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

Carlos Juega\textsuperscript{1}  José Ignacio Gómez\textsuperscript{1}  Christian Tenllado\textsuperscript{1}  Francky Catthoor\textsuperscript{2}

\textsuperscript{1}ArTeCS, Universidad Complutense de Madrid
\textsuperscript{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.

<table>
<thead>
<tr>
<th>gemm</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>40 ms</td>
</tr>
<tr>
<td>↓ parallelism, shared memory</td>
<td>12.15 ms</td>
</tr>
<tr>
<td>↓↓ parallelism, shared memory + registers</td>
<td>9.36 ms</td>
</tr>
</tbody>
</table>
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, ...).
Source-to-source compiler based on Polyhedral Model.

Contributions

1. Exposes parallelism.
3. Exploits shared memory and registers.

C code with pragmas → polyhedral extraction → dependence analysis → scheduling → tiling and GPU mapping → memory allocation → code generator → CUDA code
Polyhedral Parallel Code Generator

*Source-to-source* compiler based on Polyhedral Model.

**Contributions**

1. Exposes parallelism.
3. Exploits shared memory and registers.

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

Contributions

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

Heuristics

- Memory footprint.
- Accesses cost.
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.

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

Heuristics
- Memory footprint.
- Accesses cost.
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

Sequential data reuse analysis
- Search space construction
  - reuse analysis
    - pruning
- Restriction solver
  - Sorting
    - Optimization solver

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

Code generator → CUDA code
Motivation PPCG Adaptive Mapping Results Conclusions

Proposal Overview

C code with pragmas
- polyhedral extraction
- dependence analysis
- scheduling

Sequential data reuse analysis
- Search space construction
- Search space expansion
- reuse analysis
- pruning

Parallel data reuse analysis
- Search space construction
- Search space expansion
- reuse analysis
- pruning

Restriction solver
- [a, b]

Code generator
- [Ta, Tb, Pa, Pb]
- [Ta, Tb, Pb, Pa]
- [Ta, Pa, Tb, Pb]

CUDA code

Carlos Jueba et al.
Adaptive Mapping and Parameter Selection
C code with pragmas \( \rightarrow \) polyhedral extraction \( \rightarrow \) dependence analysis \( \rightarrow \) scheduling

Sequential data reuse analysis \( \rightarrow \) Search space construction \( \rightarrow \) reuse analysis \( \rightarrow \) pruning

Parallel data reuse analysis \( \rightarrow \) Search space expansion \( \rightarrow \) reuse analysis \( \rightarrow \) pruning

Restriction solver \( \rightarrow \) code generator

CUDA code

Compute Memory Footprint.
- \( \text{MemoryFootprint} = \text{TA} + 1 + \text{TB} \)

Compute Total Access Cost.
- \( \text{TotalAccessCost} = \text{TA} \times (\text{M/\text{TB}}) \times (\text{TA/N}) + \ldots \)
Motivation

Proposition Overview

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

Reuse analysis

pruning

Carlos Juega et al.

Adaptive Mapping and Parameter Selection
C code with pragmas → polyhedral extraction → dependence analysis → scheduling

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

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

Restriction solver → code generator → CUDA code

[\{Ta, Tb, Pb, Pa\}, \{Ta, Tb, Pb, PTa, PPa\}, \{Ta, Tb, Pb, PPa, PTa\}, \{Ta, Tb, PPa, Pb, PTa\}, ...]
- C code with pragmas
- polyhedral extraction
- dependence analysis
- scheduling

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

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

Restriction solver
- code generator
- CUDA code

- 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.
Motivation
PPCG
Adaptive Mapping
Results
Conclusions
Proposal
Overview

C code with pragmas → polyhedral extraction → dependence analysis → scheduling

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

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

Restriction solver
Sorting → Optimization solver

code generator → CUDA code

Carlos Juega et al.
Adaptive Mapping and Parameter Selection
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.

Goal function:
\[
\text{max: active_blocks} \times \text{TTA}
\]

Constraints:
- \(\text{active_blocks} \leq 8\)
- \(\text{active_blocks} \geq 1\)
- \(\text{TTA} \leq 1024\)
- \(\text{TTA} \geq 1\)
- \(\text{active_blocks} \times \text{TTA} \leq 1536\)
- \(\frac{4000}{\text{TA}} \geq 14\)
- \(4000 \mod \text{TA} = 0\)
- \(4000 \mod \text{TB} = 0\)
- \(\text{TA} \geq 36\)
- \(\text{TA} \mod \text{TTA} = 0\)
- \(\text{TB} \mod \text{TTA} = 0\)
- \(\text{TTA} \mod 32 = 0\)
- \(\text{active_blocks} \times (\text{TA}) \leq 32768\)
- \(\text{active_blocks} \times (\text{TA}+\text{TB}) \leq 12288\)
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.

\[
\begin{align*}
\text{max: } & \text{active_blocks} \times \text{TTA} \\
\text{active_blocks} & \leq 8 \\
\text{active_blocks} & \geq 1 \\
\text{TTA} & \leq 1024 \\
\text{TTA} & \geq 1 \\
\text{active_blocks} \times \text{TTA} & \leq 1536 \\
4000 & \div \text{TA} \geq 14 \\
4000 & \mod \text{TA} = 0 \\
4000 & \mod \text{TB} = 0 \\
\text{TA} & \geq 36 \\
\text{TA} \mod \text{TTA} & = 0 \\
\text{TB} \mod \text{TTA} & = 0 \\
\text{TTA} \mod 32 & = 0 \\
\text{active_blocks} \times (\text{TA}) & \leq 32768 \\
\text{active_blocks} \times (\text{TA}+\text{TB}) & \leq 12288
\end{align*}
\]
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
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 \leq 8
active_blocks \geq 1
TTA \leq 1024
TTA \geq 1
active_blocks*TTA \leq 1536
4000 / TA \geq 14
4000 \% TA = 0
4000 \% TB = 0
TA \geq 36
TA \% TTA = 0
TB \% TTA = 0
TTA \% 32 = 0
active_blocks*(TA) \leq 32768
active_blocks*(TA+TB) \leq 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

Carlos Juega et al.
Adaptive Mapping and Parameter Selection
Results on the M2070

![Graph showing performance comparison]

- **GFlops/s**
- **Manual CUDA**
- **ppcg default**
- **ppcg hand-tuned**
- **ppcg+methodology**

Carlos Juega et al.
Results on the C1060

![Graph showing performance metrics for different operations on the C1060]
State-of-art comparison

- Manual CUDA
- OpenACC
- Par4All
- ppcg+methodology

GFlop/s
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 benches.
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 benches.
Thanks for your attention!

Any question?
#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.

\[
\{ S0(i) \mid 0 \leq i < 5 \} \cup \\
\{ S1(i,j) \mid 0 \leq i < 5 \land i \leq j < 5 \}
\]
Polyhedral Model

```c
#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**

\[
\{ S0(i) \rightarrow A(i) \} \cup \{ S1(i,j) \rightarrow A(i) \}
\]

**reads**

\[
\{ S1(i,j) \rightarrow B(i,j) \}
\]
Polyhedral Model

```c
#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.

\[
\{ S0(i) \rightarrow (i, 0, 0) \} \cup \{ S1(i, j) \rightarrow (i, j, 1) \}
\]
```c
#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.

\[
\{ S0(i) \rightarrow S1(i, 0) \} \cup \{ S1(i, j) \rightarrow (i, j + 1) \}
\]
#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.

\{ S0(i) \rightarrow S1(i,0) \} \cup \{ 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 $S_i$s 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.

```c
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

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

array S

array R

matrix A
Copy candidates construction

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

gemm schedule

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

array S

array R

matrix A
Platform

Host
- 2x Intel Xeon E5530 @ 2.4GHz.
- 4 cores per chip + HyperThreading ⇒ 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
PPCG scheduling strategies

baseline: Sequential execution on CPU.
baseline: Sequential execution on CPU.
State-of-art comparison 2/2

baseline: Sequential execution on CPU.
State-of-art comparison: gemm

![Graph showing performance comparison for different methods across various problem sizes](image-url)
Compute Memory Footprint.
- MemoryFootprint = TA + 1 + TB

Compute Total Access Cost.
- TotalAccessCost = TA \times (M/TB) \times (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.