# Additive compilation to achieve high-performance on GPUs

Ulysse Beaugnon[1], Basile Clément[2], Albert Cohen[1], Andi Drebes[2], Nicolas Tollenaere[3]

October 5, 2020

[1]Google

[2]Inria et École Normale Supérieure

[3]Inria et Université Grenoble-Alpes

# Achieving high performance on GPUs

# GPU architecture 101

GPUs are designed for **throughput** of **highly parallel** computations

# GPU architecture 101

GPUs are designed for **throughput** of **highly parallel** computations

On my laptop:

$$
\begin{aligned}
& 8 \text{ SMs} \\
\times\ & 128 \text{ compute cores per SM} \\
=\ & 1024 \text{ compute cores (1.16 TFLOP/s for \$250)}
\end{aligned}
$$

(vs 4 cores x 2 units x 8 vector = 64 on my CPU – 0.25 TFLOP/s for $450)

# GPU architecture 101

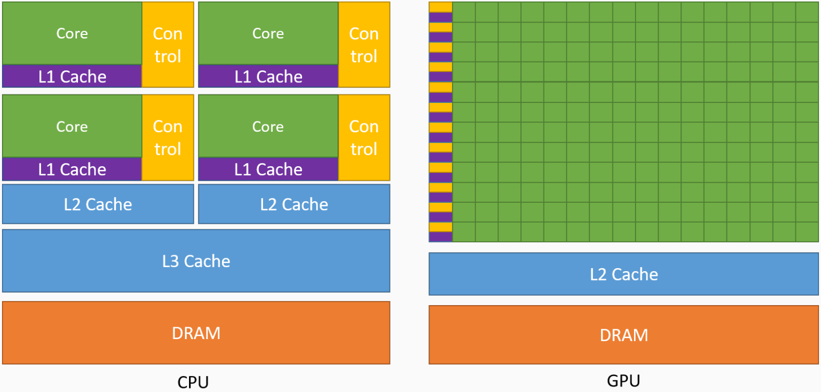
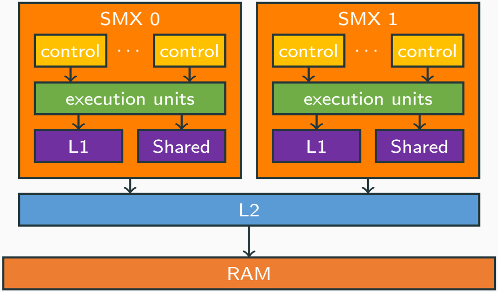



**Figure 1:** GPUs devote more transistors to data processing (source: Nvidia)

# GPU architecture 101




**Executes:**

- threads (32x)
- blocks

**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

# Proto-language

```
// 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

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

Compute-bound:

- $NP + MP$ loads
- $NM$ stores
- $2NMP$ FLOP

# Case study: matrix multiplication

**Strip-mine the loops to enable parallelism**

```
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] += A[i, k] * B[k, j]
          C[i, j] = fma(C[i, j], A[i, k], B[k, j])
```

# Case study: matrix multiplication

**Strip-mine the loops to enable parallelism**

```
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] += A[i, k] * B[k, j]
          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**

```
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] += A[i, k] * B[k, j]
          C[i, j] = fma(C[i, j], A[i, k], B[k, j])
```

Something missing? Remainders!

# Case study: matrix multiplication

**Reorder the loops and use parallelism**

```
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] += A[i, k] * B[k, j]
          C[i, j] = fma(C[i, j], A[i, k], B[k, j])
```

# Case study: matrix multiplication

Are we done?

# Case study: matrix multiplication

Are we done?

No. 2NMP loads! 10x slower than CPU.

# Shared memory blocking

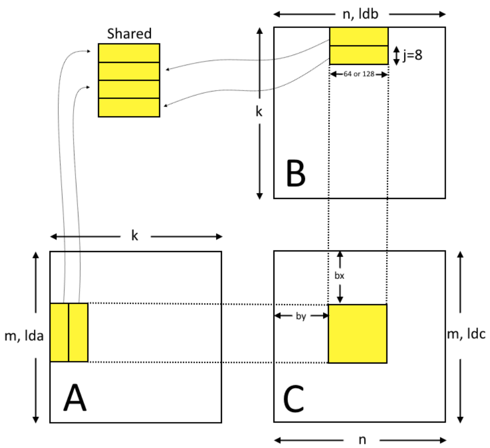



**Figure 2:** Matrix multiplication with shared memory (Nervana Systems)

# 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:

$$argmax_{t \in T} perf_M(t)$$

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) $\mapsto$ zip(x, y) » map($\times$) » reduce($+$, 0)

$\downarrow$ Rewrite rule

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

# Code Transformation and Phase Ordering

**Vectorizing scalar product (Lift)**

```
λ (x, y)  ↦  zip(x, y) » map(×) » reduce(+, 0)
```

↓ Rewrite rule

```
λ (x, y)  ↦  zip(asVector(n, x), asVector(n, y))
             » map(vectorize(n, ×))
             » asScalar
             » reduce(+, 0)
```

Code transformations suffer from the *phase ordering* problem.

# Code Transformation and Phase Ordering

**Vectorizing scalar product (Lift)**

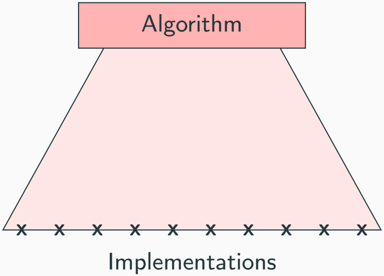
$\lambda$ (x, y) $\mapsto$ zip(x, y) » map($\times$) » reduce(+, 0)

↓ Rewrite rule

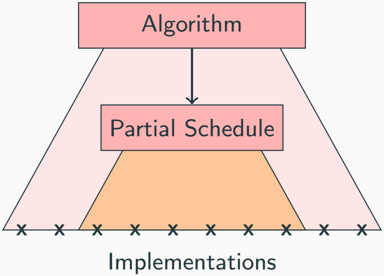
$\lambda$ (x, y) $\mapsto$ zip(asVector(n, x), asVector(n, y))
» map(vectorize(n, $\times$))
» asScalar
» reduce(+, 0)

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

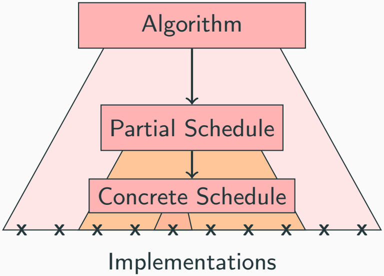
# Compilation by Refinement



# Compilation by Refinement



# Compilation by Refinement



# Choices for Linear Algebra on GPU

- Control flow structure (sequential ordering, nesting and fusion)
  $\texttt{order} : \texttt{Statements} \times \texttt{Statements} \to \{\texttt{before}, \texttt{after}, \texttt{in}, \texttt{out}, \texttt{merged}\}$
- Dimensions implementation
  $\texttt{dim\_kind} : \texttt{Dimensions} \to \{\texttt{loop}, \texttt{unroll}, \texttt{vector}, \texttt{thread}, \texttt{block}\}$
- Mapping to hardware thread dimensions
  $\texttt{thread\_mapping} : \texttt{StaticDims} \times \texttt{StaticDims} \to \{\texttt{none}, \texttt{same}, \texttt{in}, \texttt{out}\}$
- Tile sizes
  $\texttt{size} : \texttt{StaticDims} \to \mathbb{N}$
- Memory space
  $\texttt{mem\_space} : \texttt{Memory} \to \{\texttt{global}, \texttt{shared}\}$
- Cache levels to use
  $\texttt{cache} : \texttt{MemAccess} \to \{\texttt{L1}, \texttt{L2}, \texttt{read\_only}, \texttt{none}\}$

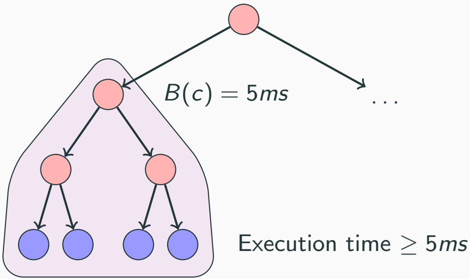
# A recipe

- 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

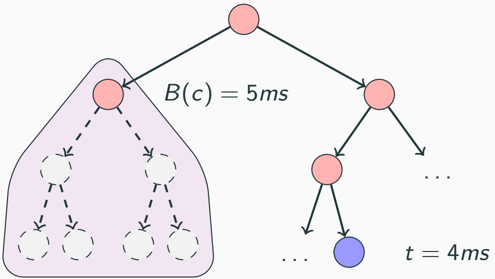
# Additivity

- The algorithm define *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**

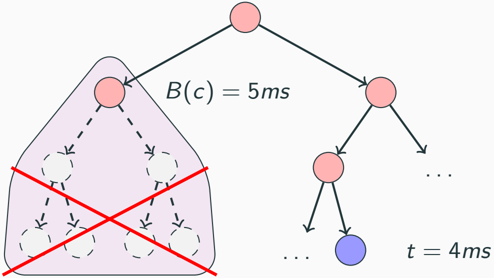
# Optimistic performance model



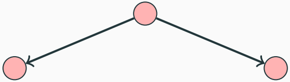
# Branch and Bound



# Branch and Bound



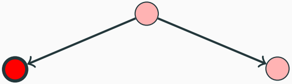
# Monte Carlo Tree Search



**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

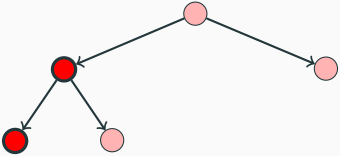
# Monte Carlo Tree Search



**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 (eg random descent)
3. Backpropagate statistics to the parent nodes

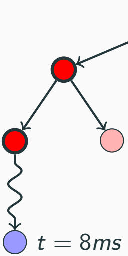
# Monte Carlo Tree Search



**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

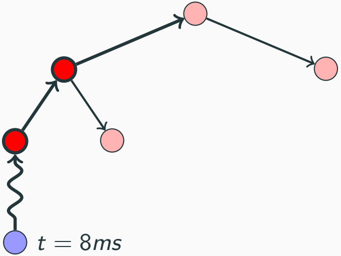
# Monte Carlo Tree Search




**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 (eg random descent)**
3. Backpropagate statistics to the parent nodes

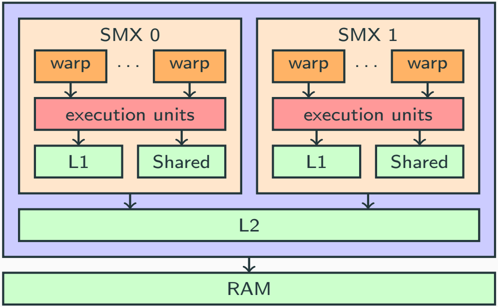
# Monte Carlo Tree Search




**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**

# Model of the Hardware



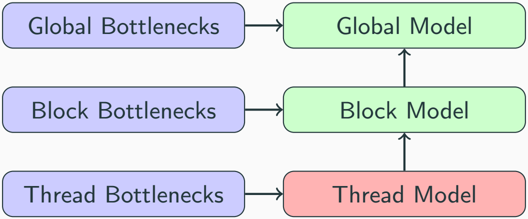

**Executes:**

- threads
- blocks of threads
- the entire kernel

**Hierarchical Parallelism**

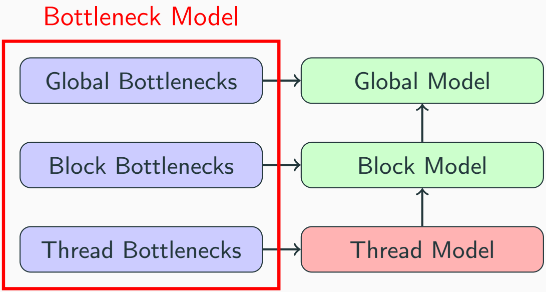
- $\eta_1$ threads in a block, $\eta_2$ blocks in a kernel
- Limited resources at each level (bottlenecks)
  - e.g. execution units, memory bandwidth

# Performance Model




- Recursive model to account for bottlenecks at each parallelism level ("hierarchical roofline")
- Separate model for dependencies within a thread

# Performance Model




- 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 $i$**

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

# Bottleneck Analysis

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

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

Consumption of resource $r$
by statement $s$

# Bottleneck Analysis

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

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

Consumption of resource $r$ by statement $s$

Number of instances of statement $s$ at level $i$

# Bottleneck Analysis

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

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

Consumption of resource $r$ by statement $s$

Amount of resource $r$ available at level $i$

Number of instances of statement $s$ at level $i$

# Bottleneck Analysis

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

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

Consumption of resource $r$ by statement $s$

Amount of resource $r$ available at level $i$

Number of instances of statement $s$ at level $i$

**Optimize** $\texttt{usage}(s, r)$ **and** $N_i(s)$ **separately**
$\Rightarrow$ Local Optimistic Assumptions

# Bottleneck Analysis

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

$$B_i = \max_{r \in \mathcal{R}} \frac{\sum_{s \in \mathcal{S}} \texttt{usage}(s, r) \cdot N_i(s)}{\texttt{resource}(r, i)}$$

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** $\texttt{usage}(s, r)$ **and** $N_i(s)$ **separately**
$\Rightarrow$ Local Optimistic Assumptions

# Bottleneck Model

Specification

$$B\{i : range(16), j : range(4)\} = i * j$$

# Bottleneck Model

Specification

$$B\{i : range(16), j : range(4)\} = i * j$$

Implementations

```
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

$$B\{i : range(16), j : range(4)\} = i * j$$

Implementations

```
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

- `imul`: $N \geq size(\mathtt{M}) \times size(\mathtt{N}) = 64$
- `I`: $N \geq 1$ (when outermost)
- `J`: $N \geq 1$ (when outermost)

# Bottleneck Model

$$B\{i : range(16), j : range(4)\} = i * j$$

| | Instances | Issues | ALU |
|---|---|---|---|
| `iadd` | | 1 | 1 |
| `imul` | | 1 | 3 |
| Total | - | | |
| Hardware | - | 1 | 8 |

# Bottleneck Model

$$B\{i : range(16), j : range(4)\} = i * j$$

| | Instances | Issues | ALU |
|---|---|---|---|
| `iadd` | | 1 | 1 |
| `imul` | | 1 | 3 |
| `I (16 iadd)` | | 16 | 16 |
| `J (4 iadd)` | | 4 | 4 |
| Total | - | | |
| Hardware | - | 1 | 8 |

# Bottleneck Model

$$B\{i : range(16), j : range(4)\} = i * j$$

| | Instances | Issues | ALU |
|---|---|---|---|
| `iadd` | - | 1 | 1 |
| `imul` | 64 | 1 | 3 |
| `I (16 iadd)` | 1 | 16 | 16 |
| `J (4 iadd)` | 1 | 4 | 4 |
| Total | - | | |
| Hardware | - | 1 | 8 |

# Bottleneck Model

$$B\{i : range(16), j : range(4)\} = i * j$$

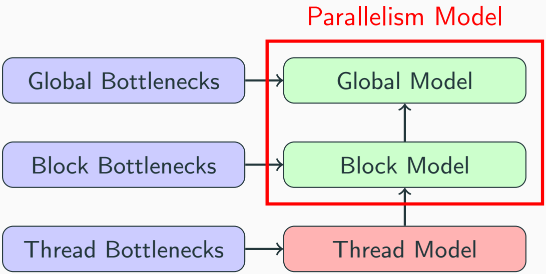
| | Instances | Issues | ALU |
|---|---|---|---|
| `iadd` | - | 1 | 1 |
| `imul` | 64 | 1 | 3 |
| `I (16 iadd)` | 1 | 16 | 16 |
| `J (4 iadd)` | 1 | 4 | 4 |
| Total | - | 84 | 212 |
| Hardware | - | 1 | 8 |

# Bottleneck Model

$$B\{i : range(16), j : range(4)\} = i * j$$

| | Instances | Issues | ALU |
|---|---|---|---|
| `iadd` | - | 1 | 1 |
| `imul` | 64 | 1 | 3 |
| `I (16 iadd)` | 1 | 16 | 16 |
| `J (4 iadd)` | 1 | 4 | 4 |
| Total | - | 84 | 212 |
| Hardware | - | 1 | 8 |
| **Min cycles** | - | **84** | 27 |

# Optimistic Performance Model




- 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

# 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

# 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**

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

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

# Bottleneck Model

| | Instances | Issues | ALU |
|---|---|---|---|
| `imul` | 64 | 1 | 3 |
| `I (16 iadd)` | 1 | 16 | 16 |
| `J (4 iadd)` | 1 | 4 | 4 |
| Total | - | 84 | 212 |
| Hardware | - | 1 | 8 |
| **Min cycles** | - | **84** | 27 |

$$B_{\mathrm{thread}} = 84$$

# Parallelism Model

| | Instances | Issues | ALU | Threads |
|---|---|---|---|---|
| `imul` | 64 | 1 | 3 | |
| `I` | 1 | 0 | 0 | |
| `J` | 1 | 0 | 0 | |
| Total | - | 64 | 192 | |
| Hardware | - | 1 | 8 | 32 |
| **Min cycles** | - | **64** | 24 | |

$$B_{\text{thread}} = 64$$

*Minimize overhead independently*

# Parallelism Model

| | Instances | Issues | ALU | Threads |
|---|---|---|---|---|
| `imul` | 64 | 1 | 3 | |
| `I` | 1 | 0 | 0 | |
| `J` | 1 | 0 | 0 | |
| Total | - | 64 | 192 | |
| Hardware | - | 1 | 8 | 32 |
| **Min cycles** | - | **64** | 24 | |

$$B_{\text{thread}} = 64$$

*Minimize overhead independently*

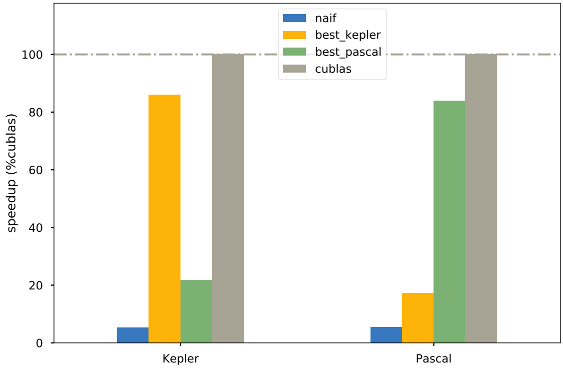
$$\eta_{\text{thread}}^{\text{min}} = 1 \qquad \eta_{\text{thread}}^{\text{lcm}} = 64$$

$$B_{\text{block}} = 64 \times \frac{1}{64} \times \left\lceil \frac{64}{32} \right\rceil$$

# Parallelism Model

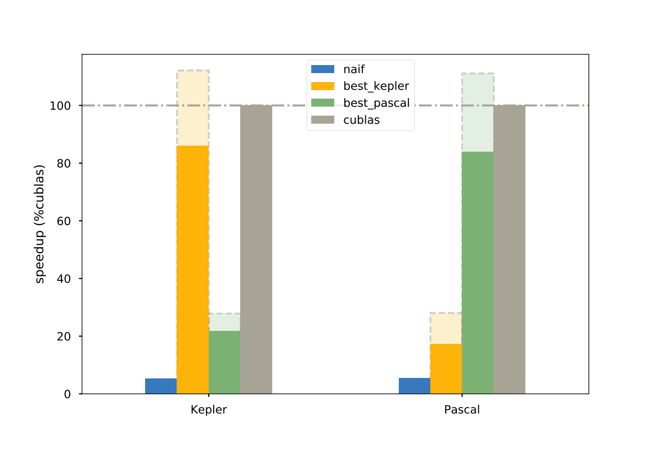
- Doesn't need to be precise
- Used to prune *catastrophic* schedules

# 1024x1024 Sgemm



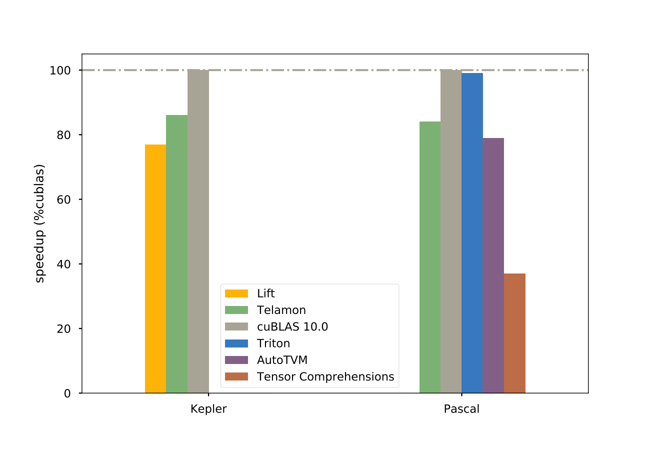

(best_platform is best generated code on platform)

# 1024x1024 Sgemm




(best_platform is best generated code on platform)

# 1024x1024 Sgemm




- 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?

# Formalisation in Coq

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

# Semantic indices and explicit reductions

```
i = ...
j = ...
D[i : M, j : N, k : {-1}] = 0
D[i : M, j : N, k : P] = fma(
  D[i, j, k - 1], A[i, k], B[k, j])
C[i : M, j : N] = 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

$$\frac{\ell_1, \ldots, \ell_n \subseteq \mathrm{dom}(\mathbb{I}) \qquad \mathbb{I}|\ell_1, \ldots, \ell_n; \Delta \vdash e : \mathbb{V} \qquad x \notin \Delta}{\mathbb{I}; \Delta \vdash x[\ell_1, \ldots, \ell_n] = e : x[\ell_1, \ldots, \ell_n]}$$

**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

$$\frac{\forall 0 \leq i < m \quad \mathcal{I}, \ell \mapsto u_i; \mu \vdash s \Downarrow \mu_i \qquad \{u_0, \ldots, u_{m-1}\} = D \qquad \mu' = \bigoplus_{0 \leq i < m} \mu_i}{\mathcal{I}; \mu \vdash \text{forall } \ell \text{ in } D \; \{s\} \Downarrow \mu'}$$

Not enough. . .

# And more...

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

# References


[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, 2018.

[2] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Matrix multiplication beyond auto-tuning: Rewrite-based gpu code generation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 15:1–15:10, New York, NY, USA, 2016. ACM.

# References ii

[3] Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pages 10–19, New York, NY, USA, 2019. ACM.

[4] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

# Thank you

- **Optimization Space Pruning Wihout Regrets**, CC 2017
  $\Rightarrow$ 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
  $\Rightarrow$ Formalization of candidates as CSPs and statistical search
- `https://github.com/ulysseB/telamon`