Adaptive Mapping and Parameter Selection Scheme to Improve Automatic Code Generation for GPUs

Carlos Juega [1] José Ignacio Gómez [1] Christian Tenllado [1] Francky Catthoor [2]

[1]ArTeCS, Universidad Complutense de Madrid

[2]IMEC

February 19, 2014

1. Motivation
2. PPCG
3. Adaptive Mapping
4. Results
5. Conclusions

Motivation

Some facts...

- GPGPU programming is still a hard task.
  - Parallelism vs locality trade-off rules the mapping.
- *Optimal* implementation usually changes across different GPU generations.

Motivation

Some facts...

- GPGPU programming is still a hard task.
  - Parallelism vs locality trade-off rules the mapping.
- *Optimal* implementation usually changes across different GPU generations.

gemm	
naive	40 ms
⇓ parallelism, shared memory	12.15 ms
⇓⇓ parallelism, shared memory + registers	9.36 ms

Motivation

Some facts...

- GPGPU programming is still a hard task.
  - Parallelism vs locality trade-off rules the mapping.
- *Optimal* implementation usually changes across different GPU generations.

There is always hope!

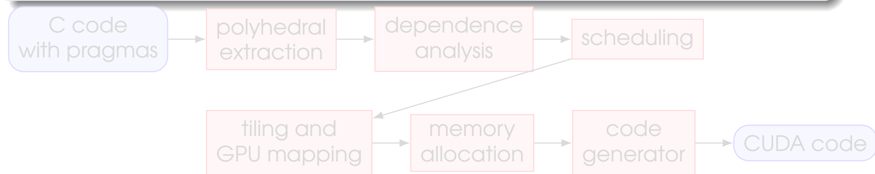
- Many source-to-source compilers for GPUs.
- Other approaches (OpenACC, CUDA-CHiLL, ...).

Polyhedral Parallel Code Generator

Source-to-source compiler based on Polyhedral Model.

Contributions

1. Exposes parallelism.
2. Maximizes temporal locality.
3. Exploits shared memory and registers.

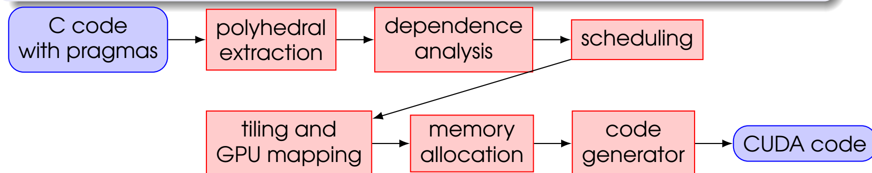


Polyhedral Parallel Code Generator

Source-to-source compiler based on Polyhedral Model.

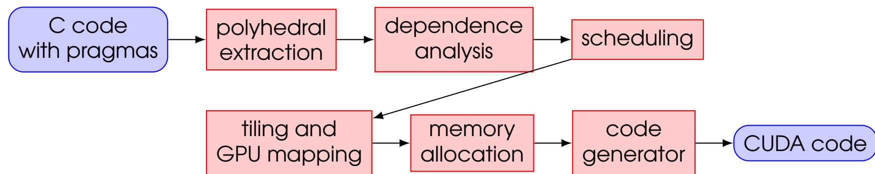
Contributions

1. Exposes parallelism.
2. Maximizes temporal locality.
3. Exploits shared memory and registers.



PPCG limitations

1. Static mapping process (it's always the same no matter the input).
2. It still needs user assistance (parametrization).
3. It's not aware of platform details.



Proposal

Basics

- Explore minor changes in PPCG's schedule.
  - Loop interchange.
  - Serialize parallel loops.
- Use parametric tiling or ranged values (just for pruning).
- Takes platform specs as problem variables.

Heuristics

- Memory footprint.
- Accesses cost.

Contributions

- Adapts PPCG schedules to the input's data reuse requirements.
- Computes common GPU parameters based on data reuse.

Proposal

Basics

- Explore minor changes in PPCG's schedule.
  - Loop interchange.
  - Serialize parallel loops.
- Use parametric tiling or ranged values (just for pruning).
- Takes platform specs as problem variables.

Heuristics

- Memory footprint.
- Accesses cost.

Contributions

- Adapts PPCG schedules to the input's data reuse requirements.
- Computes common GPU parameters based on data reuse.

Proposal

Basics

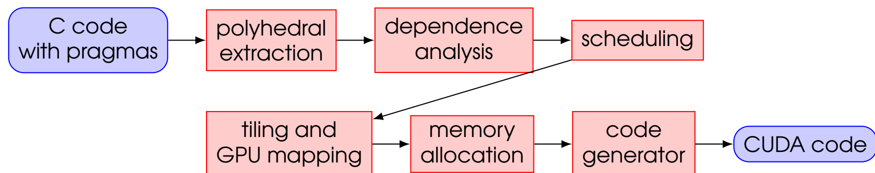
- Explore minor changes in PPCG's schedule.
  - Loop interchange.
  - Serialize parallel loops.
- Use parametric tiling or ranged values (just for pruning).
- Takes platform specs as problem variables.

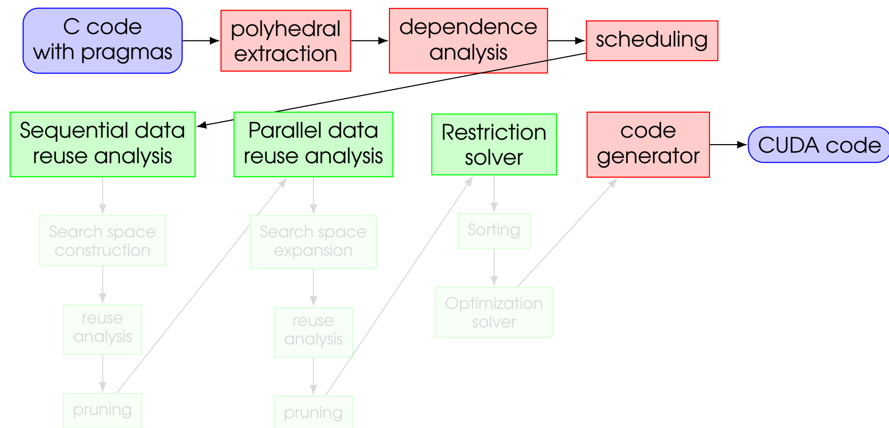
Heuristics

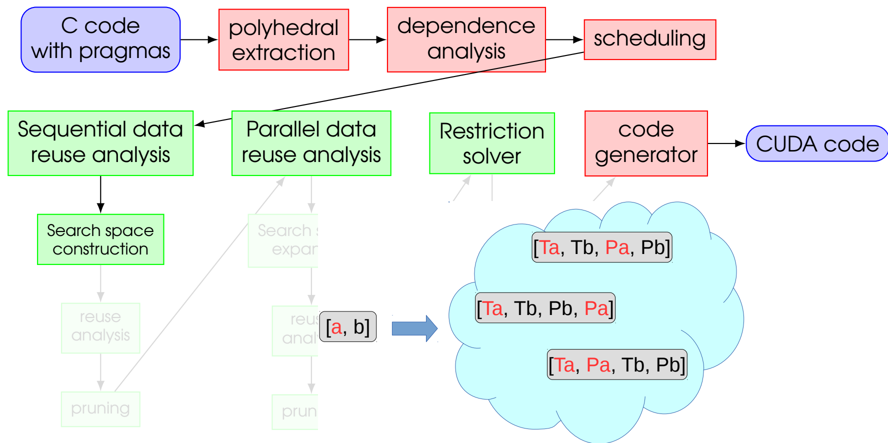
- Memory footprint.
- Accesses cost.

Contributions

- Adapts PPCG schedules to the input's data reuse requirements.
- Computes common GPU parameters based on data reuse.


C code
with pragmas
polyhedral
extraction
dependence
analysis
scheduling
tiling and
GPU mapping
memory
allocation
code
generator
CUDA code


C code
with pragmas
polyhedral
extraction
dependence
analysis
scheduling
Sequential data
reuse analysis
Parallel data
reuse analysis
Restriction
solver
code
generator
CUDA code
Search space
construction
Search space
expansion
Sorting
reuse
analysis
reuse
analysis
Optimization
solver
pruning
pruning


C code with pragmas
polyhedral extraction
dependence analysis
scheduling
Sequential data reuse analysis
Parallel data reuse analysis
Restriction solver
code generator
CUDA code
Search space construction
reuse analysis
pruning
[a, b]
[Ta, Tb, Pa, Pb]
[Ta, Tb, Pb, Pa]
[Ta, Pa, Tb, Pb]

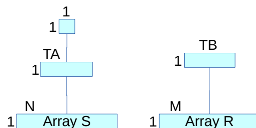
C code with pragmas → polyhedral extraction → dependence analysis → scheduling → Sequential data reuse analysis | Parallel data reuse analysis | Restriction solver | code generator → CUDA code

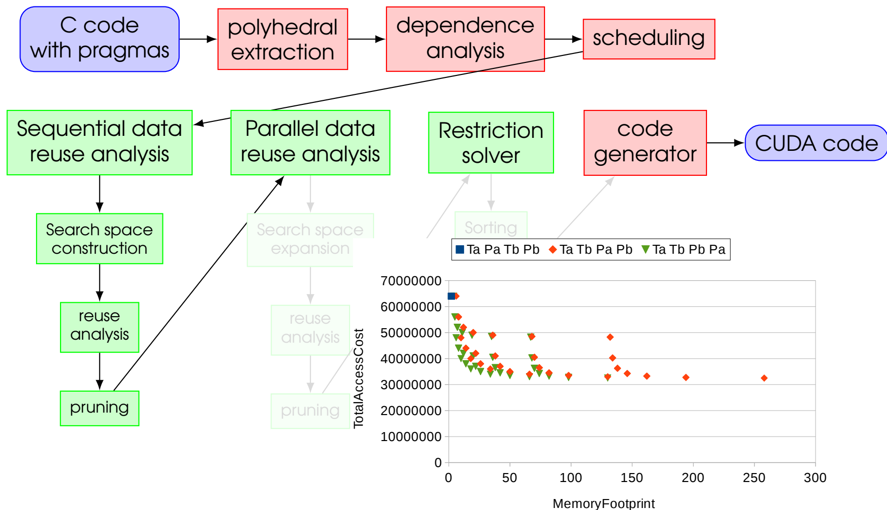
Sequential data reuse analysis → Search space construction → reuse analysis → pruning

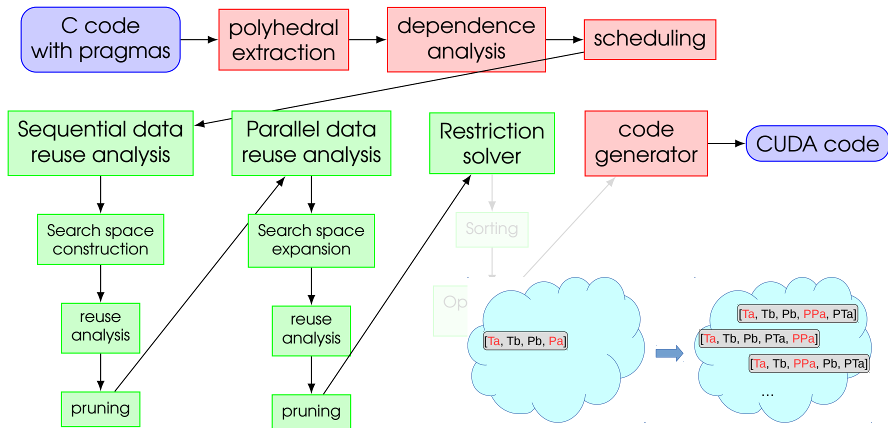
Parallel data reuse analysis → Search space expansion → reuse analysis → pruning

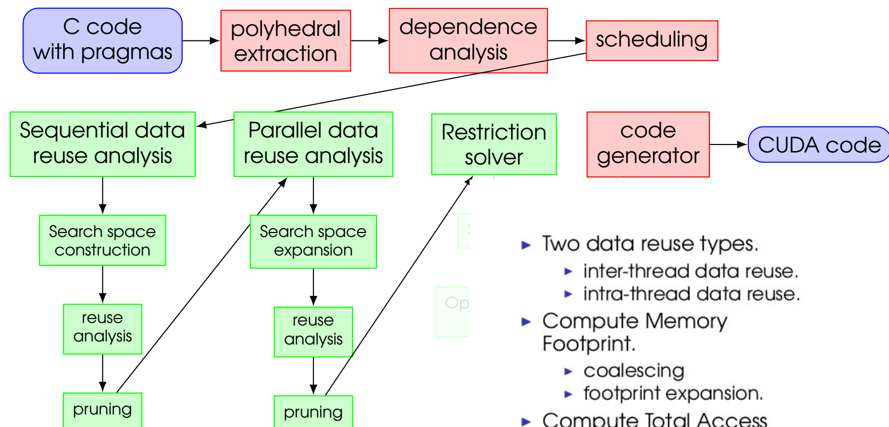
Restriction solver → Sorting

- Compute Memory Footprint.
  - MemoryFootprint = TA + 1 + TB
- Compute Total Access Cost.
  - TotalAccessCost = TA∗(M/TB)∗(TA/N) +...

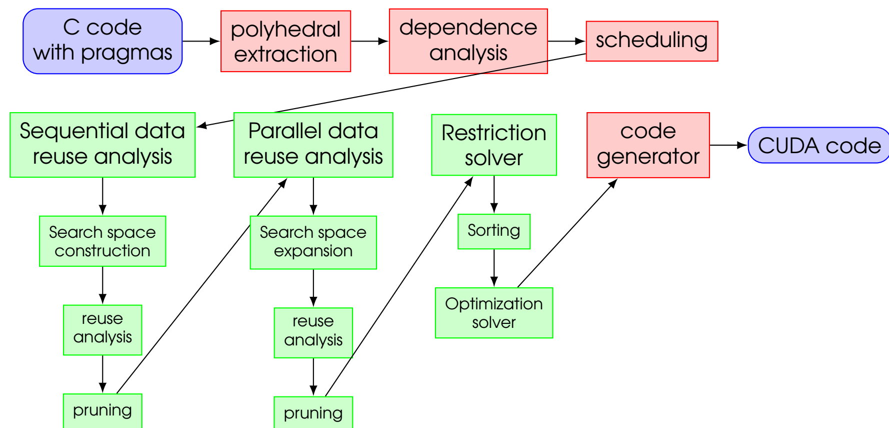



C code with pragmas
polyhedral extraction
dependence analysis
scheduling
Sequential data reuse analysis
Parallel data reuse analysis
Restriction solver
code generator
CUDA code
Search space construction
Search space expansion
Sorting
reuse analysis
reuse analysis
pruning
pruning
Ta Pa Tb Pb
Ta Tb Pa Pb
Ta Tb Pb Pa
TotalAccessCost
MemoryFootprint
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
0
50
100
150
200
250
300


C code
with pragmas
polyhedral
extraction
dependence
analysis
scheduling
Sequential data
reuse analysis
Parallel data
reuse analysis
Restriction
solver
code
generator
CUDA code
Search space
construction
Search space
expansion
Sorting
reuse
analysis
reuse
analysis
pruning
pruning
[Ta, Tb, Pb, Pa]
[Ta, Tb, Pb, PPa, PTa]
[Ta, Tb, Pb, PTa, PPa]
[Ta, Tb, PPa, Pb, PTa]
...



- Two data reuse types.
  - inter-thread data reuse.
  - intra-thread data reuse.
- Compute Memory Footprint.
  - coalescing
  - footprint expansion.
- Compute Total Access Cost.
  - registers are faster than shared mem.


C code with pragmas
polyhedral extraction
dependence analysis
scheduling
Sequential data reuse analysis
Parallel data reuse analysis
Restriction solver
code generator
CUDA code
Search space construction
reuse analysis
pruning
Search space expansion
reuse analysis
pruning
Sorting
Optimization solver

Restriction solver

Goal function

Maximize parallelism.

Restriction types

- Architecture constraints.
- Problem size constraints (when known at compile time).
- Platform reuse constraint.
- Previously built restrictions
  - Tiling constraints.
  - Optimization constraints.

```
max: active_blocks*TTA
active_blocks             <= 8
active_blocks             >= 1
TTA                       <= 1024
TTA                       >= 1
active_blocks*TTA         <= 1536
4000 / TA                 >= 14
4000 % TA                 = 0
4000 % TB                 = 0
TA                        >= 36
TA % TTA                   = 0
TB % TTA                   = 0
TTA % 32                   = 0
active_blocks*(TA)        <= 32768
active_blocks*(TA+TB)     <= 12288
```

Restriction solver

Goal function

Maximize parallelism.

Restriction types

- Architecture constraints.
- Problem size constraints (when known at compile time).
- Platform reuse constraint.
- Previously built restrictions
  - Tiling constraints.
  - Optimization constraints.

```
max: active_blocks*TTA
active_blocks            <= 8
active_blocks            >= 1
TTA                      <= 1024
TTA                      >= 1
active_blocks*TTA        <= 1536
4000 / TA                >= 14
4000 % TA                = 0
4000 % TB                = 0
TA                       >= 36
TA % TTA                  = 0
TB % TTA                  = 0
TTA % 32                  = 0
active_blocks*(TA)       <= 32768
active_blocks*(TA+TB)    <= 12288
```

Restriction solver

Goal function

Maximize parallelism.

Restriction types

- Architecture constraints.
- Problem size constraints (when known at compile time).
- Platform reuse constraint.
- Previously built restrictions
  - Tiling constraints.
  - Optimization constraints.

```
max: active_blocks*TTA
active_blocks           <= 8
active_blocks           >= 1
TTA                     <= 1024
TTA                     >= 1
active_blocks*TTA       <= 1536
4000 / TA               >= 14
4000 % TA               = 0
4000 % TB               = 0
TA                      >= 36
TA % TTA                 = 0
TB % TTA                 = 0
TTA % 32                 = 0
active_blocks*(TA)      <= 32768
active_blocks*(TA+TB)   <= 12288
```

Restriction solver

Goal function

Maximize parallelism.

Restriction types

- Architecture constraints.
- Problem size constraints (when known at compile time).
- Platform reuse constraint.
- Previously built restrictions
  - Tiling constraints.
  - Optimization constraints.

```
max: active_blocks*TTA
active_blocks            <= 8
active_blocks            >= 1
TTA                      <= 1024
TTA                      >= 1
active_blocks*TTA        <= 1536
4000 / TA                >= 14
4000 % TA                = 0
4000 % TB                = 0
TA                       >= 36
TA % TTA                  = 0
TB % TTA                  = 0
TTA % 32                  = 0
active_blocks*(TA)       <= 32768
active_blocks*(TA+TB)    <= 12288
```

Restriction solver

Goal function

Maximize parallelism.

Restriction types

- Architecture constraints.
- Problem size constraints (when known at compile time).
- Platform reuse constraint.
- Previously built restrictions
  - Tiling constraints.
  - Optimization constraints.

```
max: active_blocks*TTA
active_blocks           <= 8
active_blocks           >= 1
TTA                     <= 1024
TTA                     >= 1
active_blocks*TTA       <= 1536
4000 / TA               >= 14
4000 % TA               = 0
4000 % TB               = 0
TA                      >= 36
TA % TTA                 = 0
TB % TTA                 = 0
TTA % 32                 = 0
active_blocks*(TA)      <= 32768
active_blocks*(TA+TB)   <= 12288
```

Environment

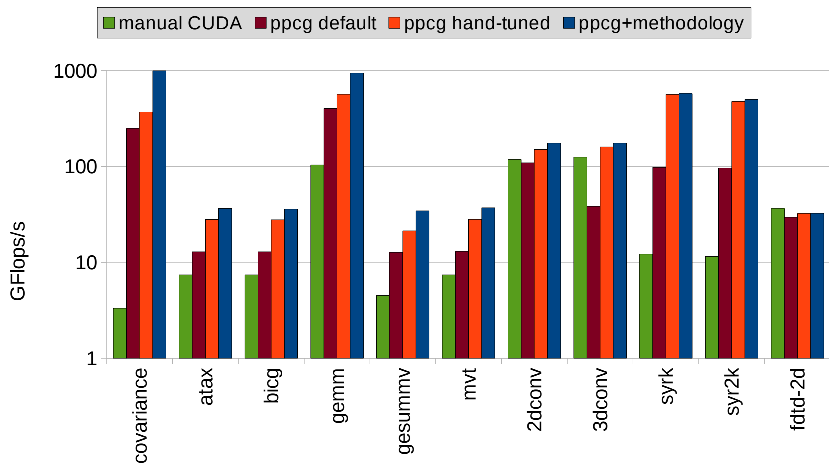
Tested GPUs

- Tesla K20 (*Kepler* architecture).
- Tesla M2070 (*Fermi* architecture).
- Tesla C1060 (*Tesla* architecture).

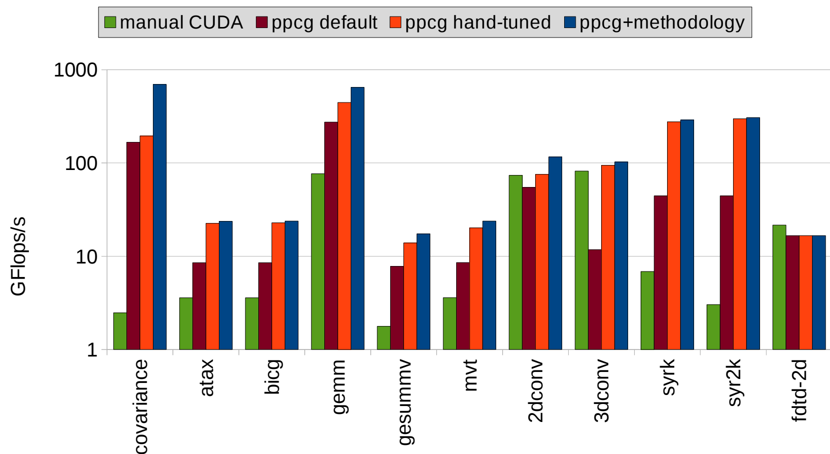
Benchmarks (compiled with CUDA 5.0)

PolyBench-gpu-1.0
(`http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU/index.html`)

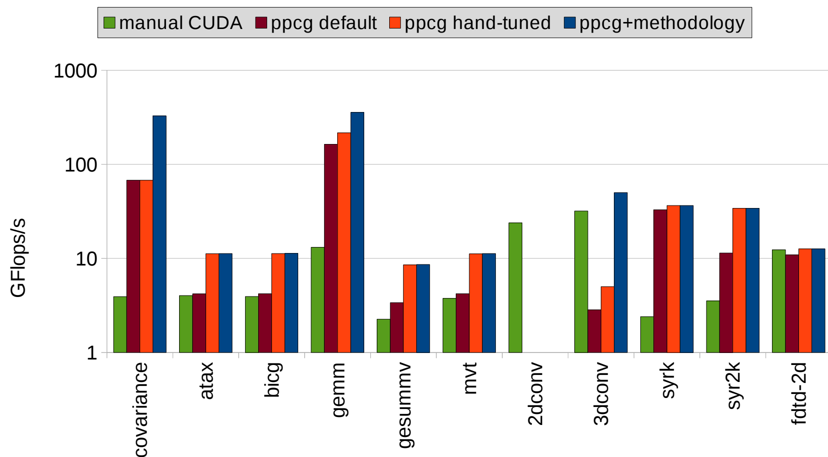
Results on the K20



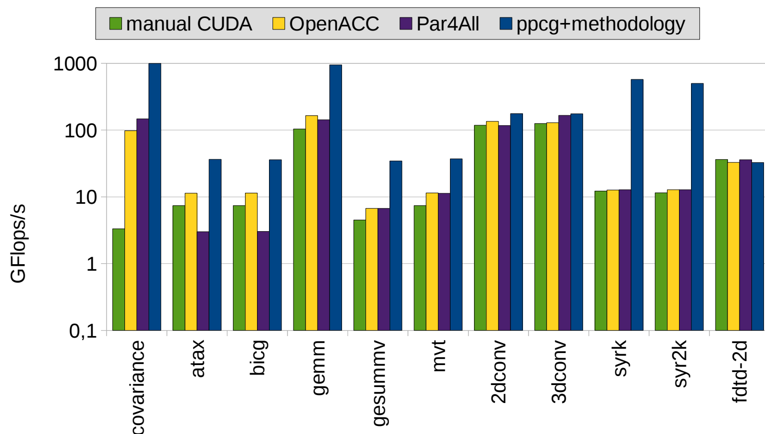
Results on the M2070



Results on the C1060



State-of-art comparison



Conclusions and future work

Conclusions

- Data reuse driven system to improve performance.
- Turns PPCG into a *platform-aware* compiler.
- Performs very well compared against previous ppcg.
  - Speedups: 2.96x, 3.23x, 2.95x (compared to *out-of-the-box* ppcg).

Future work

- Explore the effect of other loop transformations.
- Add even more restrictions to the solver.
- Validate our model with bigger benchs.

Conclusions and future work

Conclusions

- Data reuse driven system to improve performance.
- Turns PPCG into a *platform-aware* compiler.
- Performs very well compared against previous ppcg.
  - Speedups: 2.96x, 3.23x, 2.95x (compared to *out-of-the-box* ppcg).

Future work

- Explore the effect of other loop transformations.
- Add even more restrictions to the solver.
- Validate our model with bigger benchs.

Thanks for your attention!

Any question?

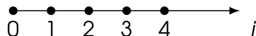
Polyhedral Model

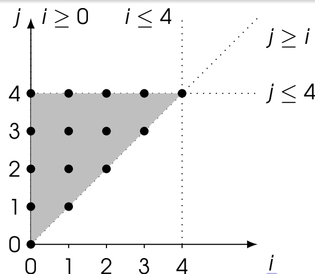
```
    #pragma scop
    for (i=0; i<4; i++){
S0      A[i] = 0;
        for(j=i; j<4; j++)
S1          A[i] += B[i][j];
    }
    #pragma endscop
```

Iteration domain

Contains the dynamic instances of the statements.

$$\{\, \mathrm{S0}(i) \mid 0 \leq i < 5 \,\} \cup$$
$$\{\, \mathrm{S1}(i,j) \mid 0 \leq i < 5 \wedge i \leq j < 5 \,\}$$





Polyhedral Model

```
    #pragma scop
    for (i=0; i<4; i++){
S0     A[i] = 0;
       for(j=i; j<4; j++)
S1        A[i] += B[i][j];
    }
    #pragma endscop
```

Access relations

map statement instances to the array elements.

writes

$$\{\, \mathrm{S0}(i) \rightarrow \mathrm{A}(i) \,\} \cup \{\, \mathrm{S1}(i,j) \rightarrow \mathrm{A}(i) \,\}$$

reads

$$\{\, \mathrm{S1}(i,j) \rightarrow \mathrm{B}(i,j) \,\}$$

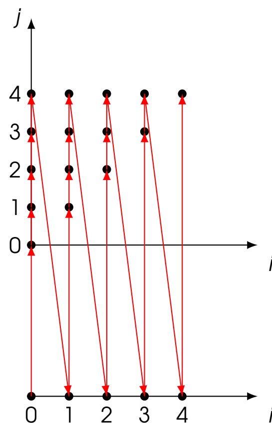
Polyhedral Model

```
        #pragma scop
        for (i=0; i<4; i++){
S0          A[i] = 0;
            for(j=i; j<4; j++)
S1              A[i] += B[i][j];
        }
        #pragma endscop
```

Schedule

specifies the order in which the statement instances are executed.

$$\{\, S0(i) \rightarrow (i, 0, 0) \,\} \cup \{\, S1(i, j) \rightarrow (i, j, 1) \,\}$$

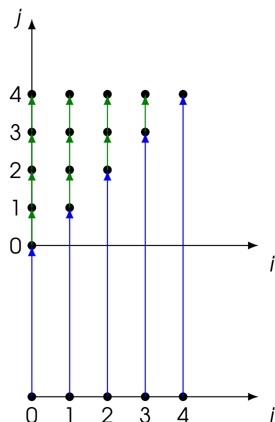


```
    #pragma scop
    for (i=0; i<4; i++){
S0      A[i] = 0;
        for(j=i; j<4; j++)
S1          A[i] += B[i][j];
    }
    #pragma endscop
```

Dependences analysis

dependence relation: determines which statement instances depends on which other statement instances.

$$\{ \mathrm{S0}(i) \rightarrow S1(i, 0) \} \cup \{ \mathrm{S1}(i, j) \rightarrow (i, j + 1) \}$$



```
    #pragma scop
    for (i=0; i<4; i++){
S0      A[i] = 0;
        for(j=i; j<4; j++)
S1          A[i] += B[i][j];
    }
    #pragma endscop
```

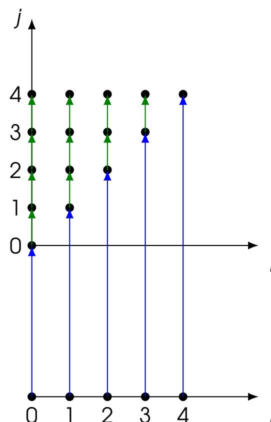
Dependences analysis

dependence relation: determines which statement instances depends on which other statement instances.

$$\{ \mathrm{S0}(i) \rightarrow S1(i,0) \} \cup \{ \mathrm{S1}(i,j) \rightarrow (i,j+1) \}$$

dependence distances: used to know if *schedule dimensions* are parallel and/or tileables.

$$\{ (0,0,1), (0,1,0) \}$$



Scheduling

1. Based on Pluto algorithm.
   - exposes parallelism.
   - exposes temporal locality.
2. Uses isl to build a new affine schedule S:

$$S = (S_i)_{1 \leq i \leq d}$$

3. The affine functions S_i are constructed one by one in such a way that the corresponding dependence distances are non-negative.
   - The sequence of S_is that are constructed in this way form what is known as a *tilable band*, or simply band.
4. Ensures at least one parallel dimension within the band. Parallel dimensions are placed outermost.
5. three scheduling strategies:
   - minimal fusion.
   - maximal fusion.
   - maximize band depth.

Tiling and Mapping to GPU

1. *tiling* splits each dimension loop into a pair of dimensions (loops):
   - tile loop that iterates over different tiles.
   - point loop that iterates inside tiles.
2. dimensions before the outermost band are executed on CPU.
3. tile the outermost band.
   - up to two parallel tile dimensions are mapped to threadblocks.
   - up to three parallel point dimensions are mapped to threads.

Tiling and Mapping to GPU

1. *tiling* splits each dimension loop into a pair of dimensions (loops):
   - tile loop that iterates over different tiles.
   - point loop that iterates inside tiles.
2. dimensions before the outermost band are executed on CPU.
3. tile the outermost band.
   - up to two parallel tile dimensions are mapped to threadblocks.
   - up to three parallel point dimensions are mapped to threads.

Memory allocation

1. Transfers to/from GPU. Done at the beginning/end of the SCoP.
   - arrays accessed inside the SCoP are copied to the GPU.
   - arrays updated inside the SCoP are copied back from the GPU.
   - read-only scalars are passed to kernels as parameters.
   - scalars updated inside the SCoP are treated as zero-dimensional arrays.

Memory allocation

2. group array references to avoid inconsistencies.

```
for(i=0; i<N; i++)
  C[i] = foo(i);

for(i=0; i<N; i++)
  a += C[i];
```

Memory allocation

3. Allocation to registers and shared memory.
   - If we are able to compute register tiles and there is any reuse, then the data is placed in registers.
   - Otherwise, if we are able to compute shared memory tiles and there is any reuse or the original accesses were not coalesced, then we place the data in shared memory.
   - Otherwise, the data is kept in global memory.

Main steps

- Sequential data reuse analysis.
- Parallel data reuse analysis.
- Restriction solver.

Data reuse analysis steps

- Prepare the search space (tiling plus loop interchange).
- Analyze data reuse (and build restrictions).
- Prune the search space.

Restriction solver steps

- Sort out the search space (penalty for *non-coalesced* schedules).
- Solve an optimization problem.

Main steps

- Sequential data reuse analysis.
- Parallel data reuse analysis.
- Restriction solver.

Data reuse analysis steps

- Prepare the search space (tiling plus loop interchange).
- Analyze data reuse (and build restrictions).
- Prune the search space.

Restriction solver steps

- Sort out the search space (penalty for *non-coalesced* schedules).
- Solve an optimization problem.

Main steps

- Sequential data reuse analysis.
- Parallel data reuse analysis.
- Restriction solver.

Data reuse analysis steps

- Prepare the search space (tiling plus loop interchange).
- Analyze data reuse (and build restrictions).
- Prune the search space.

Restriction solver steps

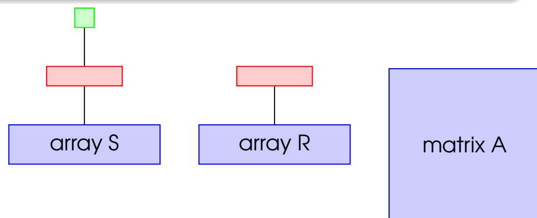
- Sort out the search space (penalty for *non-coalesced* schedules).
- Solve an optimization problem.

Copy candidates construction

Uses the polyhedral representation to compute pieces of data in which exists data reuse

gemm schedule

(Ti, Tj, Tk, Pi, Pj, Pk)

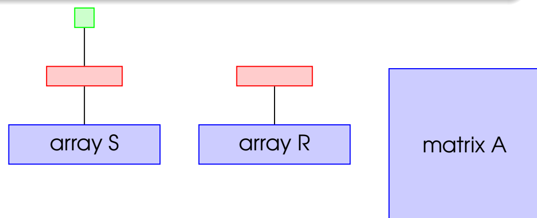


Copy candidates construction

Uses the polyhedral representation to compute pieces of data in which exists data reuse

gemm schedule

(Ti, Tj, Tk, Pi, Pj, Pk)

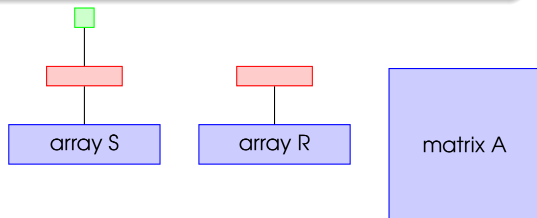


Copy candidates construction

Uses the polyhedral representation to compute pieces of data in which exists data reuse

gemm schedule

(Ti, Tj, Tk, Pi, Pj, Pk)



Platform

Host

- 2xIntel Xeon E5530 @ 2.4GHz.
- 4 cores per chip + HyperThreading $\Rightarrow$ 16 virtual cores.
- GCC 4.6.

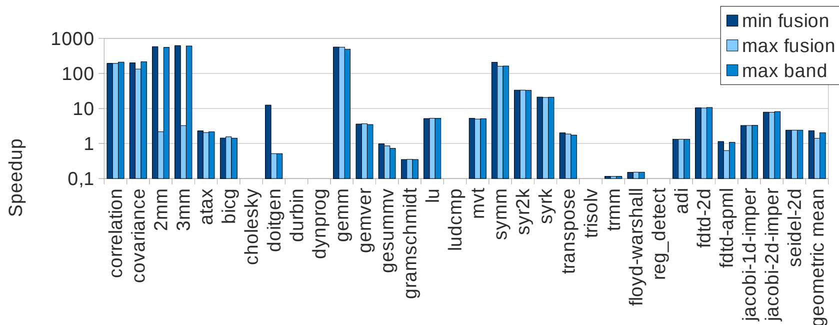
GPU: Tesla M2070

- 14 multiprocessors @ 1.15GHz, 32 cores per multiprocessor.
- 32768 register per multiprocessor.
- 64KB L1 per multiprocessor.
- CUDA 4.0.

Benchmarks

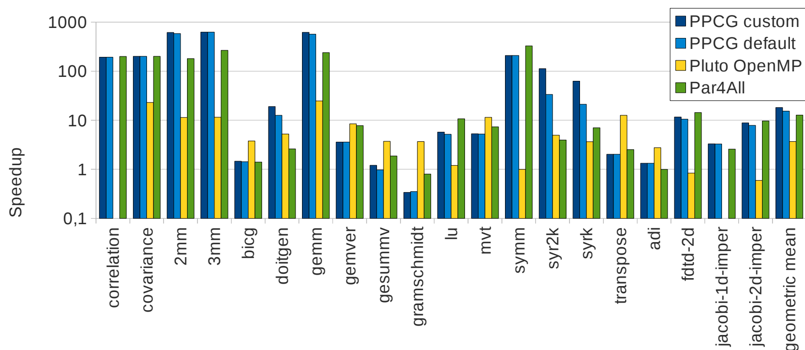
PolyBench-3.1 (`http://artecs.dacya.ucm.es/?q=node/861`).

PPCG scheduling strategies



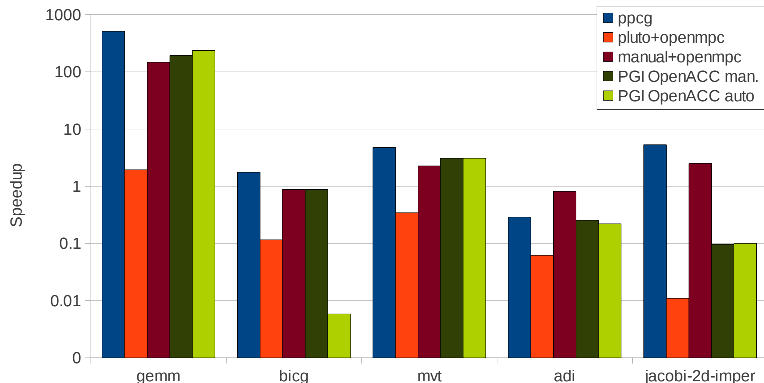
baseline: Sequential execution on CPU.

State-of-art comparison 1/2



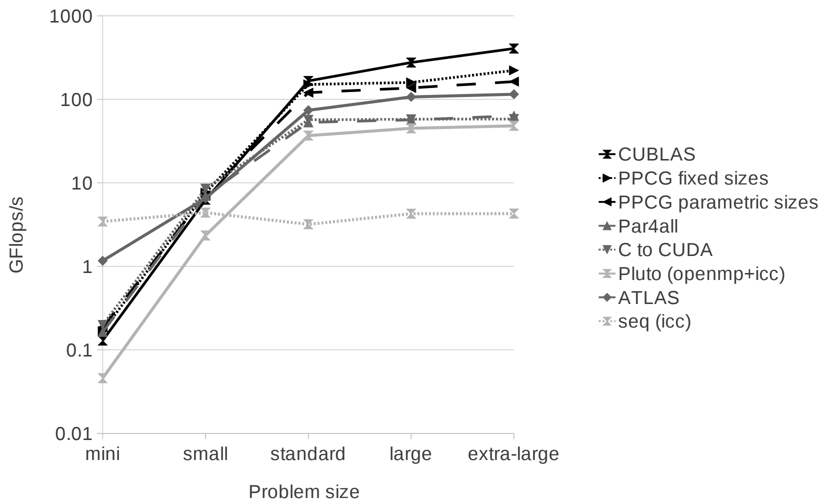
baseline: Sequential execution on CPU.

State-of-art comparison 2/2



baseline: Sequential execution on CPU.

State-of-art comparison: gemm



- Compute Memory Footprint.
  - MemoryFootprint = TA + 1 + TB
- Compute Total Access Cost.
  - TotalAccessCost = TA∗(M/TB)∗(TA/N) +

- Two data reuse types.
  - inter-thread data reuse.
  - intra-thread data reuse.
- Compute Memory Footprint.
  - coalescing
  - footprint expansion.
- Compute Total Access Cost.
  - registers are faster than shared mem.