Additive compilation to achieve high-performance on GPUs

Ulysse Beaugnon¹, Basile Clément², Albert Cohen¹, Andi Drebes², Nicolas Tollenaere³
October 5, 2020

¹Google
²Inria et École Normale Supérieure
³Inria et Université Grenoble-Alpes
Achieving high performance on GPUs
GPUs are designed for **throughput** of **highly parallel** computations
GPUs are designed for **throughput** of **highly parallel** computations.

On my laptop:

- 8 SMs
  - × 128 compute cores per SM
  - = 1024 compute cores (1.16 TFLOP/s for $250)

(vs 4 cores × 2 units × 8 vector = 64 on my CPU – 0.25 TFLOP/s for $450)
Figure 1: GPUs devote more transistors to data processing (source: Nvidia)
Hierarchical parallelism

- SIMD model with 2 levels of parallelism
- Blocks are assigned to SMs
- Inside each SM, warps (group of 32 threads) are assigned to schedulers
// i: index variable
// x: array variable
// v: constant value
// P: parameter

// Index expression
ei ::= i | ei + ei | ei * P

// Expression
e ::= x[i, ..., i] | v
    | e - e | e + e | e * e | fma(e, e, e)

// Statement
s ::= x[i, ..., i] = e | i = ei
    | s ; s | loop i in P do s
Case study: matrix multiplication

```plaintext
loop i in N do
    loop j in M do
        C[i, j] = 0;
        loop k in P do
            C[i, j] = fma(C[i, j], A[i, k], B[k, j])
```

Compute-bound:

- \(NP + MP\) loads
- \(NM\) stores
- \(2NMP\) FLOP
Strip-mine the loops to enable parallelism

```plaintext
loop i1 in N/Nt do
    loop i2 in Nt do
        i = i1 * Nt + i2
    loop j1 in M/Mt do
        loop j2 in Mt do
            j = j1 * Mt + j2
            C[i, j] = 0;
        loop k in P do
            C[i, j] = fma(C[i, j], A[i, k], B[k, j])
```
Strip-mine the loops to enable parallelism

```c
loop i1 in N/Nt do
    loop i2 in Nt do
        i = i1 * Nt + i2
    loop j1 in M/Mt do
        loop j2 in Mt do
            j = j1 * Mt + j2
            C[i, j] = 0 ;
        loop k in P do
            C[i, j] = fma(C[i, j], A[i, k], B[k, j])
```

Something missing?
Case study: matrix multiplication

Strip-mine the loops to enable parallelism

\[
\begin{align*}
\text{loop } i_1 \text{ in } N/Nt & \text{ do} \\
& \text{loop } i_2 \text{ in } Nt \text{ do} \\
& \quad i = i_1 \times Nt + i_2 \\
& \text{loop } j_1 \text{ in } M/Mt \text{ do} \\
& \quad \text{loop } j_2 \text{ in } Mt \text{ do} \\
& \quad \quad j = j_1 \times Mt + j_2 \\
& \quad C[i, j] = 0 ; \\
& \text{loop } k \text{ in } P \text{ do} \\
& \quad // C[i, j] += A[i, k] \times B[k, j] \\
& \quad C[i, j] = \text{fma}(C[i, j], A[i, k], B[k, j])
\end{align*}
\]

Something missing? Remainders!
Reorder the loops and use parallelism

```plaintext
loop.block i1 in N/Nt do
    loop.block j1 in M/Mt do
        loop.thread i2 in Nt do
            loop.thread j2 in Mt do
                i = i1 * Nt + i2
                j = j1 * Mt + j2
                C[i, j] = 0
            loop k in P do
                C[i, j] = fma(C[i, j], A[i, k], B[k, j])
```

Are we done?
Case study: matrix multiplication

Are we done?
No. 2NMP loads! 10x slower than CPU.
Shared memory blocking

**Figure 2:** Matrix multiplication with shared memory (Nervana Systems)
Shared memory blocking

```
loop.block i1 in N/32, j1 in M/32 do
    loop.thread i2 in 32, j2 in 32 do
        C[i1 * 32 + i2, j1 * 32 + j2] = 0

loop k1 in P/32 do
    loop.thread k2 in 32, ij2 in 32 do
        As[k2, ij2] = A[i1 * 1024 + ij2, k1 * 32 + k2]
        Bs[k2, ij2] = B[k1 * 32 + k2, j1 * 1024 + ij2]

loop.thread i2 in 32, j2 in 32 do
    i, j = ...
    loop k2 in 32 do
        C[i, j] = fma(
            C[i, j],
            As[k2, i2],
            Bs[k2, j2])
```
Case study: matrix multiplication

What did we do?

- Loop splitting, interchange and fusion
- Parallelization
- Picking tile sizes (surprisingly hard)
- Temporary copies (including layout!)
- Bonus: register allocation, double buffering, ...

All of this is "easy" to do; what is hard is figuring out what to do.
Additive compilation
Compilation as an Optimization Problem

Given:

- A source language $S$
- A program $s$ in language $S$
- A target language $T$
- A concrete machine $M$ to execute $T$

Solve:

$$\arg\max_{t \in T} \text{perf}_M(t)$$

Under the constraint:

$$t \sim s$$
Separate schedule from algorithm (Halide)

Algorithm

Var i, j;
RDom k;
Func P("P"), C("C"));
P(i, j) = 0
P(i, j) += A(i, k)
    * B(k, j)
C(i, j) = P(i, j)

Schedule

C.tile(x, y, xi, yi, 24, 32)
    .fuse(x, y, xy)
    .parallel(xy)
    .vectorize(xi, 8)
    .unroll(xi);

// ...

//...
Vectorizing scalar product (Lift)

\[ \lambda (x, y) \mapsto \text{zip}(x, y) \rightarrow \text{map}(\times) \rightarrow \text{reduce}(+, 0) \]

Rewrite rule

\[ \lambda (x, y) \mapsto \text{zip}(\text{asVector}(n, x), \text{asVector}(n, y)) \rightarrow \text{map}(\text{vectorize}(n, \times)) \rightarrow \text{asScalar} \rightarrow \text{reduce}(+, 0) \]
Vectorizing scalar product (Lift)

\[ \lambda (x, y) \mapsto zip(x, y) \mapsto map(\times) \mapsto reduce(+, 0) \]

Rewrite rule

\[ \lambda (x, y) \mapsto zip(asVector(n, x), asVector(n, y)) \mapsto map(vectorize(n, \times)) \mapsto asScalar \mapsto reduce(+, 0) \]

Code transformations suffer from the *phase ordering* problem.
Code Transformation and Phase Ordering

Vectorizing scalar product (Lift)

\[ \lambda (x, y) \mapsto \text{zip}(x, y) \rightarrow \text{map}(\times) \rightarrow \text{reduce}(+, 0) \]

\[ \lambda (x, y) \mapsto \text{zip} (\text{asVector}(n, x), \text{asVector}(n, y)) \rightarrow \text{map} (\text{vectorize}(n, \times)) \rightarrow \text{asScalar} \rightarrow \text{reduce}(+, 0) \]

Code transformations suffer from the *phase ordering* problem. Can we do better?
Compilation by Refinement

 Algorithm

 Implementations
Compilation by Refinement
Compilation by Refinement

Algorithm

Partial Schedule

Concrete Schedule

Implementations
Choices for Linear Algebra on GPU

- Control flow structure (sequential ordering, nesting and fusion)
  \[ \text{order} : \text{Statements} \times \text{Statements} \rightarrow \{\text{before, after, in, out, merged}\} \]

- Dimensions implementation
  \[ \text{dim\_kind} : \text{Dimensions} \rightarrow \{\text{loop, unroll, vector, thread, block}\} \]

- Mapping to hardware thread dimensions
  \[ \text{thread\_mapping} : \text{StaticDims} \times \text{StaticDims} \rightarrow \{\text{none, same, in, out}\} \]

- Tile sizes
  \[ \text{size} : \text{StaticDims} \rightarrow \mathbb{N} \]

- Memory space
  \[ \text{mem\_space} : \text{Memory} \rightarrow \{\text{global, shared}\} \]

- Cache levels to use
  \[ \text{cache} : \text{MemAccess} \rightarrow \{\text{L1, L2, read\_only, none}\} \]
A recipe

- Define choices (see previous slide)
- Write correctness constraints
- Write a performance model
- Randomly generate schedules (using the performance model)
- Benchmark the schedules
- Pick the best one
Additivity

- The algorithm defines objects (loops, arrays, instructions, ...)
- Objects have properties (order, dim_kind, ...)
- Schedules
  - Add information about property values
  - Add new objects **without losing information on existing objects**
Optimistic performance model

\[ B(c) = 5ms \]

Execution time \( \geq 5ms \)
Branch and Bound

\[ B(c) = 5ms \]

\[ t = 4ms \]
Branch and Bound

\[ B(c) = 5ms \]

\[ t = 4ms \]
Iterative construction of a search tree, focusing on "promising" branches

1. **Descent based on previous iterations**
   - Choice = game (maximize probability to contain the best implementation)
   - Use an appropriate statistical model

2. Heuristic evaluation when first selected (eg random descent)

3. Backpropagate statistics to the parent nodes
Iterative construction of a search tree, focusing on "promising" branches

1. Descent based on previous iterations
   - Choice = game (maximize probability to contain the best implementation)
   - Use an appropriate statistical model

2. Heuristic evaluation when first selected (e.g., random descent)

3. Backpropagate statistics to the parent nodes
Iterative construction of a search tree, focusing on "promising" branches

1. **Descent based on previous iterations**
   - Choice = game (maximize probability to contain the best implementation)
   - Use an appropriate statistical model

2. Heuristic evaluation when first selected (eg random descent)

3. Backpropagate statistics to the parent nodes
Monte Carlo Tree Search

Iterative construction of a search tree, focusing on "promising" branches

1. Descent based on previous iterations
   - Choice = game (maximize probability to contain the best implementation)
   - Use an appropriate statistical model

2. **Heuristic evaluation when first selected (eg random descent)**

3. Backpropagate statistics to the parent nodes
Monte Carlo Tree Search

Iterative construction of a search tree, focusing on "promising" branches

1. Descent based on previous iterations
   - Choice = game (maximize probability to contain the best implementation)
   - Use an appropriate statistical model

2. Heuristic evaluation when first selected (e.g., random descent)

3. Backpropagate statistics to the parent nodes
Model of the Hardware

Hierarchical Parallelism
- $\eta_1$ threads in a block, $\eta_2$ blocks in a kernel
- Limited resources at each level (bottlenecks)
  - e.g. execution units, memory bandwidth

Executes:
- ▢ threads
- □ blocks of threads
- □ the entire kernel
• Recursive model to account for bottlenecks at each parallelism level ("hierarchical roofline")
• Separate model for dependencies within a thread
• Recursive model to account for bottlenecks at each parallelism level ("hierarchical roofline")
• Separate model for dependencies within a thread
Bottleneck Analysis

Lower Bound on the Execution Time of Parallelism Level $i$

$$B_i^r = \frac{\sum_{s \in S} \text{usage}(s, r) \cdot N_i(s)}{\text{resource}(r, i)}$$
Lower Bound on the Execution Time of Parallelism Level $i$

$$B_i^r = \frac{\sum_{s \in S} \text{usage}(s, r) \cdot N_i(s)}{\text{resource}(r, i)}$$

Consumption of resource $r$ by statement $s$
Bottleneck Analysis

Lower Bound on the Execution Time of Parallelism Level $i$

$$B_i^r = \sum_{s \in S} \text{usage}(s, r) \cdot N_i(s) \cdot \text{resource}(r, i)$$

- Consumption of resource $r$ by statement $s$
- Number of instances of statement $s$ at level $i$
Lower Bound on the Execution Time of Parallelism Level $i$

$$B_i^r = \max_{r \in R} \sum_{s \in S} \text{usage}(s, r) \cdot N_i(s)$$

- Consumption of resource $r$ by statement $s$
- Number of instances of statement $s$ at level $i$
- Amount of resource $r$ available at level $i$
Bottleneck Analysis

Lower Bound on the Execution Time of Parallelism Level $i$

$$B_i^r = \max_{r \in R} \sum_{s \in S} \text{usage}(s, r) \cdot N_i(s) / \text{resource}(r, i)$$

- Consumption of resource $r$ by statement $s$
- Number of instances of statement $s$ at level $i$
- Amount of resource $r$ available at level $i$

Optimize $\text{usage}(s, r)$ and $N_i(s)$ separately
⇒ Local Optimistic Assumptions
Bottleneck Analysis

Lower Bound on the Execution Time of Parallelism Level $i$

$$B_i = \max_{r \in R} \sum_{s \in S} \text{usage}(s, r) \cdot N_i(s)$$

- Consumption of resource $r$ by statement $s$
- Number of instances of statement $s$ at level $i$
- Amount of resource $r$ available at level $i$

Optimize $\text{usage}(s, r)$ and $N_i(s)$ separately
$\Rightarrow$ Local Optimistic Assumptions
Bottleneck Model

Specification

\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]
Bottleneck Model

Specification

\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]

Implementations

```python
for i in range(16):
    # Loop I
    for j in range(4):
        # Loop J
        B[i, j] = i * j  # imul

for j in range(4):
    # Loop J
    for i in range(16):
        # Loop I
        B[i, j] = i * j  # imul
```
Bottleneck Model

Specification

\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]

Implementations

```python
for i in range(16):  # Loop I
    for j in range(4):  # Loop J
        B[i, j] = i * j  # imul

for j in range(4):  # Loop J
    for i in range(16):  # Loop I
        B[i, j] = i * j  # imul
```

Minimal number of instances

- \text{imul}: N \geq \text{size}(M) \times \text{size}(N) = 64
- I: N \geq 1 (when outermost)
- J: N \geq 1 (when outermost)
**Bottleneck Model**

\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>imul</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Total**

<table>
<thead>
<tr>
<th></th>
<th>-</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>
Bottleneck Model

\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>imul</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>I (16 iadd)</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>J (4 iadd)</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>-</strong></td>
<td><strong>-</strong></td>
<td><strong>-</strong></td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td><strong>-</strong></td>
<td><strong>1</strong></td>
<td><strong>8</strong></td>
</tr>
</tbody>
</table>

Min cycles: 84 - 27 = 57
Bottleneck Model

\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>imul</td>
<td>64</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>I (16 iadd)</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>J (4 iadd)</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>-</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>imul</td>
<td>64</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>I (16 iadd)</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>J (4 iadd)</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>84</td>
<td>212</td>
</tr>
<tr>
<td>Hardware</td>
<td>-</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>
\[ B\{i : \text{range}(16), j : \text{range}(4)\} = i \times j \]

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>imul</td>
<td>64</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>I (16 iadd)</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>J (4 iadd)</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>84</td>
<td>212</td>
</tr>
<tr>
<td>Hardware</td>
<td>-</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Min cycles</td>
<td>-</td>
<td>84</td>
<td>27</td>
</tr>
</tbody>
</table>
• Recursive model to account for bottlenecks at each parallelism level ("hierarchical roofline")
• Separate model for dependencies within a thread
Parallelism Bound

Lower Bound on the Execution Time

\[ T_{i+1} \geq \max \left( B_{i+1}, B_i \cdot \left\lceil \frac{\eta_i}{\mu_i} \right\rceil \right) \]
Parallelism Bound

Lower Bound on the Execution Time

\[ T_{i+1} \geq \max \left( B_{i+1}, B_i \cdot \left\lceil \frac{\eta_i}{\mu_i} \right\rceil \right) \]

Lower bound of the inner level
Parallelism Bound

Lower Bound on the Execution Time

\[ T_{i+1} \geq \max \left( B_{i+1}, B_i \cdot \left\lceil \frac{\eta_i}{\mu_i} \right\rceil \right) \]

Lower bound of the inner level

Number of instances to execute
Lower Bound on the Execution Time

\[ T_{i+1} \geq \max \left( B_{i+1}, B_i \cdot \left[ \begin{array}{c} \eta_i \\ \mu_i \end{array} \right] \right) \]

- Lower bound of the inner level
- Number of instances to execute
- Number of instances that can execute in parallel

- \( T_{i+1} \): Lower bound on the execution time
- \( B_{i+1} \): Number of instances to execute
- \( B_i \): Number of instances that can execute in parallel
- \( \eta_i \): Lower bound of the inner level
- \( \mu_i \): Number of instances to execute
Parallelism Bound

Lower Bound on the Execution Time

\[ T_{i+1} \geq \max \left( B_{i+1}, B_i \cdot \left\lceil \frac{\eta_i}{\mu_i} \right\rceil \right) \]

Lower bound of the inner level

Number of instances to execute

Number of instances that can execute in parallel

⇒ What about unspecified choices?

- \( B_i \) and \( \eta_i \) depend on how dimensions are parallelized
- Optimizing \( B_i \) and \( \eta_i \) separately incurs too much inaccuracy
Parallelism Bound

Lower Bound on the Execution Time

\[ T_{i+1} \geq \max \left( B_{i+1}, B_i \cdot \frac{\eta_i^{\min}}{\eta_i^{\lcm}} \cdot \left\lceil \frac{\eta_i^{\lcm}}{\mu_i^{\max}} \right\rceil \right) \]

- Optimize \( \mu_i^{\max} \) independently to limit available resources
- Compute \( B_i \) and \( \eta_i^{\min} \) by mapping to lower level(s) when possible
- Compute \( \eta_i^{\lcm} \) by mapping to level \( i \) when possible
### Bottleneck Model

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>imul</td>
<td>64</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>I (16 iadd)</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>J (4 iadd)</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>-</strong></td>
<td><strong>84</strong></td>
<td><strong>212</strong></td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td><strong>-</strong></td>
<td><strong>1</strong></td>
<td><strong>8</strong></td>
</tr>
<tr>
<td><strong>Min cycles</strong></td>
<td><strong>-</strong></td>
<td><strong>84</strong></td>
<td><strong>27</strong></td>
</tr>
</tbody>
</table>

\[ B_{\text{thread}} = 84 \]
### Parallelism Model

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>imul</td>
<td>64</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>-</td>
<td>64</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>-</td>
<td>1</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td><strong>Min cycles</strong></td>
<td>-</td>
<td>64</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

\[ B_{\text{thread}} = 64 \]

*Minimize overhead independently*
## Parallelism Model

<table>
<thead>
<tr>
<th></th>
<th>Instances</th>
<th>Issues</th>
<th>ALU</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>imul</td>
<td>64</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>-</td>
<td>64</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>-</td>
<td>1</td>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>

**Min cycles**

\[ B_{\text{thread}} = 64 \]

Minimize overhead independently

\[ \eta_{\text{thread}}^{\text{min}} = 1 \]

\[ \eta_{\text{thread}}^{\text{lcm}} = 64 \]

\[ B_{\text{block}} = 64 \times \frac{1}{64} \times \left\lceil \frac{64}{32} \right\rceil \]
Parallelism Model

- Doesn’t need to be precise
- Used to prune catastrophic schedules
(best_platform is best generated code on platform)
1024x1024 Sgemm

(best_platform is best generated code on platform)
- **Lift** [2]: Rewrite rules + heuristics + exhaustive search
- **Triton** [3]: Skeleton + exhaustive search
- **AutoTVM** [1] (from [3]): Transformations + statistical cost model
- **Tensor Comprehensions** [4] (from [3]): Polyhedral compilation
Is this correct?
Work in progress — Comments welcome!
Key ideas

- Only check the concrete generated schedule w.r.t. the algorithm (= do not verify the constraints or performance model)
- Use validation: keep enough information in the schedule to map indices back to the original semantic indices
- A hierarchy of language: from generic and structured to concrete
i = ...  

j = ...  

\[ D[i : M, j : N, k : \{-1\}] = 0 \]

\[ D[i : M, j : N, k : P] = \text{fma}( \]

\[ D[i, j, k - 1], A[i, k], B[k, j] \]

\[ C[i : M, j : N] = \text{proj}[k = P - 1](D[i, j, k]) \]

- The union of domains matches the domains in the algorithm
- The domains are covered by the iterations
- Typing rules to ensure dependencies are respected
Typing rule: non-interference

\[
\ell_1, \ldots, \ell_n \subseteq \text{dom}(I) \quad I|_{\ell_1, \ldots, \ell_n}; \Delta \vdash e : \forall \quad x \not\in \Delta \\
I; \Delta \vdash x[\ell_1, \ldots, \ell_n] = e : x[\ell_1, \ldots, \ell_n]
\]

During execution, a memory location \(x[i_1, \ldots, i_n]\) is either undefined or has an unique value matching its definition.
Potentially parallel loop

\[ \forall 0 \leq i < m \quad I, \ell \mapsto u_i; \mu \vdash s \downarrow \mu_i \]
\[ \{u_0, \ldots, u_{m-1}\} = D \quad \mu' = \bigoplus_{0 \leq i < m} \mu_i \]

\[ I; \mu \vdash \text{forall } \ell \text{ in } D \{s\} \downarrow \mu' \]

Not enough...
And more...

- Traditional lowering compiler once the schedule is fixed
- GPU semantics? Hierarchical parallelism
References


Thank you

- **Optimization Space Pruning Without Regrets**, CC 2017
  ⇒ Idea of candidates and primitive lower bound performance model

- **On the Representation of Partially Specified Implementations and its Application to the Optimization of Linear Algebra Kernels on GPU**, preprint arXiv
  ⇒ Formalization of candidates as CSPs and statistical search

- [https://github.com/ulysseB/telamon](https://github.com/ulysseB/telamon)