DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption

Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, and Ju Min Lee, Yonsei University; Sunchul Jung and Taekyung Kim, CryptoLab. Inc.; Dongyoon Lee, Stony Brook University; Hanjun Kim, Yonsei University

https://www.usenix.org/conference/usenixsecurity24/presentation/cheon

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA
978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
DaCapo: Automatic Bootstrapping Management for Efficient Fully Homomorphic Encryption

Seonyoung Cheon  
Yonsei University  
Yongwoo Lee  
Yonsei University  
Dongkwan Kim  
Yonsei University  
Ju Min Lee  
Yonsei University  
Sunchul Jung  
CryptoLab. Inc  
Taekyung Kim  
CryptoLab. Inc  
Dongyoon Lee  
Stony Brook University  
Hanjun Kim  
Yonsei University

Abstract

By supporting computation on encrypted data, fully homomorphic encryption (FHE) offers the potential for privacy-preserving computation offloading. However, its applicability is constrained to small programs because each FHE multiplication increases the scale of a ciphertext with a limited scale capacity. By resetting the accumulated scale, bootstrapping enables a longer FHE multiplication chain. Nonetheless, manual bootstrapping placement poses a significant programming burden to avoid scale overflow from insufficient bootstrapping or the substantial computational overhead of unnecessary bootstrapping. Additionally, the bootstrapping placement affects costs of FHE operations due to changes in scale management, further complicating the overall management process.

This work proposes DaCapo, the first automatic bootstrapping management compiler. Aiming to reduce bootstrapping counts, DaCapo analyzes live-out ciphertexts at each program point and identifies candidate points for inserting bootstrapping operations. DaCapo estimates the FHE operation latencies under different scale management scenarios for each bootstrapping placement plan at each candidate point, and decides the bootstrapping placement plan with minimal latency. This work evaluates DaCapo with deep learning models that existing FHE compilers cannot compile due to a lack of bootstrapping support. The evaluation achieves 1.21× speedup on average compared to manually implemented FHE programs.

1 Introduction

Fully homomorphic encryption [22] (FHE) has emerged as a promising solution for privacy-preserving computation in untrusted environments, like financial and healthcare applications on a third-party cloud. FHE allows computations on encrypted data, and its decrypted results are identical to those obtained from unencrypted computations. Since Gentry’s pioneering work on FHE using lattice-based cryptography [22], many FHE schemes [8,11,12,14,20], and supporting libraries [1,2,17,21,25,26,46,48,54] have been proposed. Among the FHE schemes, RNS-CKKS [11] is suitable for privacy-preserving machine learning models as it offers optimized fixed-point arithmetic and SIMD support.

In RNS-CKKS, a ciphertext embeds a scaled integer with the maximum scale capacity, called coefficient modulus Q. Each FHE multiplication increases its accumulated scale, like the multiplication of two scaled numbers. When the accumulated scale exceeds the coefficient modulus Q, scale overflow occurs, leading to an unrecoverable result. Thus, programmers should carefully design an RNS-CKKS program with a limited number of multiplications, or insert bootstrapping [10] operations that reset the accumulated scale of a ciphertext, enabling a longer multiplication chain. Although existing RNS-CKKS compilers [18,39,41] can automatically manage ciphertext scales reflecting constraints of RNS-CKKS operations, their applicability remains limited to small RNS-CKKS programs due to lack of bootstrapping support. Only manual bootstrapping placement [36,38] has been proposed for a long FHE program such as ResNet, but the manual placement imposes a significant burden on programmers and easily leads to inefficient performance.

Optimizing a long FHE program with manual bootstrapping placement is challenging for three reasons. First, programmers should trace the growth of the accumulated scale of each ciphertext, and insert the bootstrapping operation before scale overflow occurs. Second, a bootstrapping operation has significant latency, which is the most expensive among RNS-CKKS operations. While avoiding scale overflow from insufficient bootstrapping, programmers should reduce the number of bootstrapping operations, not to unnecessarily increase bootstrapping overheads. Third, bootstrapping placement affects scale management. Since bootstrapping resets the accumulated scales of ciphertexts, and RNS-CKKS operations have different latencies depending on their accumulated scale, the bootstrapping insertion points change the latencies of the other RNS-CKKS operations. Therefore, to optimize bootstrapping placement, the number of bootstrapping operations and their performance impact on the other RNS-CKKS operations should be considered together.
This work proposes DACapo, the first automatic bootstrapping placement compiler that heuristically places bootstrapping operations to minimize the overall latency in an FHE program. To reduce the number of bootstrapping operations, DACapo analyzes live-out ciphertexts, which are used by later instruction at each program point, and classifies program points with a minimal number of live-outs as a candidate set of bootstrapping insertion points. DACapo identifies a subset of the live-out ciphertexts that remain unused for a certain period without requiring bootstrapping, and categorizes them as bypass edges. DACapo excludes the bypass edges from the live-outs and thus achieves more precise required bootstrapping counts. To reflect the latency of the other RNS-CKKS operations, such as multiplications and rotations, affected by scale management and bootstrapping insertion points, DACapo estimates their latency under different scale management scenarios for each bootstrapping point and finds a bootstrapping placement plan with minimal latency among the candidate set.

This work implements DACapo compiler on top of MLIR [33] with a GPU-accelerated RNS-CKKS library called HEaan [25]. This work evaluates DACapo with six deep learning models such as ResNet20, ResNet-44, AlexNet, VGG-16, SqueezeNet, and MobileNet. This work implements the deep learning models with maxpooling and ReLU like the original models and also implements the same models with average pooling and SiLU that require a shorter multiplication chain. The evaluation results show that DACapo achieves 1.21× speedup on average for diverse deep learning models, compared to the manually implemented FHE programs.

Followings are the contributions of this work.

- The first automatic bootstrapping management compiler, DACapo, for fully homomorphic encryption;
- A new liveness-aware bootstrapping placement analysis that generates an insertion point candidate set with minimal live-out counts;
- A new cost-aware bootstrapping placement that reflects different FHE computation costs under different scale management affected by bootstrapping placement plan.

### 2 Background

This section describes RNS-CKKS [11] and scale management for RNS-CKKS programs. In contrast to other FHE schemes [8,12,14,20,22], RNS-CKKS provides better support for fixed-point arithmetic and SIMD parallelism by leveraging approximation (i.e., encrypted computation results could be slightly different from unencrypted ones). These properties make RNS-CKKS well-suited for machine learning applications that can accommodate a certain level of approximation.

#### 2.1 RNS-CKKS Encoding and Encryption

RNS-CKKS introduces two encoding and encryption parameters: polynomial modulus degree $N$ and coefficient modulus $Q$. RNS-CKKS encodes a vector containing real numbers into a cyclotomic polynomial with a coefficient modulus $Q$. RNS-CKKS encodes a polynomial into a ciphertext, which is a pair of polynomials $(-a \cdot s + \mu + e, a)$ with random noises $a$ and $e$ and the secret key $s$. The security of RNS-CKKS relies on the hardness of Ring Learning with Errors (RLWE) [42].

An RNS-CKKS ciphertext has a scale and a level property, defined as follows. RNS-CKKS deals with a polynomial with integer coefficients and encodes a real number as an integer over a scale. For example, the number $x = 1.1$ is encoded as an integer $v = 11$ with the scale $m = 10^1$. The multiplication of two scaled numbers increases the result’s scale; i.e., the scale of a ciphertext gradually accumulates on a multiplication. For instance, $y = x \cdot x$ leads to an integer $v = 121$ with the scale $m = 10^2$. The scale is limited by the maximum scale capacity (defined by coefficient modulus $Q$), which is described as a product of the small fixed-size scaling factors $S_f$; i.e., $Q \approx S_f^L$, where $L$ is the initial level. The level $l$ denotes the number of scaling factors $S_f$ available in the ciphertext.

Figure 1a illustrates an RNS-CKKS ciphertext with scale and level. The ciphertext has the maximum scale capacity $S_f^L$.
Table 1: FHE Parameters and their descriptions. The rightmost column represents the values used in actual evaluation.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>In our evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>Polynomial modulus degree</td>
<td>$2^{127}$</td>
</tr>
<tr>
<td>$Q$</td>
<td>Coefficient modulus</td>
<td>$2^{1479}$</td>
</tr>
<tr>
<td>$S_f$</td>
<td>Rescaling factor</td>
<td>$2^{51}$</td>
</tr>
<tr>
<td>$L_{max}$</td>
<td>Maximum level of ciphertext</td>
<td>29</td>
</tr>
<tr>
<td>$L$</td>
<td>Level after bootstrapping</td>
<td>16</td>
</tr>
</tbody>
</table>

where the initial level $L = 5$ (depicted with five boxes). Starting from the initial level $L$, the level $l$ decreases on modswitch and rescale operations and becomes reset to the initial level $L$ on bootstrap operation (which will be explained later). Two dotted gray boxes on the left represent two scale factors consumed by modswitch or rescale operations, rendering the current level $l = 3$. The accumulated scale denotes the total scale accumulated (by multiplications) so far, and it should be lower than the scale capacity $S_f$. Otherwise, the scale overflow occurs, leading to an irrecoverable result. The scale $m$ excludes the consumed scale factors and it should be larger than the predefined minimum scale, called *waterline* $S_w$, to control the noises introduced by RNS-CKKS operations.

Table 1 shows description and values of FHE parameters used in the evaluation, including a polynomial modulus degree ($N$) of $2^{127}$, and a ciphertext can contain 65536 elements. The evaluation uses a coefficient modulus composed of primes with the same scaling factor ($R$) of 51 bits, and its value is equal to $2^{1479}$. The bootstrapping implementation provided by library consumes 13 levels. In other words, when bootstrapping is performed using parameters with a maximum level of $L_{max}$, the resulting ciphertext will have 16 levels ($L$).

### 2.2 RNS-CKKS Operations

#### 2.2.1 Arithmetic and Rotation Operations

RNS-CKKS supports basic arithmetic and rotation operations, which can be used to implement application algorithms.

**Addition** (Figure 1b) and **Multiplication** (Figure 1c): RNS-CKKS supports two vectorized arithmetic operations: addition and multiplication. These operations add/multiply two ciphertexts (CAdd and CMul, respectively) or one plaintext and one ciphertext (PAdd and PMul). The addition operation requires the levels and scales of two operands to be the same, and produces a ciphertext of the same level $l$ and scale $m$. On the other hand, multiplication requires only the levels of two operands to be the same. The scale of the resulting ciphertext accumulates to the product of the scales of the operands: i.e., $m = m_1 \cdot m_2$. Table 2 presents the runtime latency of (some) RNS-CKKS operations at different levels measured in the GPU-accelerated HEAAN library [25]. Multiplications are slower than additions. Arithmetic operations at a high level are more expensive than ones at a low level.

#### 2.2.2 Scale Management Operations

RNS-CKKS further requires programmers to use the following scale management operations to meet the RNS-CKKS constraints (which are irrelevant to application algorithms).

**Upscale**: The upscale operation is a syntactic sugar (not a separate RNS-CKKS) operation that multiplies an operand ciphertext at scale $m_1$ with a multiplicative identity (i.e., 1) at scale $m_2$, increasing the scale of the operand to $m = m_1 \cdot m_2$ (by an arbitrary amount). The operation can be used to match the scale of an addition operand if needed.

**Modswitch** (Figure 1d): The modswitch operation consumes one scale factor $S_f$ without affecting the scale of a ciphertext, which in effect reduces the level by one. The operation can be used to adjust the level of an addition or multiplication operand if they do not match.

**Rescale** (Figure 1e): Multiplications accumulate the scales of ciphertexts. To avoid scale overflow (where the accumulated scale exceeds the maximum scale capacity $S_f$), RNS-CKKS supports the rescale operation that divides coefficient modulus $Q$ by the scale factor $S_f$. In effect, the result’s scale decreases from $m$ to $m/S_f$ (unlike modswitch) and its level decreases from $l$ to $l - 1$. Note that the rescale operation does not change the accumulated scale. To accommodate noises in RNS-CKKS operations, the rescale operation should be used only if the scale after rescaling still remains higher than the minimum scale waterline: i.e., $m/S_f \geq S_w$.

**Bootstrap** (Figure 1f): Even with rescaling, for a program with a large number of multiplications, all available scale budgets will be eventually consumed and the accumulative scale will reach the cap. Further operations become infeasible due to a scale overflow. As a solution, RNS-CKKS supports the bootstrap operation that resets the scale and level of a ciphertext. After bootstrapping, the accumulated scale (and the current scale at level $L$) decreases to scale factor $S_f$. Table 2 shows the runtime latency of bootstrap from a different

### Table 2: Latency of FHE operations for different levels (µs)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>PAdd</td>
<td>24</td>
</tr>
<tr>
<td>CAdd</td>
<td>33</td>
</tr>
<tr>
<td>PMult</td>
<td>214</td>
</tr>
<tr>
<td>CMult</td>
<td>1218</td>
</tr>
<tr>
<td>Rot</td>
<td>890</td>
</tr>
<tr>
<td><strong>Boot</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Rotation**: For vectorized arithmetic operations, RNS-CKKS additionally supports the rotate operation that performs a circular shift on a message vector by a given offset: e.g., rotating $(a_1,a_2,\ldots,a_{10})$ by offset 2 leads to $(a_3,a_4,\ldots,a_{10},a_1,a_2)$. The rotate operation is a relatively expensive operation, next to a CMul operation.
level to the maximum level \(L(=16)\). The bootstrap latency solely depends on the target level and does not depend on the level gap. As reported, bootstrap is the most expensive operation in RNS-CKKS: e.g., 3774 times slower than CAdd, and 112 times slower than CMul at level 16.

3 Motivation

This section discusses the limitations of existing RNS-CKKS compilers and the difficulties in manual bootstrapping, motivating this work: new compiler support for bootstrapping.

3.1 Prior RNS-CKKS Compilers

Existing RNS-CKKS compilers [18, 41] aim to support automatic scale management that takes as input a program that consists of only arithmetic and rotation operations (implementing application computation logic), and inserts scale management operations upscale, modswitch, and rescale (but not bootstrap) to produce a new program that satisfies the RNS-CKKS constraints for security and correctness.

Although EVA [18] and Hecate [41] successfully automate many parts of scale management in RNS-CKKS, they both lack support for automatic bootstrapping management. As a result, they have been applied to relatively small deep-learning applications such as LeNet-5 [34], leaving more complex applications with a deep multiplicative depth out of reach.

3.2 Manual Bootstrapping Placement

Unfortunately, the state-of-the-art that supports large RNS-CKKS applications is manually adding bootstrap operations, which is error-prone and time-consuming. The manual placement is challenging for the following three reasons.

First, to avoid a scale overflow problem and ensure correctness, users should place a bootstrap operation before the accumulated scale (after the last bootstrapping) exceeds the maximum scale capacity \(S/L\). Manually tracking the growth of the accumulated scales of a large number of ciphertexts is indeed very difficult because they depend on not only the existing arithmetic and rotation operations, but also newly added upscale or rescale operations for scale management.

Second, besides correctness, conducting performance-aware bootstrap operation placement manually (leading to lower latency) is even harder. As discussed earlier with Table 2, the bootstrap operation is much more expensive than any other operations. Thus, in most cases, it is a good idea to minimize the number of bootstrap operations (later we show that in some cases, more bootstrapping may count-intuitively lead to better performance).

Third, the bootstrapping placement affects scale management and its performance. The bootstrap operation changes the scale and level of a ciphertext (like other scale management operations), implying that the location of a bootstrap operation has a great influence on the level of the following ciphertexts, impacting the latencies of subsequent RNS-CKKS operations. For example, what should we do if we have multiple solutions with the same minimum number of bootstrap operations? Reasoning about the cascading impacts of multiple bootstrapping options is not a trivial task.

4 Key Observations and Insights

The RNS-CKKS compilers (e.g., EVA [18] and Hecate [41]) can keep track of the scale and level of ciphertexts while placing scale management operations. A naive compiler-based bootstrapping solution would be to extend them to place a bootstrap operation right before the RNS-CKKS operation that exceeds the maximum scale capacity (to avoid a scale overflow). However, we made two novel observations that such a naive extension (based on an accumulated scale analysis) would lead to inefficient performance.

Observation 1: A simple accumulated scale analysis may introduce (unnecessarily) many bootstrap operations, incurring significant performance overhead. Consider a simple deep learning example in Figure 2a that consists of three convolution layers (Conv1, Conv2, and Conv3), two activation functions (Act), and one addition operation (Add). Figure 2a (right) shows ciphertexts (white box) and RNS-CKKS arithmetic and rotation operations (arrows). A simple convolution layer includes three pairs (channels) of one Rot and two PMul operations, aggregated by CAdd at the end. A simple activation function (Act) uses a single CMul operation.

Suppose that the scale of the input ciphertext \(x\) is \(2^{40}\), and its waterline, rescaling factor and the maximum scale capacity are \(2^{40}, 2^{60}\) and \(2^{300}\) (the initial level is 5). Figure 2b illustrates how the accumulated scale (left y-axis) and the level (right y-axis) of a ciphertext change in this example program (while scale management operations are omitted for simplicity). The x-axis reports the time latency, calculated using scaled profiled costs based on Table 2. Figure 2b shows the execution model of a naive bootstrapping approach in which a bootstrap operation is added right before the RNS-CKKS operation that exceeds the maximum scale capacity. During the third Conv3, the accumulated scale would become larger than the maximum scale capacity on each of the three second PMul operations. To prevent scale overflow, the naive compiler would add three bootstrap operations to reset the scales/levels of three ciphertexts \(M_s\).

In Table 2, the bootstrap operation is much more expensive than any other operation. It is thus often profitable to use fewer bootstrap operations (if possible). The next two alternatives (analyzed by our compiler DACapo) in Figure 2c and Figure 2d demonstrate that it is indeed possible to reduce the latency by placing fewer bootstrap operations at better locations. They are based on the following two key insights.

Insight 1A: A ciphertext liveness analysis can help reduce the number of bootstrap operations. Our first in-
Figure 2: Motivating example and its execution models with different bootstrapping placement policies. The example assumes $2^{40}$ waterline, $2^{60}$ scaling factor, and $2^{300}$ scale capacity (the maximum level is 5).

sight is that the naïve approach added three bootstrapping operations because a ciphertext dataflow diverges and three ciphertexts $M$s are later “used” so they all have to be bootstrapped for correctness. More formally in a compiler term, there were three live-out variables $M$s in a dataflow graph that must be bootstrapped. If the optimization goal is to minimize the number of bootstrapping operations, then it would be beneficial to consider an alternative (and earlier) program point with fewer live-outs, or equivalently, a program point before a dataflow diverges or after it converges.
Based on this insight, DACapo performs liveness-based bootstrapping management. More specifically, DACapo counts the live-out ciphertexts at each program point, and regards minimum live-out points as candidate bootstrapping points. For instance, Figure 2c inserts the first bootstrap operation at the earlier program point with one live-out ciphertext Au (after a dataflow converges from Conv1(x) and before it diverges to Conv2(a)), even though the accumulated scale is still much lower than the maximum capacity.

Insight 1B: A long-lived unused ciphertext that bypasses other RNS-CKKS operations is unlikely to require bootstrapping, so it is often profitable to exclude the ciphertext from the live-out counts. In a ciphertext dataflow graph (e.g., Figure 2a), a bypass ciphertext implies the ciphertext that was defined but not used for a while. For instance, the ciphertext A is defined at the end of the Conv1(x); not used by (bypassing) Conv2(a), Act(b), and Conv3(c); and finally used by Add(a,d). The long-lived ciphertext is represented as a long edge from A in Conv1(x) to a in Add(a,d). Such a long-lived unused (bypass) ciphertext is unlikely to require bootstrap. A strict inclusion of those ciphertexts in liveness analysis may unnecessarily increase live-out counts at some program points, and remove bootstrapping placement candidates, thus losing further optimization opportunities. Then, DACapo would not consider those program points as bootstrapping candidates.

To address them, DACapo excludes the long-lived (unused-for-a-while) ciphertexts from the live-out counts, achieving better bootstrapping management. For example, consider the ciphertext T in Act(b). At that program point, the actual live-out count is two including the short edge from A in Conv1(x) to b in Act(b), and the bypass edge from A in Conv1(x) to a in Add(a,d). The plain liveness-based DACapo would not consider the program as bootstrapping candidates and produces the bootstrapping result in Figure 2c. On the other hand, once the bypass edge is excluded, the new live-out count will be one. DACapo could find a better plan in Figure 2d with only one bootstrap operation.

Observation 2: Different bootstrapping placements (locations and numbers) may lead to widely different end-to-end latency because bootstrapping may allow other RNS-CKKS operations to be performed at lower levels. As the bootstrap operation resets the scale and level of a ciphertext, adding one bootstrap in effect can be viewed as partitioning a given program into two smaller sub-programs. Interestingly, if one partition has fewer multiplications (less multiplicative depth), the arithmetic and rotation operations in that partition may be executed at lower levels, resulting in less end-to-end latency. Note that RNS-CKKS operations run faster at lower levels (See Table 2). For the same reason, we also make a rather counter-intuitive observation that fewer bootstrap operations do not always lead to lower latency. More bootstrap operations imply more partitioning of a given program. Each partition would have fewer operations, which can be performed at lower levels, resulting in less latency. The performance improvement with lower-level operations may be greater than the performance overhead of additional bootstrap operations.

Insight 2: A cost-aware bootstrapping analysis is needed for performance. Learned from the above observation, DACapo leverages the aforementioned liveness analysis to collect different bootstrapping candidates, and compares static cost (latency) estimates among them to select the lowest. For example, compare two different bootstrapping locations between Figure 2d and Figure 2e for the same number (one) of bootstrap operations. Figure 2e performs bootstrapping before CMul in Act(b), allowing many arithmetic and rotation operations in the first partition (the ones in Conv1(x) and Conv2(a)) to be performed at level 4 or less after early modswitch (shown in a red arrow). On the other hand, Figure 2d performs many of these operations at a higher level 5, leading to higher end-to-end latency.

5 Overview

This work proposes DACapo, the first FHE compiler that automatically places bootstrap operations. Given an input program written in plain operations, DACapo transforms the plain operations into RNS-CKKS operations with scale management operations including bootstrap, and generates LLVM IR codes that invoke GPU-accelerated HEaaN library [25] for RNS-CKKS operations. To support cost-aware bootstrapping placement, DACapo consists of two components: candidate selector and bootstrapping planner. Figure 3 shows DACapo’s two-step workflow on the left along with an example on the right side, which will be explained later.

Candidate selector identifies candidate points in the program where bootstrapping can be inserted. The candidate selection consists of three steps: liveness analysis, bypass edge analysis, and candidate filtering. The liveness analysis step analyzes live-out ciphertexts at each program point, and the bypass edge analysis step analyzes long-lived unused edges like the edge from A in Conv1(x) to a in Add(a,d) in Figure 2a. Combined, the candidate selector can analyze the number of valid live-out ciphertexts that may need bootstrapping at each program point. The candidate filtering step finalizes the candidate points. If the candidate points are too few to generate a correct FHE program due to scale overflow, the candidate filtering step increases the threshold number of live-outs until the scale overflow is resolved.

DACapo’s bootstrapping candidate program points selection implies that it adds bootstrap operations synchronously to all the live-out ciphertexts except for bypass ones at those program points. If a compiler bootstraps only a subset of live-out ciphertexts that needs to be refreshed (i.e., not all at once), chances are that the refreshed accumulated scale may be wasted because of the level match constraints of RNS-CKKS arithmetic operations. When the levels of two addition or multiplication operands are different, a scale management
Figure 3: Overview design of DACAPO with the example in Figure 2a. Candidate selector analyzes live-outs and bypass edges, and identifies candidate points where bootstrapping can be inserted. Bootstrapping planner estimates computation costs at each candidate point, and determines the bootstrapping insertion points with minimum overall latency among the candidate points. The bold numbers at the end of each candidate indicate the minimum computation cost. Gray arrows and numbers are the best estimated costs, and finalizes the bootstrapping placement plan. The star indicates the selected plan.

6 Bootstrapping Candidate Selector

This section introduces the bootstrapping candidate selector with an algorithm that analyzes appropriate candidates for bootstrapping placement. The candidate selector analyzes the live-out of each program point via liveness analysis, and identifies the required number of bootstrap for each program point by excluding bypass edges (§6.1). Then, the candidate selector finalizes the candidates from the candidate sets grouped by the number of bootstrap (§6.2).

6.1 Liveness and Bypass Edge Analysis

Liveness and bypass edge analysis find all the ciphertexts that should be bootstrapped if DACAPO decides to insert bootstrap at a program point. With the liveness analysis, DACAPO can find all the live-out ciphertexts that will be used later at a certain program point (LiveVariable in Algo-
Algorithm 1: Bootstrap Candidate Selection

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Input:</strong> Func: Function of an HE application</td>
</tr>
<tr>
<td>2</td>
<td><strong>Output:</strong> Set: Set of candidates</td>
</tr>
<tr>
<td>3</td>
<td><strong>Function</strong> CandidateSelector (Func) :</td>
</tr>
<tr>
<td>4</td>
<td>foreach target ∈ Func do</td>
</tr>
<tr>
<td>5</td>
<td>BootTarget ← LiveVariable(target) − BypassEdges</td>
</tr>
<tr>
<td>6</td>
<td>CandidatesSet[BootTarget.size] += target</td>
</tr>
<tr>
<td>7</td>
<td>Managed ← ScaleManagement(Func, target)</td>
</tr>
<tr>
<td>8</td>
<td>foreach op ∈ Managed after target do</td>
</tr>
<tr>
<td>9</td>
<td>if op.AccumulatedScale &gt; Sth and EndOperation(target) after op then</td>
</tr>
<tr>
<td>10</td>
<td>BypassEdges += {target}</td>
</tr>
<tr>
<td>11</td>
<td>if op.AccumulatedScale &gt; Sth then</td>
</tr>
<tr>
<td>12</td>
<td>target.Coverage ← op</td>
</tr>
<tr>
<td>13</td>
<td>break</td>
</tr>
<tr>
<td>14</td>
<td>end</td>
</tr>
<tr>
<td>15</td>
<td>EndOperation(target)</td>
</tr>
<tr>
<td>16</td>
<td>Candidates ← CandidatesSet[threshold]</td>
</tr>
<tr>
<td>17</td>
<td>while not success do</td>
</tr>
<tr>
<td>18</td>
<td>success ← true</td>
</tr>
<tr>
<td>19</td>
<td>Managed ← ScaleManagement(Func, Candidates)</td>
</tr>
<tr>
<td>20</td>
<td>foreach op ∈ Managed do</td>
</tr>
<tr>
<td>21</td>
<td>if op.AccumulatedScale &gt; Sth then</td>
</tr>
<tr>
<td>22</td>
<td>threshold += 1</td>
</tr>
<tr>
<td>23</td>
<td>Candidates += CandidatesSet[threshold]</td>
</tr>
<tr>
<td>24</td>
<td>success ← false</td>
</tr>
<tr>
<td>25</td>
<td>end</td>
</tr>
<tr>
<td>26</td>
<td>return Candidates</td>
</tr>
</tbody>
</table>

The result of the operation at target as a bypass ciphertext if the result is live after an operation whose accumulated scale exceeds the threshold Sth (line 9-11). Furthermore, the algorithm checks the accumulated scale of operations in managed and identifies the operations that can be executed without additional bootstrap (line 12-14). The remaining algorithm will be explained in §6.2.

Figure 3 illustrates how the liveness and bypass edge analysis analyze the example program. First, the liveness analysis gives the number of live variables for each program point. For example, there are 3 live variables (3 Rs) after Rot in Conv1. Note that this analysis counts the live-out variables, not the uses of the variables, so there is 1 live variable after Conv1. Then, the bypass edge analysis detects and removes a bypass edge from the live-out variable sets. For example, Aa in Conv1 is live-out until Add, but the accumulated scale of Add in Conv2 before Add is 280 (see Figure 2c) that exceeds the threshold Sth (e.g., 180). Therefore, the analysis classifies Aa as a bypass edge and then removes Aa from the live variable set of other operations. Then, the number of live variables after Ab (Conv2) is reduced from 2 (\{Aa, Ab\}) to 1 (\{Ab\}).

The Sth value, which is heuristically determined by users, impacts the performance of the generated code. A higher Sth makes Algorithm 1 detect fewer bypass edges, thus causing DACAPO to insert unnecessary bootstrapping operations for long-lived unused ciphertext. On the other hand, a lower Sth makes Algorithm 1 mark more edges with a shorter life-span as bypass edges, so DACAPO may exclude edges requiring bootstrapping from the candidates. Thus, assigning an appropriate Sth value is crucial for optimizing the bootstrapping placement. This work uses Sth as half of the maximum scale capacity for the evaluation.

### 6.2 Candidates Filtering

DACAPO generates a candidate set by filtering the bootstrapping targets based on the number of bootstrap. If the scale management cannot generate a valid program for a given candidate set, meaning that the accumulated scale of a ciphertext exceeds its scale capacity, the generated candidate set needs to be expanded. On the other hand, to minimize the number of bootstrap, the algorithm should find the minimal threshold of filtering. Hence, to find the minimal filtering threshold, the candidate filtering gradually increases the threshold from 1 to a value that can generate a valid scale management plan.

Line 17-29, describe the candidate filtering algorithm. The algorithm initializes the candidate set with targets requiring only one bootstrap (line 19). The algorithm executes scale management for the candidate set (line 22), and checks if the accumulated scale of a ciphertext exceeds the scale capacity. If so, the algorithm increases the threshold and expands the candidates (line 23-27). If the filtering is finished, the algorithm returns the finalized bootstrap candidate set (line 30).

Figure 3 illustrates how the candidate filtering works for the
This section describes the bootstrapping planner that finds valid live variables to expand the candidate set.

**Algorithm 2: Bootstrapping Planner Algorithm**

```
Algorithm 2: Bootstrapping Planner Algorithm

Input: Set: Candidates
Output: Set: Bootstrapping Targets
Function BootstrappingPlanner (Func, Candidates):
    Candidates.push (Func.ReturnOp)
    foreach to ∈ Candidates do
        foreach from ∈ Candidates do
            if to before from.Coverage then
                Managed ← ScaleManagement (Func, BestPlan[from] + to)
                cost ← EstimateCost (Managed, from, to)
                cost ← cost + MinCost[from]
                if cost < MinCost[to] then
                    MinCost[to] ← cost
                    BestPlan[to] ← BestPlan[from]
            BestPlan[to] += to
    return BestPlan[ReturnOp]
```

example. First, the candidate filtering algorithm only considers the bootstrapping candidate set \( \{C_1, C_2, C_3, C_4, C_5\} \) that has only one valid live variable \( \{A_a, A_b, A_T, A_d, A_e\} \), respectively. For the candidate set, the algorithm inserts `bootstrap` for each candidate and then performs scale management. If there is no operation whose accumulated scale exceeds the scale capacity, the candidate set is valid. If the candidate set is not valid, the algorithm increases the threshold number of valid live variables to expand the candidate set.

**7 Cost-aware Bootstrapping Planner**

This section describes the bootstrapping planner that finds bootstrapping insertion positions with minimal latency costs, for the given bootstrapping placement candidates from bootstrapping candidates selection (§6). Since `bootstrap` operation resets the scale and level of a ciphertext, depending on the bootstrapping placement, the scale management results vary, and thus the latency (level) of each ciphertext varies also. To optimize the bootstrapping placement plan reflecting the varying latency, the bootstrapping planner estimates the cost for each candidate and decides the placement plan with minimum overall cost. Here, this work employs a simple cost estimation that adds the profiled latency of each RNS-CKKS operation (e.g., Table 2).

Algorithm 2 illustrates the bootstrapping placement plan decision algorithm. For each `to` candidate, the algorithm analyzes the minimum estimated cost from the beginning of the program to the `to` program point. Iterating other `from` candidates before `to`, the algorithm executes scale management from `from` to `to`, estimates its latency cost, adds the estimated cost and the minimum cost at `from`, and finds the minimum cost by comparing the added result with the existing minimum estimated cost at `to`. Note that the scale management result is memorized while not reflected in the algorithm. Since the scale management of subprograms before and after `bootstrap` is decoupled, and the input scale and level of the succeeding subprogram only depend on `bootstrap`, the minimum cost can be computed by adding the current estimated cost and the minimum cost at `from`. The algorithm returns the bootstrapping plan at `ReturnOp` as the final placement plan.

Figure 3 illustrates how the bootstrapping planner algorithm works in the example. First, the planner executes scale management and estimates the cost of the partial program. For example, the planner estimates the cost of a partial program until `C_1` as 343. Then, to find a bootstrapping placement plan with the minimal estimated cost until the next candidate `C_2`, the planner considers two different cases. For the case that `bootstrap` is not inserted at `C_1`, the planner estimates the cost (935) in the same way as `C_1`. For the case that `bootstrap` is inserted at `C_1`, the planner reuses the previous estimation result for `C_1` (343), and adds the latency of `bootstrap` (27000) and the latency of the partial program from `C_1` to `C_2` (604), calculating the resulting cost (27947). The planner compares the estimated costs of two cases and selects the lower one (no `bootstrap`). The selected plan for `C_2` is reused for the other candidates such as `C_3`, `C_4`, `C_5`, and `Ret`.

**8 Evaluation**

The evaluation section first describes our experimental setup in §8.1 and demonstrates the overall performance improvement of DCAPO compared to manually implemented programs in §8.2. We also show the effectiveness of DCAPO by comparing the bootstrapping counts in §8.3, and highlight its practicality by analyzing the compile time in §8.4. Furthermore, we present the accuracy of performance estimation in §8.5. Lastly, we perform the latency sensitivity study with varying waterlines for each benchmark application in §8.6.

**8.1 Experimental Setup**

This paper evaluates multiple complex deep learning models with two different activation functions, including ResNet-20/44 [24], AlexNet [31], VGG16 [49], SqueezeNet [29], and MobileNet [27]. ResNet-20 was initially introduced as a HE-friendly model in [38], and we use the improved versions of ResNet-20/44 presented in [36]. The activation functions were implemented based on the algorithm proposed in [37]. In detail, we use the Remez algorithm to approximate SiLU as a 96th-order single polynomial (multiplicative depth of 7). On the other hand, ReLU employed a minimax composite polynomial approximation using degrees \{15, 15, 27\} (multiplicative depth of 13) in [37]. For linear operations such as convolution, we adopt the multiplexed parallel convolution
Table 3: Classification accuracy for multiple CIFAR-10 images using models on the RNS-CKKS scheme generated by DA CAPO.

<table>
<thead>
<tr>
<th>Model</th>
<th>ResNet-20</th>
<th>ResNet-44</th>
<th>AlexNet</th>
<th>VGG16</th>
<th>SqueezeNet</th>
<th>MobileNet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ReLU</td>
<td>SiLU</td>
<td>ReLU</td>
<td>SiLU</td>
<td>ReLU</td>
<td>SiLU</td>
</tr>
<tr>
<td>Pytorch</td>
<td>90.6%</td>
<td>92.6%</td>
<td>91.2%</td>
<td>92.7%</td>
<td>89.0%</td>
<td>86.6%</td>
</tr>
<tr>
<td>DA CAPO</td>
<td>90.7%</td>
<td>92.6%</td>
<td>91.3%</td>
<td>92.7%</td>
<td>89.0%</td>
<td>86.6%</td>
</tr>
</tbody>
</table>

Figure 4: Speedup of algorithms based on the Manual with waterline $2^{40}$. The activation functions utilized in each deep learning model are denoted as (R) for ReLU and (S) for SiLU.

We validate our implementation by comparing the accuracy using 1,000 CIFAR-10 images between PyTorch’s and the DA CAPO programs (with a waterline of $2^{40}$). Table 3 reports the classification accuracy results. ResNet-20(R), SqueezeNet(R) and (S), and MobileNet(R) exhibited a minor difference of 0.1% in accuracy. The usage of approximate activation functions introduced a maximum error of 13 bits, leading to negligible differences in accuracy.

Given the lack of bootstrapping compilers, our evaluation uses manually implemented bootstrapping placement (Manual) as the baseline. According to the state-of-the-art ResNet-20/44 implementation with bootstrapping [36], which inserts bootstrap operations before activation functions only when deemed necessary, we manually insert bootstrapping operations for the benchmark applications. The manual insertion, verification, and optimization take approximately 4 hours for each benchmark for a single waterline. Since an incorrect placement of bootstrapping causes FHE operation errors such as ciphertext scale overflow, sophisticated time-consuming placement analysis and verification are required. For this version, bootstrap operations are added between each layer. In addition, the sign function used in ReLU and maxpooling cannot be executed without bootstrapping, so we incorporate additional bootstrapping at the point where the second 15th-degree term is multiplied. The manual implementation assumes a scale management waterline of $2^{40}$.

We compare DA CAPO’s three different variants with the manual implementation (Manual). Liveness only considers the number of live-out values at program points (without bypass edge or cost analysis). This method greedily utilizes the scale budget in that it inserts bootstrap synchronously while maximizing the utilization of the scale capacity by inserting bootstrap into the closest candidates when the accumulated scale exceeds the scale capacity. Figure 2c represents this scheme. Bypass considers more bootstrap candidates by excluding the bypass edges. Like Liveness, it greedily maximizes the scale capacity utilization but does not perform cost-aware bootstrapping. Figure 2d is an example of this scheme. DA CAPO use all the proposed liveness, bypass, and cost analyses, as illustrated in Figure 2e. It regards the latency of all operations and determines the placement of bootstrap based on the bypass-aware candidates set.

This evaluation runs on Intel(R) Core(TM) i7-12700 for CPU and NVIDIA GeForce RTX 3090 with 24GB memory for GPU, utilizing unified virtual memory (UVM).
Table 4: Bootstrapping counts for each benchmark in manual placement and DA CAPO

<table>
<thead>
<tr>
<th>Model</th>
<th>ResNet-20</th>
<th>ResNet-44</th>
<th>AlexNet</th>
<th>VGG16</th>
<th>SqueezeNet</th>
<th>MobileNet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ReLU</td>
<td>SiLU</td>
<td>ReLU</td>
<td>SiLU</td>
<td>ReLU</td>
<td>SiLU</td>
</tr>
<tr>
<td>Manual</td>
<td>37</td>
<td>19</td>
<td>85</td>
<td>43</td>
<td>66</td>
<td>12</td>
</tr>
<tr>
<td>DA CAPO</td>
<td>37</td>
<td>19</td>
<td>85</td>
<td>43</td>
<td>67</td>
<td>12</td>
</tr>
</tbody>
</table>

8.2 Performance Evaluation

Figure 4 represents the speedup of inference time for a single CIFAR-10 image achieved by each approach (Liveness, Bypass, and DA CAPO) compared to Manual.

In Figure 4, Liveness allows for effective reduction of # bootstrap, which achieves comparable performance to Manual in VGG16(R) and VGG16(S), and even improves the performance in cases like AlexNet(S) and SqueezeNet(S). The main reason Liveness improves the performance is that the compiler considers the bootstrapping points the user cannot consider, such as the points within an activation function. Bypass increases the number of valid liveness candidates, finding more beneficial bootstrap candidates. In ResNet-20/44(R) and VGG16(R), improved latencies can be observed by identifying appropriate points considering liveness. However, in the case of VGG16(S) and AlexNet(S), the total latency actually increases as shown in Figure 5, due to increased latencies in other operations. The geometric mean of speed up for DA CAPO is 1.21x. The reason why DA CAPO achieves better results compared to Manual is that DA CAPO takes into consideration the aspects that users may overlook. DA CAPO has the ability to analyze the detailed RNS-CKKS operations that make up deep learning model APIs. While users can analyze the provided library and insert bootstrap, due to the variability of scale management with each function call, the optimal point for bootstrap can vary each time. DA CAPO, on the other hand, is aware of scale management variations based on the placement of bootstrap, allowing it to discover a lower latency plan than Manual.

8.3 Bootstrapping Counts

In Table 4, the number of bootstrap operations is similar between Manual and DA CAPO. However, in Figure 5, DA CAPO yields apparently better latency for AlexNet, VGG16, and SqueezeNet because DA CAPO places the bootstrapping where the latency of arithmetic is reduced. Interestingly, in the case of MobileNet, DA CAPO involves a higher number of bootstrap operations, yet it still achieves a lower total latency. This is strong evidence that the latency of arithmetic operations has an impact on the overall latency, although each arithmetic operation has multiple orders lower latency than bootstrapping. In the case of MobileNet(R), there are a large number of channels leading to increased rotate and multiplication operations, which outweigh the latency of bootstrap, increasing the number of bootstrap operations can actually result in a lower total latency by lowering the levels of intermediate operations. This demonstrates that simply minimizing the bootstrap count does not guarantee optimal results.

8.4 Compile Time

Table 5 shows the total compile time and bootstrapping management time for each benchmark, along with a comparison of the number of operations and bootstrap candidates. Benchmarks with ReLU have more candidates because the composite polynomial approximation of ReLU has multiple points with a single live-out. For all the benchmarks except AlexNet, the management time is less than 5 minutes, which is acceptable as usual. To discuss the scalability of the algorithm, we perform an in-depth analysis of the compile time.
In bootstrapping management, time-consuming parts are bypass edge detection and the bootstrapping plan decision. The bypass edge detection algorithm, which calculates coverage through scale management for each operation, has a time complexity of $O(N^2)$, where $N$ is the number of operations. On the other hand, Algorithm 2 has a time complexity of $O(D \times d \times N)$, where $D$ is the number of bootstrap candidates and $d$ is the maximum number of bootstrap candidates within the coverage of a certain bootstrap candidate.

Comparison between the ResNet-20(S) and the AlexNet(S) shows the impact of the number of operations on the bypass edge detection clearly ($(50535/9144)^2 = 30.54$ vs $336.5/14.4 = 23.37$). The scale management unit [41] will greatly reduce the effective number of operations, enhancing its scalability. On the other hand, the comparison between AlexNet(R) and AlexNet(S) clearly shows the impact of the number of candidates on the plan decision algorithm. Since the intermediate feature maps cannot be packed in a single ciphertext in AlexNet(R), the candidate filtering threshold is larger than other benchmarks, increasing candidates a lot. We believe that further research on a fine-grained filtering threshold that reflects the program context can improve the scalability of the plan decision algorithm.

### 8.5 Performance Estimation

To evaluate the accuracy of the latency estimation used in Algorithm 2, we compared the estimated latency with the actual latency with 108 data points using 12 benchmarks across 9 different waterlines in Figure 6. The equation, $y = 1.034x$, is a regression model between the estimated and actual costs, and $R^2$ indicates predicted latency closely aligns with the actual latency. Since the coefficient of the regression model and the $R^2$ value (0.9986) are close to 1, the overall precision is quite satisfactory. The small variances in the coefficient and $R^2$ are caused by the memory transfer latency in UVM. We use a GPU that has 24GB of physical RAM, whereas the memory requirement exceeds the capacity in some benchmarks. Therefore, UVM was used for the experiment to fulfill the memory usage, which might have imposed an additional latency to the actual execution time compared to the estimated time in the memory-starving benchmark. Nonetheless, the variance in the performance estimation will not degrade the efficacy of the bootstrapping plan decision, because the penalty of memory starvation will affect the bootstrapping placement plans of the same benchmark consistently.

#### 8.6 Waterline Sensitivity

Figure 7 illustrates the latency for deep learning benchmarks with respect to the waterline. The approximation of the activation function has a maximum 13-bit error. To align the error caused by RNS-CKKS operations with the approximation error of activation functions, the waterline needs to be larger than $2^{30}$ in our measurement. The latency of Manual is plotted as a point based on a waterline of $2^{40}$. A lower waterline gives a different scale management plan that slows down the scale accumulation, leading to different optimal positions for bootstrapping placement. Greedy bootstrapping placements (Liveness and Bypass) can find reasonable positions as observed in ResNet-20 and SqueezeNet but result in a significant variance of latency for AlexNet and MobileNet, because the position of bootstrap significantly changes the latency of arithmetic operation. On the other hand, DA CAPO exhibits lower variation with respect to the waterline and shows a decreasing trend of total latency as the waterline decreases. For Figure 7k, there is a small variance depending on the waterline. The variance comes from the scale management scheme (the proactive rescaling of Hecate) that does not optimize the latency of arithmetic operation consistently and produces sub-optimal scale management plans. Hecate [41] conducts...
space exploration to find the optimal points but requires significantly longer compile time due to the larger benchmark size. Further improvement on the scale management scheme could lead to improvements in this aspect.

### 9 Related Work

**FHE compilers.** The existing compiler works [4–6, 9, 13, 15, 18, 19, 35, 39–41, 43, 50, 51, 53], have proposed optimization techniques to improve performance, accuracy, and programmability. EVA [18] introduces the concept of *waterline* which signifies the minimum required scale and automates inserting scale management operations. However, EVA misses the opportunity for optimization as it uses fixed-factor rescale. To address the limitation of EVA, HECATE [41] proposes downscale operation which enables proactive rescale and finds an optimal scale management plan by using space explorations. ELASM [39] points out that using a fixed waterline in HE applications does not reflect errors during scale management. With a new error-latency-aware scheme, ELASM produces better latency and error results than prior works [18,41].

DaCAPO and the existing compilers such as Hecate and ELASM have complementary relations. This work focuses on how to partition DNN models using bootstrapping, while the other compilers focus on scale management optimization of each partition. The bootstrapping management algorithm of DaCAPO can be applied in conjunction with any scale management scheme. However, simply applying Hecate or ELASM to this work is impractical due to their high exploration overheads. Hecate and ELASM optimize ciphertext scales by exploring scale management spaces, and scale management space exploration takes huge compilation overheads. Since DaCAPO generates multiple different bootstrapping insertion candidates, and optimizes ciphertext scales that are reinitialized at each bootstrapping insertion point, exploration-base scale management like Hecate and ELASM would severely amplify the compilation time. Thus, this work adopts only a part of Hecate such as the proactive rescaling scheme to manage ciphertext scales, while excluding the scale management space exploration from Hecate.
In addition, performance improvement is achieved through effective data layout and parallelization methods for ciphertexts. CHET [19] automatically selects the encryption parameters and data layout while ensuring security and accuracy of the target tensor circuit. EVA [18] and ALCHEMY [16] also provide data packing, but still lack usability as programmers have to perform vectorization manually. Coyote [43] allows automatic vectorizing for arbitrary applications considering data movement. HECO [50] incorporates circuit optimizations to reduce noise growth, as well as target hardware and scheme-specific optimizations, enabling developers to achieve performance similar to experts. However, none of the compilers support large benchmarks that require bootstrapping.

**Privacy Preserving Machine Learning (PPML).** A Convolution algorithm for homomorphic encryption is proposed by Gazelle [30], and [3,28,32,44,47,52] perform Convolution Neural Network (CNN) based on this algorithm. These works adapt multi-party computation to perform non-linear functions on secure side, which compromised the advantages of homomorphic encryption and required significant communication overhead. [38] first implements the standard ResNet-20 with the RNS-CKKS FHE with bootstrapping, but a substantial amount of bootstrapping operations and convolutions lead to performance degradation. To improve the efficiency of the application, [36] constructs an efficient very deep standard CNN model on FHE, which minimized the bootstrapping runtime by applying multiplexed packing and multiplexed parallel convolution algorithm. All existing PPML are hand-optimized and do not reflect scale management. Additionally, programmers need to manually insert bootstrapping for each change in waterline and model structure, which increases their burden and might create target programs inefficiently.

**Automatic Bootstrap Placement.** Previous work [45] proves that finding the minimal number of bootstrap for a given FHE program and corresponding bootstrapping placement is an NP-complete problem. They propose heuristics for bootstrapping placement based on a mixed-integer linear programming method. Compared to DAPO, they only consider the multiplicative depth not reflecting scale management, and do not consider the latency of the other operations. Furthermore, they do not automatically generate the bootstrap-managed code to reduce the programming burden.

This work proposes an automatic bootstrapping management compiler to reduce the burden on developers and enable easy privacy-preserving use of large-size models. Also, the results show improved performance to manual-implemented models and demonstrate the minimum number of bootstrapping counts does not always result in better performance.

### 10 Conclusion

This paper introduces DAPO, which is the first automatic bootstrapping management compiler. DAPO efficiently restricts bootstrapping candidates through liveness analysis, considering bypass edges. Additionally, DAPO finds the bootstrapping plan with the minimum cost based on these candidates. The results demonstrate that DAPO consistently outperforms manually implemented FHE programs on deep learning models, achieving an average speedup of $1.21\times$.

### Acknowledgements

We appreciate the valuable feedback and shepherding from anonymous reviewers and the support from CoreLab members during this work. This work is partly supported by IITP-2020-0-01847, IITP-2020-0-01361, IITP-2021-0-00853, IITP-2022-0-00050, and No. RS-2023-00277060 funded by the Ministry of Science and ICT, South Korea. This work is also supported by Samsung Electronics and CryptoLab. This work is in part supported by the grants CNS-2135157, CCF-2153747, and CNS-2214980, funded by the National Science Foundation, U.S.A. (Corresponding author: Hanjun Kim)

### Availability

The source code of DAPO will be accessible to the public at [https://github.com/corelab-src/elasm](https://github.com/corelab-src/elasm).

### References


Appendix A Existing RNS-CKKS Compilers

Existing RNS-CKKS compilers propose automatic scale management schemes that insert scale management operations like rescale and modswitch to the input program that does not include scale management operations with optimizing performance and error.

EVA [18] proposes the first scale management algorithm that attempts to minimize the accumulated scale. A smaller accumulated scale is preferred as it allows the compiler to use smaller encryption parameters coefficient modulus $Q$ and polynomial modulus $N$ at the same security level, which in turn is expected to result in better performance. To this end, EVA regards the minimum scale of input ciphertexts to a program as the minimum scale requirement, called waterline $S_{w}$, of all ciphertexts. Then it inserts a rescale operation if the scale after rescaling is higher than the waterline: i.e., $m/S_{f} \geq S_{w}$. Besides, EVA uses upscale and modswitch to meet the same level and scale requirements of arithmetic operations as needed.

Hecate [41] introduces a new scale management operation, called downscale, which combines upscale and rescale operations and supports decreasing the scale of a ciphertext by an arbitrary amount. Note that rescale can only reduce the scale by the fixed rescale factor $S_{f}$. Hecate makes use of the downscale operation to proactively bring the scale of a ciphertext down to the waterline $S_{w}$, and shows that it can achieve a lower accumulated scale than EVA. Furthermore, Hecate shows that a lower accumulated scale does not always lead to better performance, and presents a greedy latency-aware scale management algorithm based on static cost estimation. Recall that scale management operations modswitch and rescale change the levels of ciphertexts, affecting the latencies of the following arithmetic and rotation operations (Table 2). Combined with downscale and a greedy search algorithm, Hecate demonstrates a better performance (lower latency) than EVA.

ELASM [39] improves Hecate by reflecting the noise of RNS-CKKS operations. Because the RNS-CKKS operations add operation-specific and scale-independent noise to the result, the scale of the result affects the error of the operation by dividing the noise by the scale. To incorporate error awareness into the scale management scheme, ELASM proposes a noise-aware waterline that controls the waterline of each operation in a fine-grain manner reflecting the operation noise. To control the error level, ELASM controls the waterline based on the user-level compilation parameter called Scale-to-Noise-Ratio (SNR). Furthermore, ELASM proposes an efficient error estimator and error-aware exploration-based optimizer called Error-Latency-Aware Scale Management (ELASM) based on the error estimation. Combined with a noise aware waterline and an efficient error estimation scheme, ELASM achieves 4.2 bits better error and 16.7% better performance compared to Hecate.
Table 6: Function Parameters of AlexNet. Based on the multiplexed parallel packing method presented in [36], we implemented the convolution and pooling layers, using the same parameter notation.

| Layer   | $n_i$ | $n_o$ | $f_h$ | $f_w$ | $s$ | $h_i$ | $h_o$ | $w_i$ | $w_o$ | $c_i$ | $c_o$ | $k_i$ | $k_o$ | $t_i$ | $t_o$ | $p_i$ | $p_o$ | $q$ |
|---------|-------|-------|-------|-------|-----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----|
| ConvBN1 | 1     | 2     | 3     | 3     | 1   | 32    | 32    | 32    | 32    | 3     | 96    | 1     | 1     | 3     | 96    | 16    | 1    | 6   |
| AvgPool1| 2     | 1     | 3     | 3     | 2   | 32    | 16    | 32    | 16    | 96    | 96    | 1     | 2     | 96    | 24    | 1     | 2    | 96  |
| ConvBN2 | 1     | 1     | 5     | 5     | 1   | 16    | 16    | 16    | 16    | 96    | 256   | 2     | 2     | 24    | 64    | 2     | 1    | 128 |
| AvgPool2| 1     | 1     | 3     | 3     | 2   | 16    | 8     | 16    | 8     | 256   | 256   | 2     | 4     | 64    | 16    | 1     | 4    | 256 |
| ConvBN3 | 1     | 1     | 3     | 3     | 1   | 8     | 8     | 8     | 8     | 256   | 384   | 4     | 4     | 16    | 24    | 4     | 2    | 96  |
| ConvBN4 | 1     | 1     | 3     | 3     | 1   | 8     | 8     | 8     | 8     | 384   | 384   | 4     | 4     | 24    | 24    | 2     | 2    | 192 |
| ConvBN5 | 1     | 1     | 3     | 3     | 1   | 8     | 8     | 8     | 8     | 384   | 384   | 4     | 4     | 24    | 24    | 2     | 2    | 192 |
| AvgPool3| 1     | 1     | 3     | 3     | 2   | 8     | 4     | 8     | 4     | 256   | 256   | 4     | 8     | 16    | 4     | 4    | 16  | 64  |

Appendix B Parameters Used in Evaluation

Table 6 shows the function parameters that are used in each component of the AlexNet structure. ConvBN refers to a function that combines convolution and batch normalization, and AvgPool represents average pooling. The implementation of convolution and pooling layers is based on the paper from [36]. $n_i$ and $n_o$ denote the numbers of input and output ciphertexts. The tuples \{$w_i, h_i, c_i\}, \{$w_o, h_o, c_o\}$ represent input and output tensors. A kernel with a size $f_h \times f_w$ strides by $s$. The remaining parameters such as $k_i,k_o,t_i,t_o,p_i,p_o,$ and $q$ are related to the multiplexed parallel convolution method presented in [36]. Finally, the fully connected layer is implemented using the diagonal method in [23].