Optimization Space Pruning without Regrets

Ulysse Beaugnon, Jacques Pienaar, Albert Cohen, Marc Pouzet, Antoine Pouille

École Normale Supérieure, INRIA

5th February 2017, CC’17
Graphic Processing Units (GPUs)

Strengths
- High parallelism
- High memory bandwidth
- High peak GFlop/Watt ratio
- Hide latency with parallelism

Critical for linear algebra, deep learning, image processing, . . .

Need to generate code that fully exploits the power of GPUs
How to Find the Best Implementation?

Kernel → ?

How to implement a given kernel on GPU?
How to Find the Best Implementation?

Kernel → ?

Loop Parallelisation?

Parallelism Levels?
How to Find the Best Implementation?

Kernel → ?

Unrolling?  Loop Parallelisation?

Vectorization?

Parallelism Levels?
How to Find the Best Implementation?

Unrolling?  Loop Parallelisation?

Tiling?

Kernel → ?

Thread Blocking?

Vectorization?

Parallelism Levels?

Tiling Factor?
How to Find the Best Implementation?

Unrolling?  Loop Parallelisation?

Execution Order?  Tiling?

Kernel  

Thread Blocking?  Vectorization?

Nesting Order?

Parallelism Levels?

Tiling Factor?
How to Find the Best Implementation?

- Unrolling?
- Loop Parallelisation?
- Execution Order?
- Tiling?
- Loop Fusion?

Kernel → ?

- Thread Blocking?
- Vectorization?
- Nesting Order?
- Parallelism Levels?
- Tiling Factor?
How to Find the Best Implementation?

Kernel → ?

Synchronization? Unrolling? Loop Parallelisation?
Execution Order? Tiling?
Loop Fusion?
Thread Blocking?
Vectorization?
Nesting Order?
Parallelism Levels?
Tiling Factor?
How to Find the Best Implementation?

Synchronization? Unrolling? Loop Parallelisation?

Execution Order? Tiling?

Loop Fusion?

Kernel → ?

Scratchpad Memory?

Thread Blocking? Vectorization?

Nesting Order? Caching Level?

Parallelism Levels? Tiling Factor?
How to Find the Best Implementation?

- Synchronization?
- Unrolling?
- Loop Parallelisation?
- Execution Order?
- Loop Fusion?
- Tiling?
- Scratchpad Memory?
- Thread Blocking?
- Vectorization?
- Nesting Order?
- Caching Level?
- Parallelism Levels?
- Tiling Factor?
Problem Statement

Given
1. A kernel to implement
2. A sample input
3. A search space

Find
▶ the fastest implementation
▶ optimized for given input
▶ in the given search space

Focus on regular code
▶ Perfectly nested loop without if-conditions

All possible implementations are known upfront
▶ List available choices for each implementation decision
▶ Contrasts with rewrite-rule approaches
Existing Solutions

**Exhaustive Evaluation:** Search Space too big

**Analytical Heuristics:** Far from optimal performance on GPU
  ▶ Does not take Evaluation bottleneck into account

**Stochastic Search:** Usually good, but not optimal
  ▶ May miss the best implementation by far

**Manual Implementation:** Optimal but time consuming
  ▶ Provided by GPU vendors only for most important kernels
  ▶ Not scalable to many problem sizes or many architectures
Questions to answer

How to find the exact best implementation?
- Must guarantee it is the fastest in the search space
- Cannot evaluate all the candidate implementations

How to avoid enumerating all the candidate implementations?
- Enumerating the candidates takes too much time
Questions to answer

How to find the exact best implementation?
  ▶ Must guarantee it is the fastest in the search space
  ▶ Cannot evaluate all the candidate implementations

⇒ Need to prune candidates without missing the best one

How to avoid enumerating all the candidate implementations?
  ▶ Enumerating the candidates takes too much time

⇒ Need to prune many candidates at once

⇒ Branch and Bound Pruning Algorithm
Solution: Use a performance model to prune the search space

Kernel + Available Decisions + Input Size = Performance Bound
1. Give a lower bound on the execution time
   ▶ Can safely prune if a better candidate is known
Key Ideas

1. Give a lower bound on the execution time
   - Can safely prune if a better candidate is known

2. Bound a whole part of the search space at once
   - Kernel + Decisions List = Partially Specified Implementation
   - Prune many candidates at once
Search Tree: Recursively Split the Search Space

Full Search Space
Search Tree: Recursively Split the Search Space

Full Search Space

$\text{loop}_0$ unrolled

$\text{loop}_0$ not unrolled
Search Tree: Recursively Split the Search Space

Full Search Space

$\text{loop}_0 \text{ unrolled}$

$\text{loop}_0 \text{ not unrolled}$

Partially Specified Implementation
Search Tree: Recursively Split the Search Space

- Full Search Space
  - $\text{loop}_0$ unrolled
  - $\text{loop}_0$ not unrolled
  - Partially Specified Implementation
- ...
Search Tree: Recursively Split the Search Space

- **Full Search Space**
- **loop₀ unrolled**
- **loop₀ not unrolled**
- **Partially Specified Implementation**
- **Candidate Implementation**
Search Tree Pruning: Branch and Bound

Bound with the Performance Model

...
Search Tree Pruning: Branch and Bound

Execution Time

Bound with the Performance Model

≥ 5

≥ 10

≥ 15

...
Search Tree Pruning: Branch and Bound

- Execution Time
- Bound with the Performance Model
- Evaluate on the GPU
Search Tree Pruning: Branch and Bound

≥ 5

≥ 10

≥ 15

← Execution Time

Bound with the Performance Model

Evaluate on the GPU

Current Best Candidate

Evaluate on the GPU
Search Tree Pruning: Branch and Bound

≥ 5

≥ 10

≥ 15

← Execution Time

Bound with the Performance Model

Safely Cut when Bound ≥ Best Candidate

Evaluate on the GPU

Current Best Candidate

Evaluate on the GPU

10 / 28
Key Technical Ingredients

1. Partially Specified Implementations
   - How to describe possible implementations?
   - How to ensure decisions are compatible?

2. Performance Model

3. Branch and Bound Algorithm
Search Space Representation

List available choices for each decision

Kernel

# in: A, out: B
# Load A into x

d_0: for i in 0..4:
   i_0: x[i] = A[i]

   # Use x to compute B

d_1: for i in 0..N:

d_2: for j in 0..4:
   i_1: y = x[j] + i
   i_2: B[i][j] = y

A flag for each memory access:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_0</td>
<td>L1, L2, RAM</td>
</tr>
<tr>
<td>i_2</td>
<td>L2, RAM</td>
</tr>
</tbody>
</table>
Search Space Representation

List available choices for each decision

Kernel

# in: A, out: B
# Load A into x

d₀:  for i in 0..4:
    i₀:      x[i] = A[i]
    # Use x to compute B

d₁:  for i in 0..N:

d₂:  for j in 0..4:
    i₁:      y = x[j] + i
    i₂:      B[i][j] = y

An implementation for each loop:

<table>
<thead>
<tr>
<th>Loop</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>d₀</td>
<td>P, T, B, U, V</td>
</tr>
<tr>
<td>d₁</td>
<td>P, B</td>
</tr>
<tr>
<td>d₂</td>
<td>P, T, B, U</td>
</tr>
</tbody>
</table>

P  Plain loop
T  Mapped to a thread dimension
B  Mapped to a block dimension
U  Fully unrolled
V  Replaced by vector instructions
Search Space Representation

List available choices for each decision

**Kernel**

```plaintext
# in: A, out: B
# Load A into x

d0: for i in 0..4:
i0: x[i] = A[i]
   # Use x to compute B

d1: for i in 0..N:

d2: for j in 0..4:
i1: y = x[j] + i
i2: B[i][j] = y
```

**A sequential or nesting order between each pair of loop and instruction:**

<table>
<thead>
<tr>
<th></th>
<th>d0</th>
<th>d1</th>
<th>d2</th>
<th>i0</th>
<th>i1</th>
<th>i2</th>
</tr>
</thead>
<tbody>
<tr>
<td>d0</td>
<td>/</td>
<td>I, B</td>
<td>B, F</td>
<td>O</td>
<td>O, B</td>
<td>O, B</td>
</tr>
<tr>
<td>d1</td>
<td>O, A</td>
<td>/</td>
<td>I, O</td>
<td>O, A</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>d2</td>
<td>A, F</td>
<td>I, O</td>
<td>/</td>
<td>O, A</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>i0</td>
<td>I</td>
<td>I, B</td>
<td>I, B</td>
<td>/</td>
<td>B, A</td>
<td>B</td>
</tr>
<tr>
<td>i1</td>
<td>I, A</td>
<td>I</td>
<td>I</td>
<td>B, A</td>
<td>/</td>
<td>B</td>
</tr>
<tr>
<td>i2</td>
<td>I, A</td>
<td>I</td>
<td>I</td>
<td>A</td>
<td>A</td>
<td>/</td>
</tr>
</tbody>
</table>

- I The first is nested in the second
- O The second is nested in the first
- A The first is after the second
- B The first is before the second
- F The two loops are fused
### Search Space Representation

List available choices for each decision

#### Kernel

- # in: A, out: B
- # Load A into x
- \(d_0\): for \(i\) in 0..4:
  - \(i_0\): \(x[i] = A[i]\)
- # Use x to compute B
- \(d_1\): for \(i\) in 0..\(N\):
- \(d_2\): for \(j\) in 0..4:
  - \(i_1\): \(y = x[j] + i\)
  - \(i_2\): \(B[i][j] = y\)

#### A storage for local arrays:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x[i])</td>
<td>R, S, G</td>
</tr>
</tbody>
</table>

- R use registers
- S use an array in scratchpad memory
- G use an array in global memory
Constraint Propagation

Make a decision $\iff$ Restrict a list of alternatives

Not all combinations of choices are valid
  - Must enforce constraints between decisions
  - When a decision is made, restrict other choices accordingly

Example of constraint: $\forall d_0, d_1, d_2$ three loops,

\[ d_0 \text{ nested in } d_1 \land d_1 \text{ nested in } d_2 \implies d_0 \text{ nested in } d_2 \]
Key Technical Ingredients

1. Partially Specified Implementations

2. Performance Model
   - How to give a correct lower bound?
   - How to deal with open implementation decisions?

3. Branch and Bound Algorithm
Performance Model: Lower Bound

Look independently at each hardware bottleneck

Within a single thread:
- Longest dependency chain
- Pressure on instruction issue
- Pressure on ALUs
- Pressure on FPUs
Performance Model: Lower Bound

Look independently at each hardware bottleneck

Within a single block:
- Execution time of a thread
- Pressure on instruction issue
- Pressure on ALUs
- Pressure on FPUs
- Pressure on memory units
- Pressure on RAM bandwidth
Look independently at each hardware bottleneck

On the whole GPU:

- Number of blocks executed in parallel
- Pressure on instruction issue
- Pressure on ALUs
- Pressure on FPUs
- Pressure on memory units
- Pressure on RAM bandwidth
Performance Model: Unspecified Decisions

Make optimistic assumptions for each bottleneck
- Assume the most optimistic choice for each open decision
- Optimize for a single bottleneck at once
⇒ Valid lower bound

Make optimistic assumptions local to each decision
- Relax consistency among assumptions
- Overapproximate the search space with an hypercube
  - Extends the subspace with invalid candidates
- Can optimize each assumption separately
⇒ Valid lower bound
Performance model

Provides a valid lower bound

- Hardware bottlenecks cannot be overcome
- Missing bottlenecks do not invalidate the bound

The bound can be interpreted

- Help enrich the search space with new optimizations
- Highlight architectural limits
Key Technical Ingredients

1. Partially Specified Implementations

2. Performance Model

3. Branch and Bound Algorithm
   - How to explore the search tree efficiently?
Branch and Bound Algorithm

- Pick an open decision using a heuristic
- Instantiate the decision
- Evaluate on the GPU
- Never consider a node that may later be pruned
- Try to maximize early pruning when picking a decision
Approach implemented in a tool named **Telamon**

 Evaluation on the SGEMM kernel:  

\[ C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C \]

**Handcrafted Search Space**
- 2.7 billion candidate implementations
- Takes 5 hours to enumerate on a 12 core machine
- Only 17,664 remain after pruning
- Best implementation found in 13 minutes \(^1\)

\(^1\)for 1024 × 1024 matrices on a Quadro K4000 GPU
Generated Code Performance on SGEMM

Cublas  Hand-optimized vendor provided implementation
PPCG  Code generator for GPU (heuristics + exhaustive search)
Pruning efficiency

Only 17K out of 2.7B candidates evaluated on the GPU

77% of the candidates pruned in the first 2 levels

For 1024 × 1024 matrices, on a Quadro K4000 GPU
Telamon’s Strengths

Guarantee to Never Prune the Best Candidate
- Combines a lower bound performance model with evaluation
- True even if some parts of the architecture are not modeled

Efficient Early Pruning
- Manipulate partially specified implementations
- No need to enumerate candidate implementations

The Model Provides Information on the Search Space
- Helps enrich the search space with new optimizations
- Highlight architectural bottlenecks
Future Work

Use a DSL to describe constraints between optimization choices

- Constraint propagation code is redundant and hard to write
- Allow fast prototyping of the search space representation

Improve the existing implementation

- Port to new architectures
- Improve the performance model
- Express new optimizations in the search space
Questions?

Key Ideas

- Predict a lower bound on the execution time
- Enables a branch and bound search
- Prune early on partially specified implementations
- Guarantee the best candidate is never pruned

Want more information? ulysse.beaugnon@ens.fr