**DietCode**: Automatic Code Generation for Dynamic Tensor Programs

Bojian Zheng¹, ², ³ *, Ziheng Jiang⁴ *, Cody Yu², Haichen Shen²,
Josh Fromm⁵, Yizhi Liu², Yida Wang²,
Luis Ceze⁶,¹⁶, Tianqi Chen⁵, ⁷, Gennady Pekhimenko¹, ², ³

* Equal Contributions
Executive Summary

• Challenges posed by dynamic-shape workloads:
  • Vendor Libraries: Hard to be Engineered for Efficiency
  • Existing Auto-Schedulers: Long Compilation Time (days for a single operator)

• DietCode addresses the challenges with
  ① shape-generic search space
  ② micro-kernel-based cost model.

• Key Results:
  • Compilation Time: 5.88× saving vs. Ansor.
  • Performance: Up to 1.70× better vs. Ansor and 1.19× vs. the vendor library on modern GPUs.
Background: ML Framework Stack

Application

Image Classification\(^{[1]}\)

Machine Translation\(^{[2]}\)

Speech Recognition\(^{[3]}\)

Framework

TensorFlow\(^{[4]}\)

PyTorch\(^{[5]}\)

JAX\(^{[6]}\)

---

[3] https://github.com/NVIDIA/NeMo
Background: ML Framework Stack

Framework


Interpretation: Graph of Operators [7]

Background: Vendor Libraries

Operator (Matrix Multiply)

Vendor Libraries

- cuBLAS

- cuDNN

Built-in Kernel 1

Built-in Kernel 2

Dispatcher

Hardware

NVIDIA GPU

Background: Vendor Libraries

Operator (Matrix Multiply)

Invoke

Vendor Libraries

cuBLAS[1]

cuDNN[2]

Dispatcher

Built-in Kernel 2

Built-in Kernel 1

Hardware

NVIDIA GPU[3]

Background: Vendor Libraries

Operator
(Matrix Multiply)

Invoke

Vendor Libraries

cuBLAS\(^1\)

cuDNN\(^2\)

Dispatcher

Built-in Kernel 2

Built-in Kernel 1

Hardware

NVIDIA GPU\(^3\)

Background: Vendor Libraries

- **Challenges**
  - Performance of built-in kernels can be **suboptimal** on the given shapes or hardware [4, 5, 6, 7, 8, 9, ...].
  - **Huge** engineering efforts and expertise required to tune for specific use cases.

---

Background: Auto-Scheduler

- **Objective:**

  - **Operator**
  - **Shape Description**

  Auto-Scheduler

  **Automatically Generate**

  **Schedule** (i.e., Implementation)

  **Hardware**

Background: Auto-Scheduler

Background: Auto-Scheduler [1, 2, 3, 4]

Operator

Shape Description

E.g.,

Operator:
```
for (int i = 0; i < 50; ++i) {
    A[i] = ...;
}
```

Frontend

Auto-Scheduler

Hardware

---

Background: Auto-Scheduler \textsuperscript{[1, 2, 3, 4]}

Operator Shape Description

E.g.,

Search Candidate: tile size $t$

Loop Tiling Schedule:
\begin{verbatim}
for (int io = 0; io < |S|/t; ++io) {
  for (int ii = 0; ii < t; ++ii) {
    if (io*t + ii < 50) A[io*t + ii] = ...
  }
}\end{verbatim}

Background: Auto-Scheduler \[1, 2, 3, 4\]

Loop Tiling Schedule:
for (int io = 0; io < \text{50} / \mathbf{t}; ++io) {
  for (int ii = 0; ii < \mathbf{t} / \mathbf{t}; ++ii) {
    if (\omega \times t + ii < 50) A[\omega \times t + ii] = ...}
}

Search Candidate: tile size \(t \in [2, \infty]\)

Search Techniques:
1. Shape-Dependent Search Space
2. Complete Program Cost Model

\[1\] A. Adams et al. \textit{Halide Auto-Scheduler}. SIGGRAPH 2019
\[2\] N. Vasilache et al. \textit{Tensor Comprehensions}. TACO 2019
\[3\] L. Zheng et al. \textit{Ansor}. OSDI 2020
\[4\] S. Feng, B. Hou et al. \textit{TensorIR}. arXiv 2022
1. Shape-Dependent Search Space

Search Candidate: tile size $t \in [2, \infty)$

**Loop Tiling** Schedule:
```java
for (int io = 0; io < |S|; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if ($\omega \times t + ii < 50$) A[$\omega \times t + ii$] = ...
    }
}
```

Frontend

Auto-Scheduler

Hardware
1. Shape-Dependent Search Space

Loop Tiling Schedule:
```c
for (int io = 0; io < S[1]; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if (i*tx + ii < 50) A[i*tx + ii] = ...
    }
}
```
1. Shape-Dependent Search Space

Loop Tiling Schedule:

```
for (int io = 0; io < S[y]; ++io) {
    for (int ii = 0; ii < t; ++ii) {
        if ((io*t + ii) < 50) A[(io*t + ii)] = ...
    }
}
```
2. Complete Program Cost Model

Performance = Predictor(Complete Program Cost Model $P(\text{tile size } t))$

Loop Tiling Schedule:

```java
for (int io = 0; io < [50/t]; ++io) {
    for (int ii = 0; ii < $t$; ++ii) {
        if ($\omega*xt + ii < 50$) $A[\omega*xt + ii] = ...$
    }
}
```

Search Candidate: tile size $t$

Operator Shape Description

Shape-Dependent Search Space

Complete-Program Cost Model

Train

Frontend

Auto-Scheduler

Hardware

Program 1 (using tile size $t_1$)

Program 2

Measure

Train

Sample
Challenges from Dynamic-Shape Workloads

Diagram:
- Operator
- Shape Description
- Shape-Dependent Search Space
- Complete-Program Cost Model
- Program 1
- Program 2
- Measure
- Train
- Auto-Scheduler
- Hardware
Challenges from Dynamic-Shape Workloads

• **Cannot** efficiently handle **dynamic-shape** operators, common in
Challenges from Dynamic-Shape Workloads

- **Cannot** efficiently handle **dynamic-shape** operators, common in
  - Translation\(^1\)
  - Speech Recognition\(^2\)
  - Text Auto-Complete\(^4\)

whose input sentences/audios have dynamic lengths.

\(^1\) [https://translate.google.com/]
\(^2\) [https://github.com/NVIDIA/NeMo]
\(^3\) J. Devlin et al. BERT. NAACL-HTL 2019
\(^4\) A. Radford et al. GPT-2. 2019
Challenges from Dynamic-Shape Workloads

• **Cannot** efficiently handle **dynamic-shape** operators, due to
  • Humongous Search Space
  • Inaccurate Performance Prediction

• **DietCode**’s Key Ideas:
  • Shape-Generic Search Space
  • Micro-Kernel-based Cost Model
Challenge #1. Humongous Search Space

• **Hard** to share search space between operators of different shapes.
Challenge #1. Humongous Search Space

- **Hard** to share search space between operators of different shapes.
  - \( \cap \) search space: Tiny
  - \( \cup \) search space: Humongous
Challenge #1. Humongous Search Space

- **Hard** to share search space between operators of different shapes.
  - $\cap$ search space: Tiny
  - $\cup$ search space: Humongous
    $\Rightarrow$ Huge Compilation Time
    *(days* for a single operator)*
Key Idea #1. Shape-Generic Search Space

- Composed of **micro-kernels**, each
  - Does a tile of the entire compute.
  - Sampled uniformly from maximum shapes and constrained by hardware parameters.
  - Can be ported to **all** shapes of the same operator.
Challenge #2. Inaccurate Performance Prediction

• Cost model trained on one shape can be inaccurate on others.
  • E.g., Performance of $Y = XW^T$
    $X$: $[16 \times T, 768]$, $W$: $[2304, 768]$ w.r.t. $T$ on a NVIDIA Tesla T4 GPU$^1$, all sharing the same micro-kernel.

Challenge #2. Inaccurate Performance Prediction

- Cost model trained on one shape can be **inaccurate** on others.
  - E.g., Performance of $Y = XW^T$

  

Predictions are **inaccurate** on other shapes.
Key Idea #2. Micro-Kernel-based Cost Model

• Key Observation: Performance scales proportionally with hardware core occupancy.

\[ f_{\text{MicroKernel}} \cdot f_{\text{Penalty}} \]
Key Idea #2. Micro-Kernel-based Cost Model

• Key Observation: Performance scales proportionally with hardware core occupancy.

\[ f_{\text{MicroKernel}} \cdot f_{\text{Penalty}} \]

Trainable function for peak prediction

![Graph showing compute throughput vs. shape dimension T]

Micro-Kernel-based Cost Model
Key Idea #2. Micro-Kernel-based Cost Model

- Key Observation: Performance scales proportionally with hardware core occupancy.

\[ f_{\text{MicroKernel}} \cdot f_{\text{Penalty}} \]

Analytical linear function of the core occupancy.
Key Idea #2. Micro-Kernel-based Cost Model

- Key Observation: Performance scales proportionally with hardware core occupancy.

\[ f_{\text{MicroKernel}} \cdot f_{\text{Penalty}} \]

- More Accurate Predictions
**DietCode** System Overview

How is **DietCode** performing on real workloads?

- Operator
- Shape Description 1
- Shape-Generic Search Space
- Micro-Kernel-based Cost Model
- Micro-Kernel 1
- Micro-Kernel 2
- Measure
- Hardware
- Frontend
Evaluation

Hardware

NVIDIA Tesla T4 GPU\[1\]

NVIDIA RTX 3090 GPU\[2\]

Evaluation

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>TVM[3] v0.8.dev0</td>
<td>CUDA v11.3</td>
</tr>
</tbody>
</table>

[4] https://docs.nvidia.com/cuda/archive/11.3.0/
Evaluation

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA Tesla T4 GPU</td>
<td>TVM v0.8.dev0</td>
<td>Dynamic sequence lengths uniformly sampled within the range [1, 128]</td>
</tr>
<tr>
<td>NVIDIA RTX 3090 GPU</td>
<td>v11.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cuDNN v8.3</td>
<td></td>
</tr>
</tbody>
</table>

[1] T. Chen et al. TVM. OSDI 2018
[4] https://docs.nvidia.com/cuda/archive/11.3.0/
## Evaluation

### Hardware

| NVIDIA Tesla T4 GPU | NVIDIA RTX 3090 GPU |

### Software

| TVM v0.8.dev0 | v11.3 | cudNN v8.3 |

### Application

Dynamic sequence lengths uniformly sampled within the range $[1, 128]$.

### Baselines

- PyTorch with the Vendor Library's Auto-Scheduler Ansor
- NVIDIA Tesla T4 GPU
- NVIDIA RTX 3090 GPU

---

[4] https://docs.nvidia.com/cuda/archive/11.3.0/
**Evaluation**

<table>
<thead>
<tr>
<th>Hardware</th>
<th>NVIDIA Tesla T4 GPU[^1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Google BERT[^6]</td>
</tr>
</tbody>
</table>

[^3]: T. Chen et al. *TVM*. OSDI 2018  
[^4]: https://docs.nvidia.com/cuda/archive/11.3.0/  
[^6]: J. Devlin et al. *BERT*. NAACL-HTL 2019  
Compilation Time vs. Ansor

Better DietCode

Time (Hours)

0 1 2 3 4 5 6 7 8 9 10 11 12

Ansor DietCode

1,000 × 0.108 ×
Compilation Time

• Ansor: Time estimated to be more than **1 week** for only key operators.

![Graph showing compilation time]

- Better

\[16\times\text{if all sequence lengths are compiled (rather than the 8 sampled ones)}\]
Compilation Time vs. Ansor

- **Ansor**: Time estimated to be more than **1 week** for only key operators.

- **DietCode** reduces the compilation time by **5.88×** vs. Ansor
Compilation Time vs. Ansor

• Ansor: Time estimated to be more than 1 week for only key operators.

• *DietCode* reduces the compilation time by $5.88 \times$ vs. Ansor, as it only needs to compile once for all shapes.
  • $16 \times$ increase in compilation time
Compilation Time vs. Ansor

• Ansor: Time estimated to be more than 1 week for only key operators.

• *DietCode* reduces the compilation time by $5.88 \times$ vs. Ansor, as it only needs to compile once for all shapes.
  - 16× increase in compilation time
Latency vs. Vendor/Ansor

[Bar chart showing latency comparisons between Vendor, Ansor, and DietCode across different sequence lengths of BERT.]
Latency vs. Vendor/Ansor

Uniformly sampled and includes composite and prime numbers.
Latency vs. Vendor/Ansor

√ Up to $1.70 \times / 1.19 \times$ better than Ansor/Vendor.
Latency vs. Vendor/Ansor

√ Up to $1.70 \times / 1.19 \times$ better than Ansor/Vendor.
√ $1.30 \times / 1.05 \times$ on average.
Latency vs. Vendor/Ansor

Contributed by √

√ Up to 1.7 × better than Ansor/Vendor.

√ 19 × better than Ansor/Vendor. on average.
Latency vs. Vendor/Ansor

Better

Contributed by Shape

Sequence Length of BERT

√ Up to 1.70×/1.19× better than Ansor/Vendor.
√ 1.30×/1.05× on average.
Future Directions

• We are working on upstreaming *DietCode* to the TVM main branch: https://github.com/apache/tvm-rfcs/pull/72, together with improvement of the tuning algorithms.

  Many thanks to the community!

• Evaluations on more hardware platforms (CPUs and NVIDIA GPUs using tensor core operations)

Future Directions

• We are working on upstreaming DietCode to the TVM main branch: https://github.com/apache/tvm-rfcs/pull/72, together with improvement of the tuning algorithms.

Many thanks to the community!

• Evaluations on more hardware platforms (CPUs and NVIDIA GPUs using tensor core operations) and workloads.

[2] https://github.com/NVIDIA/NeMo

Speech Recognition

Object Detection
Conclusion

• Challenges posed by dynamic-shape workloads:
  • Vendor Libraries: Hard to be Engineered for Efficiency
  • Existing Auto-Schedulers: Long Compilation Time

• DietCode addresses the challenges with
  ① shape-generic search space
  ② micro-kernel-based cost model.

• Key Results:
  • Compilation Time: $5.88 \times$ saving vs. Ansor.
  • Performance: Up to $1.70 \times$ better vs. Ansor and $1.19 \times$ vs. the vendor library on modern GPUs.
**DietCode**: Automatic Code Generation for Dynamic Tensor Programs

Bojian Zheng\(^1, 2, 3\), Ziheng Jiang\(^4\), Cody Yu\(^2\), Haichen Shen\(^2\), Josh Fromm\(^5\), Yizhi Liu\(^2\), Yida Wang\(^2\), Luis Ceze\(^5, 6\), Tianqi Chen\(^5, 7\), Gennady Pekhimenko\(^1, 2, 3\)

\(^*\) Equal Contributions
Optimization Strategy

• **Objective**: End-to-End Latency

• All workloads have “optimization priority” as
  \[ \text{FLOPs} \times \text{weight (user-defined, 1 by default)} \]

• Consequently, workloads with higher FLOPs will be given more
  attention compared with smaller ones.
NVIDIA RTX 3090[1] Preliminary Results

Micro-Kernel-based Cost Model

![Graph showing comparison between real measurements, prior cost model, and micro-kernel-based cost model.]

Performance

Up to $1.52 \times / 1.26 \times$ better than Ansor/Vendor ($1.16 \times / 1.11 \times$ on average).

Conv2D

- NCHW is usually implemented using the Winograd algorithm, which is in essence batched matrix multiplies.

- NHWC:

Up to $1.11 \times / 2.01 \times$ better than Ansor/Vendor ($1.02 \times / 1.80 \times$ on average).
vs. Nimble [1]

• Focuses on the **runtime system** for dynamic-shape workloads, with one section (i.e., Section 3.5) discussing about the code generation.

Nimble cannot cover all shapes efficiently.

Local Padding vs. Loop Partitioning

• Common:
  • Key Observation: Out-of-boundary checks in the compute stage are what negatively affect performance the most.
  • Performance difference is usually less than 5%.
Local Padding vs. Loop Partitioning

**Local Padding**
- Key Idea: Pads tensors by the size of the local workspace when loading from the off-chip device memory.
- (-) Redundant computations

**Loop Partitioning**
- Key Idea: Partition the regions that have predicates and regions that do not.
- (-) Cannot remove overheads in some pathological cases.
- (-) Cannot support compute intrinsics such as the tensor core operations.