Unrolled Memory Inner-Products: An Abstract GPU Operator for Efficient Vision-Related Computations

Yu-Sheng Lin
National Taiwan Univ.
johnjohnlys@media.ee.ntu.edu.tw

Wei-Chao Chen
Skywatch Inc.
weichao.chen@skywatch24.com

Shao-Yi Chien
National Taiwan Univ.
sychien@ntu.edu.tw

Abstract

Recently, convolutional neural networks (CNNs) have achieved great success in fields such as computer vision, natural language processing, and artificial intelligence. Many of these applications utilize parallel processing in GPUs to achieve higher performance. However, it remains a daunting task to optimize for GPUs, and most researchers have to rely on vendor-provided libraries for such purposes.

In this paper, we discuss an operator that can be used to succinctly express computational kernels in CNNs and various scientific and vision applications. This operator, called Unrolled-Memory-Inner-Product (UMI), is a computationally-efficient operator with smaller code token requirement. Since a naïve UMI implementation would increase memory requirement through input data unrolling, we propose a method to achieve optimal memory fetch performance in modern GPUs. We demonstrate this operator by converting several popular applications into the UMI representation, and achieve 1.3x-26.4x speedup against frameworks such as OpenCV and Caffe.

1. Introduction

The recent resurgence of convolution neural networks (CNNs) in vision applications are enabled in large part by the rapid advances of graphics processing units (GPUs), and researchers have been utilizing deeper networks with more training data than ever before [14, 27]. However, owing to the complexities in computational scheduling, it remains difficult to program these GPUs at full efficiency despite progress in programming tools. Because of this, processor vendors have created higher-level primitives for deep learning, such as NVIDIA's cuDNN [4], that can achieve extremely high computational efficiencies on GPUs. However, details of the implementation are inaccessible to the research and open-source community, making it difficult to adapt these techniques to newer algorithms. GPUs also can be sub-optimal in terms of power efficiency because, for example, the math precision requirements for DNNs tend to be less than what is needed for scientific computations.

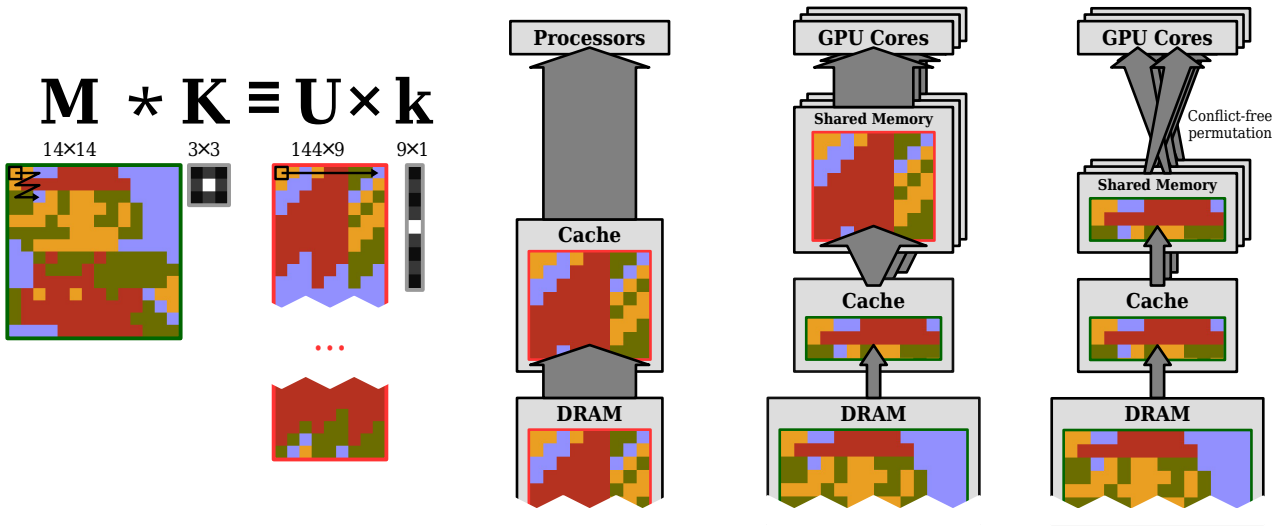
In order to pursue a higher computation-to-power ratio, several application-specific integrated circuits (ASICs) have been created, and some can even operate with entropy-compressed data directly [8]. These ASICs tend to be equipped with fairly large and expensive on-chip memory, and it can be difficult to adapt these ASICs to newer algorithms, such as batch normalization [12] and deconvolution layer [22]. We can gain additional flexibilities with field programmable gate arrays (FPGAs), but they tend to be slower and less area-efficient. Programming on these FPGAs can also be difficult and hardware-specific, making it difficult to transfer between different platforms.

Recently, researchers have proposed domain-specific languages or abstract models of computation, such that programmers can focus on algorithm designs, and leave the specific optimizations to the tools. To achieve maximal performance, someone uses complex algorithms to search for the best parameters and scheduling [25], which can be very time-consuming and again, platform and algorithm dependent.

In this paper, we propose a new primitive which focuses on kernels frequently appear in computer vision and scientific computations. This primitive, the unrolled memory inner-product (UMI) operator, is implemented with C++11 and CUDA, and can be easily integrate into existing algorithms to achieve higher performance than hand-tuned open-source code. Specifically, our contributions are

- An UMI operator that can be used as computational kernels for various vision-related applications,
- an abstraction model that results in smaller code tokens compared to naïve CPU implementations, and
- a methodology to implement UMI to achieve conflict-free and optimal memory access pattern in modern parallel processors.

The rest of the paper is organized as follows. We start by describing memory unrolling, or lowering, techniques for



(a) Convolution via unrolled matrix multiplication (b) A naïve implementation (c) Unroll via shared memory (d) Direct implementation

Figure 1. Taxonomy of convolution techniques. The implementation difficulty increases from (b) to (d). The proposed operator is simple to implement as if in (b), but its memory efficiency is equivalent to (d).

CNN, and generalize the techniques to the proposed UMI operator. We then describe how to implement this operator on NVIDIA GPUs, utilize it on several relevant applications, and compare the efficiency of our operator against state-of-the-art implementations.

2. Related Work

High-Performance CNN Implementations. Figure 1 shows the taxonomy of convolution techniques, which is the core kernel operation in CNNs. Many CNN implementations on GPUs inherit techniques directly from linear algebra libraries [15, 20], which tend to be highly-optimized routines that rely on non-disclosed knowledge about GPU designs. Algorithms reducing to these primitives can utilize these libraries to achieve fairly good performance. For example, Caffe [13] and early versions of cuDNN [4] unroll the input feature maps into matrices, and compute convolution through matrix multiplications similar to Figure 1(a). Vasilache *et al*. [30] converts the problem to frequency domain, but this approach is more suitable for larger kernel sizes.

Unrolling the input data can be expensive, so it is preferable to unroll as late as possible on, say, the cache or on-chip memory. For example in DianNao [2], a (16×16)-by-(16) matrix-vector processor is used for both CNN and FC layers as in Figure 1(c); Lavin *et al*. [16] implement an optimized version of convolution directly on the GPUs; Chen *et al*. [3] create a systolic array architecture to tackle this problem as in Figure 1(d).

Abstract Computational Models. Several domain-specific languages have been designed to overcome the difficulties in developing for various GPUs or ASICs platforms. Programs are represented in abstract models and compiled to kernels specifically optimized for target platforms. For example, Pochoir and PolyMage [28, 19] support stencil and resample operations such as Harris Corner [10] and PDE solving. Halide [25] defines images as N-dimension functions, and describes the image pipelines as functional compositions. It also includes an auto-tuning framework by searching through all parameters within the image pipeline; In their recent work [18], the tuning time is decreased by limiting the parameter search space. Darkroom [11], a subset of Halide, retains the stencil operations and uses linear programming to determine the number of buffers during hardware architecture generation.

Our research is also related to MapReduce [6, 5], where a `map` function maps the input to `(key,value)` pairs, and then matched data are collected and joined by a `reduce` function. In this context, programmers only need to define some critical strategy functions and do not need to worry about implementation details. Convolution Engine [24] is a hardware map-reduce engine with restrictions; it only allows mapping and reduction within a local window with simple pre-defined operators on basic types, such as addition, multiplication and logic.

3. The UMI Operator

Given an input image $\mathbf{A}$ and a kernel $\mathbf{B}$, a convolution between them produces a new image $\mathbf{C}$, where

$$\underbrace{\begin{bmatrix} a_{00} & a_{01} & \dots \\ a_{10} & a_{11} & \\ \vdots & & \end{bmatrix}}_{\text{Input}} * \underbrace{\begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}}_{\text{Kernel}} = \underbrace{\begin{bmatrix} c_{00} & c_{01} & \dots \\ c_{10} & c_{11} & \\ \vdots & & \end{bmatrix}}_{\text{Output}} \quad (1)$$

An early CNN implementation technique involves converting convolutions into matrix multiplications (Figure 1(a)). This method is still being used in the open-source implementation of Caffe. This process requires unrolling the input image $\mathbf{A}$ and vectorizing the kernel $\mathbf{B}$, which allows one to convert convolutions into GEMVs (GEneral Matrix-Vector multiplication), as shown in Eq. (2),

$$\underbrace{\begin{bmatrix} a_{00} & a_{01} & a_{10} & a_{11} \\ a_{01} & a_{02} & a_{11} & a_{12} \\ & \vdots & & \end{bmatrix}}_{\text{Unrolled input}} \underbrace{\begin{bmatrix} b_{00} \\ b_{01} \\ b_{10} \\ b_{11} \end{bmatrix}}_{\text{Vectorized kernel}} = \underbrace{\begin{bmatrix} c_{00} \\ c_{01} \\ c_{02} \\ \vdots \end{bmatrix}}_{\text{Vectorized output}} \quad (2)$$

For simplicity, we will denote the unrolled input of $\mathbf{A}$ in Eq. (2) as $U_1(\mathbf{A})$, which we shall formally define later.

3.1. Generalized Inner-Product

Pushing the process to extreme, we could unroll both the input and the kernel to the same sizes. In Eq. (3), we apply a different unroll operator to $\mathbf{B}$ by repeatedly stacking the kernels,

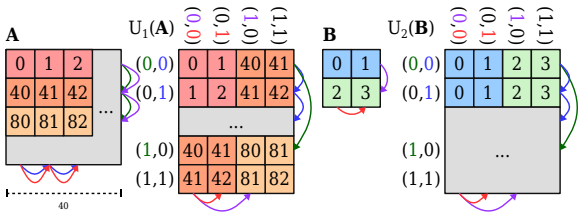
$$U_2(\mathbf{B}) = \begin{bmatrix} b_{00} & b_{01} & b_{10} & b_{11} \\ b_{00} & b_{01} & b_{10} & b_{11} \\ & \vdots & & \end{bmatrix}, \quad (3)$$

and the GEMV in Eq. (2) can be reformulated into Eq. (4), which defines a UMI operator::

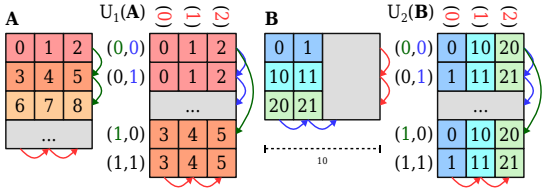
$$U_1(\mathbf{A}) \odot U_2(\mathbf{B}), \quad (4)$$

where $\odot$ is a generalized inner-product operator between two vectors, and Eq. (4) is just a convenient notation which means $\odot$ is applied to every rows of $U_1(\mathbf{A})$ and $U_2(\mathbf{B})$. When $\odot$ is defined as the dot-product, Eq. (4) exactly produces the same results as Eq. (2); when $\mathbf{v}_1 \odot \mathbf{v}_2 \equiv \max(\mathbf{v}_1^T \mathbf{v}_2, 0)$, Eq. (4) produces a CNN+ReLU layer; when $\mathbf{v}_1 \odot \mathbf{v}_2 \equiv ||\mathbf{v}_1 - \mathbf{v}_2||_1$, this operator can express patch-wise differences such as motion estimation or stereo matching.

We will demonstrate later that with the formal definition of $U(\cdot)$, many vision or scientific kernels can be expressed through the UMI operator.

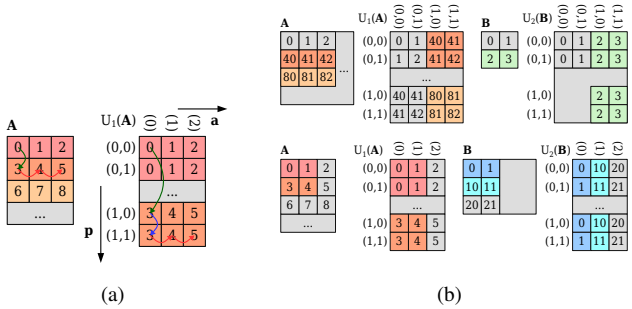


(a) Convolution: each row in $U(\mathbf{A})$ is a local window from $\mathbf{A}$.



(b) Matrix multiplication: each row in $U(\mathbf{B})$ is a column from $\mathbf{B}$.

Figure 2. Different memory layouts of Eq. (4) when the UMI operator is used to represent different tasks.



(a) (b)

Figure 3. (a) illustrates accessing an element of $U(\mathbf{A})$ with indices $\mathbf{p} = (1,1), \mathbf{a} = (2)$; (b) shows that the footprint of a sub-tensor in $U(\cdot)$ is also a sub-tensor in the original tensor.

3.2. Tensor Representation of Unrolling $U(\cdot)$

Figure 2(a) shows the detailed memory layout of Eq. (2,3), where rows and columns of $U_1(\mathbf{A})$ and $U_2(\mathbf{B})$ are assigned with grid indices like $(0,0)$, $(0,1)$, $(0,2)$, $(1,0)\cdots$. Figure 2(b) illustrates another layout when the desired task is GEMM (GEneral Matrix-Matrix multiplication). Increasing an index component in an unrolled matrix, says $U_1(\mathbf{A})$, corresponds to travelling along a particular axis at a certain stride in the original matrices $\mathbf{A}$, and such correspondences are highlighted with colored numbers and arrows in Figure 2. Notice that some components may have zero stride where the values are just repeated, and their corresponding arrows are omitted in the original matrices $\mathbf{B}$.

Based on the discussions above, the operator $U(\cdot)$ is actually a function mapping a tensor to another, and Figure 2 is a visualization of flattening high-dimensional tensors $U_1(\mathbf{A})$ and $U_2(\mathbf{B})$ into 2D matrices. The flattening pro-

cess is based on splitting the tensor indices into two parts, row indices $\mathbf{p}$ and column indices $\mathbf{a}$, and we use the notation $U(\mathbf{A})_{\mathbf{p},\mathbf{a}}$ to refer to one element in the unrolled tensors. Changing the j-th component of the indices k_j, which might belong to either $\mathbf{p}$ or $\mathbf{a}$, is equivalent to moving along a particular axis d_j of $\mathbf{A}$ with a certain stride and offset s_j, o_j:

$$U(\mathbf{A})_{\mathbf{p},\mathbf{a}} = \mathbf{A}_{\mathbf{x}}, \text{ where } x_i = \sum_j \delta_{i,d_j}(k_j s_j + o_j). \quad (5)$$

The equation above first calculates an indices $\mathbf{x}$ with $\mathbf{p}, \mathbf{a}$ and unroll parameters d_j, s_j, o_j, then uses $\mathbf{x}$ to locate a specific element $\mathbf{A}_{\mathbf{x}}$ in $\mathbf{A}$.

Figure 3(a) illustrates access of element $U_1(\mathbf{A})_{(1,1),(2)}$, which is equal to accessing $\mathbf{A}_{(1,2)} = 5$ by moving down once and toward the right twice on $\mathbf{A}$.

3.3. The Programmatic Aspect of UMI

In the previous section we define $U(\mathbf{A})$ as a tensor, but from the programmatic view $U(\mathbf{A})$ is a function object, or a functor, constructed by binding the unroll parameters and underlying tensor $\mathbf{A}$ to a function. Accessing the element $U(\mathbf{A})_{\mathbf{p},\mathbf{a}}$ then becomes calling the functor $U(\mathbf{A})$ with arguments $\mathbf{p}$ and $\mathbf{a}$ to locate the element in $\mathbf{A}$ by an inverse look-up with Eq. (5). As a result, programmers simply code as if they were operating on the large unrolled tensors $U(\mathbf{A})$, while the *on-the-fly unroll via loop-up* mechanism prevents the overhead of actually storing $U(\mathbf{A})$.

In the detailed implementation, the operator $\odot$ is not necessarily limited to a binary operator, so Eq. (4) can be rewritten to a more general form:

$$U_3(\mathbf{C}), \cdots = \odot(U_1(\mathbf{A}), U_2(\mathbf{B}), \cdots), \quad (6)$$

where the operator $\odot$ becomes a callable function with multiple arguments and returns. We will later illustrate the usefulness of such generalization by applying UMI to more computational kernels in Section 5. Also, since vectorizing can be considered as a special case of unrolling, we write the vectorized $\mathbf{C}$ of Eq. (2) as $U_3(\mathbf{C})$, which make the abstraction more concise and simple to implement.

4. Tile-Based Execution Partition

A basic implementation of the UMI operator includes several nested *for loops*, and is therefore well suited for parallel processing. Before presenting the algorithm, we would like to point out the necessity to represent $\odot$ as a *strategy class*, as shown through an example in Listing 1,

Listing 1. The $\odot$ definition for CNN+ReLU layer

```
class ReLUStratgy {
   float act;
   void PreLoop() {act = 0;}
   void Loop(float a, float b) {act += a*b;}
   float PostLoop() {return max(act,0);}
};
```

Here, `PreLoop` initializes the activation value `act` to zero, and then `Loop` is called over the range of rows to accumulate the pairwise products to `act`. Finally `PostLoop` returns the activation value `max(act,0)` as the output. Procedure 1 shows the pseudo code of a basic UMI operator.

Procedure 1 The basic UMI algorithm

Input: Unrolled memory $U_1(\mathbf{A}), U_2(\mathbf{B}), U_3(\mathbf{C})$
Input: 4D parallelism size $\mathbf{w}_p = (w_1, w_2, w_3, w_4)$
Input: 4D accumulation size $\mathbf{w}_a = (w_4, w_5, w_6, w_7)$
Input: Inner-product strategy UmiStrategy
  for $\mathbf{p}$ in NDRange($\mathbf{w}_p$) **do**
    UmiStrategy umi; {Initialize a class object}
    umi.PreLoop();
    for $\mathbf{a}$ in NDRange($\mathbf{w}_a$) **do**
      umi.Loop($U_1(\mathbf{A})_{\mathbf{p},\mathbf{a}}, U_2(\mathbf{B})_{\mathbf{p},\mathbf{a}}$);
    end for
    $U_3(\mathbf{C})$ = umi.PostLoop();
  end for

In Procedure 1, the *for loop* indices $\mathbf{p}, \mathbf{a}$ are exactly the row and column indices as shown in Figure 3(a), and the sizes of $U_1(\mathbf{A})$ and $U_2(\mathbf{B})$ are both $\mathbf{w}_p \times \mathbf{w}_a$. The outer loop of Procedure 1 can be parallelized easily, and the *on-the-fly unroll* mechanism described in Section 3.3 can benefit from caches. When different elements in $U_1(\mathbf{A})$ or $U_2(\mathbf{B})$ refer to the same elements in the original $\mathbf{A}$ or $\mathbf{B}$, duplicated access can be eliminated by caches and thus the traffic to the main memory can be greatly reduced.

However, for GPU-friendly implementations, additional considerations must be taken. GPUs tend to have very long cache latency, and it is useful to provide a faster scratchpad memory for threads in the same processing groups. Such memory, called shared memory in CUDA or local memory in OpenCL, has a much higher throughput and lower latency compared to the cache, and we shall next explain how to take advantage of shared memory with the UMI operator.

4.1. Tiled Parallel Execution

For a UMI operator to support tiled execution, we can simply divide $U_1(\mathbf{A})$ and $U_2(\mathbf{B})$ into sub-tensors of size $\mathbf{t}_p \times \mathbf{t}_a$. Figure 3(b) highlights some sub-tensors from Figure 2 with $(\mathbf{t}_p, \mathbf{t}_a) = ((2,2),(1,2))$ and $((2,2),(2))$, respectively. From the figure it can be observed that, for any arbitrary tensor $\mathbf{A}$ and it's unrolled version $U(\mathbf{A})$, each sub-tensor $U(\mathbf{A})^{\text{sub}}$ of $U(\mathbf{A})$ can be fully defined by a sub-tensor $\mathbf{A}^{\text{sub}}$ of $\mathbf{A}$. The i-th side length of tensor $\mathbf{A}^{\text{sub}}$ is calculated by this footprint analysis:

$$1 + \sum_j (t_j - 1) s_j \delta_{d_j,i}. \quad (7)$$

Because computations for different rows can be carried out in parallel, grouping $\mathbf{t}_p$ rows together is thus equivalent to selecting the thread block size in CUDA, where a block with $\mathbf{t}_p$ threads could perform $\mathbf{t}_a$ `Loop()` operations within the elements of $\mathbf{A}^{\text{sub}}$. Programmers are responsible for choosing appropriate $(\mathbf{t}_p, \mathbf{t}_a)$ to ensure the size of $\mathbf{A}^{\text{sub}}$ is reasonable.

Procedure 2 describes the complete execution of the tiled UMI model. Instead of loading an elements of $U(\mathbf{A})$ directly from $\mathbf{A}$, we first load the sub-tensor of $\mathbf{A}^{\text{sub}}$ into the shared memory once, and programmatically unroll the tensor to $U(\mathbf{A})^{\text{sub}}$ on-the-fly, thereby reducing both memory bandwidth and consumption to the degree comparable to unroll-free implementations.

Procedure 2 The Tiled UMI algorithm

Input: Unrolled memory $U_1(\mathbf{A}), U_2(\mathbf{B}), U_3(\mathbf{C})$
Input: 4D parallelism/accumulation size $\mathbf{w}_p, \mathbf{w}_a$
Input: 4D parallelism/accumulation tile size $\mathbf{t}_p, \mathbf{t}_a$
Input: Inner-product strategy UmiStrategy
  for $\mathbf{p}_b$ in NDRange$(\mathbf{w}_p/\mathbf{t}_p) \times \mathbf{t}_p$ **do**
    for $\mathbf{p}_l$ in NDRange$(\mathbf{t}_p)$ **do**
      $\mathbf{p} = \mathbf{p}_b + \mathbf{p}_l$
      UmiStrategy umi;
      umi.PreLoop();
      for $\mathbf{a}_b$ in NDRange$(\mathbf{w}_a/\mathbf{t}_a) \times \mathbf{t}_a$ **do**
        {Load $\mathbf{A}^{\text{sub}}$ and $\mathbf{B}^{\text{sub}}$ into shared memory according to $\mathbf{p}_b$ and $\mathbf{a}_b$}
        for $\mathbf{a}_l$ in NDRange$(\mathbf{t}_a)$ **do**
          umi.Loop$(U_1(\mathbf{A})^{\text{sub}}_{\mathbf{p}_l,\mathbf{a}_l}, U_2(\mathbf{B})^{\text{sub}}_{\mathbf{p}_l,\mathbf{a}_l})$;
        end for
      end for
      $U_3(\mathbf{C})_{\mathbf{p},\mathbf{0}}$ = umi.PostLoop();
    end for
  end for

Register Tiling. Registers can act as another faster memory layer between the shared memory and processor cores. This technique, called register tiling, remains a key technique for thoroughly harnessing the power of GPUs. For the readers who are interested in how to implement nearly fully utilized kernels with register tiling, we recommend them to read [31, 15, 20].

In UMI, register tiling is equivalent to selecting a larger sub-tensor size $(e\mathbf{t}_p) \times \mathbf{t}_a$, where $e \in \mathcal{N}$ is a compile-time constant selected by programmers. From the previous discussion, a larger thread block size must be used in this case, but with register tiling we use the same thread block size to simulate a larger one. The benefits of register tiling include decreasing the amount of loads from shared memory, increasing the effective instruction density, and allowing the sharing of address calculation across smaller sub-tensors. In our implementation of GEMM, $e = 4$ is selected, while $e = 8$ is possible in heavily tuned kernels.

4.2. Bank Conflict Avoidance

Most parallel computing systems are designed with matched number of processors and SRAM banks, and the performance drops drastically when processors request elements from the same SRAM at the same time. This problem, called bank-conflict, is not addressed in many papers about computation abstraction [25, 18, 19, 28], and is usually discussed in more specific applications like Eyeriss [3], which supports different kernel strides for CNN.

Two of the most common solutions are padding[21, chap. 39] and XOR-hash[17, 7, 9], which have been proven to be useful for many specific cases. Both techniques cause small runtime overhead: padding wastes SRAM spaces, while XOR-hash requires extra instructions during indexing. To avoid bank-conflict for the UMI operator, we use both as well as the re-tiling technique.

Dealing with bank conflict is equivalent to finding division schemes of sub-tensor $U(\mathbf{A})^{\text{sub}}$ in UMI, such that each group is perfectly shuffled into different banks from the original $\mathbf{A}^{\text{sub}}$. First, columns in $U(\mathbf{A})^{\text{sub}}$ is split spontaneously since elements with different accumulation indices $\mathbf{t}_a$ are executed sequentially; second, since the number of processors is fixed, the elements in a column are also scheduled to be accessed at different time slots.

Figure 4 illustrates our solutions to avoid bank-conflict for UMI, where (i) shows an example that $4 \times 4 = 16$ threads perform a 3×3 convolution task under an 8-processor/8-bank configuration. With the native tiling, the thread block are split into two 2×4 groups as shown in (ii). Two possible access patterns are indicated in (ii) with different numbers. In (ii-a), conflicts for both groups are highlighted by bold borders, whereas (ii-b) resolves bank conflicts by padding 6 elements every 6 elements. To reduce the padding waste, (ii-c) uses only 2 padding with XOR-hash, obtained by swapping the first and last four elements in every other row. (iii) shows two different re-tiled grouping manners, and both could create conflict-free scenarios without any padding requirement in (iii-a).

Proof for Conflict-Free Execution for UMI In this part, we sketch out the proof of our bank-conflict avoidance scheme based on the examples in Figure 4. When a group accesses the elements in $\mathbf{A}^{\text{sub}}$, the addresses without XOR-hash can be written as a set $\mathcal{S} = \{c + pi + qj + rk \mid i, j, k \in \mathcal{Z}_2\}$, where $|\mathcal{S}| = 8$ and c is an arbitrary integer standing for the starting address of the group. For example, in (ii-b) and (iii-a), those addresses

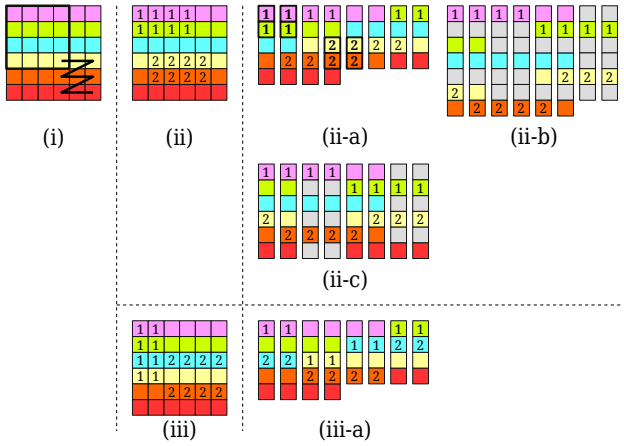


Figure 4. (i) illustrages a block of 4×4 threads is perform a 3×3 convolution, requesting 6×6 shared memory. (ii) and (iii) show a few accessing patterns under different grouping scheme; the leftmost figures show how the 36 elements are padded and distributed to 8 SRAM banks.

of the groups could be generated by $(p, q, r) = (12, 2, 1)$ and $(12, 6, 1)$. If $\mathcal{S} \bmod 8$, the bank numbers where the elements lie in $\mathbf{A}^{\text{sub}}$, are unique, then bank conflict won't happen, and such uniqueness can be proven by contradiction.

The tuple (p, q, r) is $(1, 2, 8)$ in (ii-c), and a XOR-hash function is used to prevent the bank conflict. By placing the a-th element at address $h(a) = \texttt{a\^{}(a\&8)>>1}$, a hash function XOR-ing the 3rd bit onto the 2nd bit of a, the set of bank numbers become $h(\mathcal{S}) \bmod 8$, which should be unique in order to execute without bank conflict.

To prove the uniqueness, we will establish a contradiction on finding two equal elements with $(i_1, j_1, k_1) \neq (i_2, j_2, k_2)$ in $h(\mathcal{S}) \bmod 8$. The proof starts from the number in (p, q, r) with the least trailing zeros when represented in binary, which is $p = 1 = 1_2$. Eq. (8) says that adding numbers with more trailing zeros does not flip the lower bit(s) and thus implies $i_1 = i_2$.

$$\text{bit 0 of } h(c + i + 2j + 8k) = \text{bit 0 of } h(c + i) \quad (8)$$

The proof goes on $q = 2 = 10_2$ and $r = 8 = 1000_2$, and $j_1 = j_2$, $k_1 = k_2$ can be proven by inspecting one bit at a time and by induction, establishing the contradiction as desired. Also, note that $s_j = 0$ in Eq. (5) always generates addresses to the same bank, but these become broadcasts which are free for most parallel processors.

5. Application and Evaluation

5.1. Token Efficiency

Listing 2 and 3 shows a Bilateral Filter implemented with the UMI operator. In `Loop()`, the neighbour pixels and two corresponding spatial weights are packed in the array `i`. Listing 3 specifies the input as an h-by-w image divided into 16-by-16 blocks, and each thread in a block performs a k-by-k local window scan. The middle lines represent the $U(\cdot)$s, and the range term σ_r is passed to the kernel at the last line.

Listing 2. An UMI strategy for bilateral filters.

```
class BilateralStrategy {
   float wsum, wxsum, center;
   struct Constant {float norm;};
   PreLoop(0,1) {wsum = wxsum = 0; center = i[0];}
   Loop(0,3) {
      float d = n-i[0];
      float w = expf(d*d*c.norm)*i[1]*i[2];
      wsum += w; wxsum += w*i[0];
   }
   PostLoop(1,0) {o[0] = wxsum/wsum;}
};
```

Listing 3. An UMI execution routine for bilateral filters .

```
UMI::Execute<Bilateral>(
   // size and tile size
   {{h,16}, {w,16}}, {{k,k}, {k,k}},
   // d_i, o_i, s_i for input
      {{0,0,1}, {1,0,1}},
      {{0,-k/2,1}, {1,-k/2,1}},
      {h, w} // h-by-w image
   // d_i, o_i, s_i for spatial weight
      {/*Nothing*/},
      {{0,0,1}, {/*Nothing*/}},
      {k} // vector of length k
   // More configurations of U()s ...
   Bilateral::Constant{0.1}
);
```

We collect kernels from open-source projects such as Caffe, OpenCV, and Parboil[13, 1, 26], and rewrite them using the UMI operator. Table 1 shows the number of lexical tokens required to express different algorithms. It can be observed that programming with UMI increases the number of literals for defining d_i, s_i, and o_i. However, the number of identifiers and operators consumed by UMI is even less than näive CPU implementations, because programmers only need to describe essence of the kernels without laborously moving the data between memory hierarchies. Operating on identifiers tends to be more error prone than defining separate literals, and therefore programming with UMI can greatly reduce coding efforts.

5.2. Performance Evaluation

Owing to the register pressure caused by the overly aggressive optimization strategy in the compilers, we can only discuss a subset of the kernels here. Details of the problem will be described in 5.3. The performance results are shown in Table 2. First, when compared against separable filter in OpenCV, UMI is slower for shorter convolution lengths owing to the pre-computation overhead, but prevails for longer convolution lengths. It is particularly interesting because, despite the simplistic nature of this kernel, we are

Table 1. Kernel token size comparisons. Note we excluded functional signatures. Identifiers include type name, variable name, and reserved word. Literals include tokens like `1.2f` and `0x11f`.

Kernel	Token type	UMI (ours)	Näive CPU	Tuned GPU
Motion estimation	identifier	49	80	194
	operator	11	45	106
	literal	67	12	38
Bilateral filter [29]	identifier	69	87	122
	operator	22	49	61
	literal	61	7	3
Forward propagation	identifier	53	110	204
	operator	7	65	117
	literal	62	11	14
GEMM	identifier	34	34	146
	operator	7	20	35
	literal	59	4	3
Integral image	identifier	24	23	50
	operator	6	14	22
	literal	42	3	8
Separable filter	identifier	35	50	91
	operator	5	29	47
	literal	37	7	5

Table 2. Speed up ratios of GPU implementions of algorithms. The baseline is OpenCV, Parboil and Caffe, respectively.

Kernels	Note	Speed up
Separable filter	$k = 3$	0.35
	$k = 30$	1.42
Motion estimation		6.51
Forward propagation	UMI $3 + 1s$	19.9
	UMI $9 + 1s$	26.4
	UMI $3 + 2s$	1.80
	UMI $9 + 2s$	2.83
	cuDNN $3 + 1s$	100
	cuDNN $9 + 1s$	109
	cuDNN $3 + 2s$	27.1
	cuDNN $9 + 2s$	27.3

able to extract an extra 30% performance gain. Second, although there remains a large gap between the general-purposed UMI operator and the heavily-tuned cuDNN, the UMI operator still surpasses Caffe which is built on top of the heavily optimized cuBLAS linear algebra library from NVIDIA. Also, the UMI operator holds an edge against Caffe for smaller strides and larger kernels, showing a performance pattern in line with cuDNN.

5.3. Limitations

The UMI operator focuses on abstracting and accelerating regular kernels, therefore it can not be used for data-dependent kernels such as warping and tree-based algorithm. However, targeting such kernels is less effective, since they tend to be too memory-bounded to benefit from the shared memory. For example, NVIDIA's ray tracing library Optix[23] shows no more than 10x acceleration compared to CPU versions, and the warping functions in OpenCV are not accelerated by shared memory.

Second, although the UMI operator defines an effective and general method for using shared memory as read cache for the input tensors, it does not describe how to *write* the result to the shared memory.

In terms of execution efficiency, even though our UMI implementations is faster than most open-source kernels, we are not able to reach comparable performance against vendor-provided libraries such as cuDNN. We inspected the machine code and attributed the gap to excessive common sub-expression elimination (CSE) by the compilers. For example Listing 4 is a GPU kernel performing 1-norm normalization on short vectors. Intuitively this kernel should use about 10 registers for `buf[10]`, but CSE recognizes the same pointers `mem+i` and retains the results between two loops even though it could be cheaper to calculate them again. As a result, a compiled version of this code uses roughly 10 `floats` and pointers, summed up to 30 registers in total, and this is the main reason why we couldn't use larger register tiles in 4.1.

Listing 4. Compilers can optimize incorrectly for the GPUs.

```
float buf[10], one_norm;
for (i = 0; i < 10; ++i)
   one_norm += abs(buf[i] = mem[i]);
for (i = 0; i < 10; ++i)
   mem[i] = buf[i]/one_norm;
```

The XOR-hash for conflict-free execution also introduces a few bit-level instructions, injecting overheads into the innermost loop of kernel. However its implementation is cheap enough [17, 7, 9], and it could possible be merged to existing load instructions.

6. Conclusions

In this paper, we described the UMI operator and how they can be implemented efficiently with the processor arrays in GPUs. We applied the operator to several applications and demonstrated substantial performance gains and code token reduction in most cases compared to open, state-of-the-art implementations.

In the near future, we plan to share our implementation to the open-source community, and explore possibilities for hardware implementations of the UMI in the form of a modification to the shared memory fetch unit. We will also investigate the limitations outlined in the paper, such that this operator can be used to apply to more algorithms in the fields of computer vision, pattern recognition, and scientific computation.

Acknowledgement This work was partially sponsored by the Ministry of Science and Technology of Taiwan under Grants MOST 103-2622-E-002-034, and sponsored by MediaTek Inc., Hsin-chu, Taiwan.

References

[1] D. G. R. Bradski and A. Kaehler. *Learning Opencv, 1st Edition*. O'Reilly Media, Inc., first edition, 2008.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *SIGPLAN Not.*, 49(4):269–284, Feb. 2014.

[3] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[4] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[7] C. Gou, G. K. Kuzmanov, and G. N. Gaydadjiev. Sams: Single-affiliation multiple-stride parallel memory scheme. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, MAW '08, pages 350–368, New York, NY, USA, 2008. ACM.

[8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016.

[9] D. T. Harper and D. A. Linebarger. Conflict-free vector access using a dynamic storage scheme. *IEEE Transactions on Computers*, 40(3):276–283, Mar 1991.

[10] C. Harris and M. Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[11] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[12] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[13] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[15] J. Lai and A. Seznec. Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[16] A. Lavin. maxdnn: An efficient convolution kernel for deep learning with maxwell gpus. *CoRR*, abs/1501.06633, 2015.

[17] Z. Y. Liu and X. B. Li. XOR storage schemes for frequently used data patterns. 25(2):162–173, Mar. 1995.

[18] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[19] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, Mar. 2015.

[20] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *The International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.

[21] H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[22] H. Noh, S. Hong, and B. Han. Learning deconvolution network for semantic segmentation. *CoRR*, abs/1505.04366, 2015.

[23] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[24] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. *SIGARCH Comput. Archit. News*, 41(3):24–35, June 2013.

[25] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013.

[26] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[27] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[28] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[29] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Sixth International Conference on Com-*

puter Vision (IEEE Cat. No.98CH36271), pages 839–846, Jan 1998.

[30] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A GPU performance evaluation. *CoRR*, abs/1412.7580, 2014.

[31] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.