loop and data transformations for sparse matrix code.
  In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 521–532, 2015.