Exploring Algorithmic Design Choices for Low Latency CNN Deployment

Changxin Li
Computer and Data Sciences
Case Western Reserve University
Cleveland, USA
cxl1492@case.edu

Sanmukh Kuppannagari
Computer and Data Sciences
Case Western Reserve University
Cleveland, USA
sanmukh.kuppannagari@case.edu

***Abstract*—Convolutional Neural Networks (CNNs) have demonstrated significant success in advancing image and video processing technologies, significantly outperforming traditional methods in both accuracy and efficiency. However, deploying CNNs effectively across diverse hardware platforms often faces the challenge of latency, which can critically impact real-time processing applications. In this work, we explore algorithmic design choices aimed at reducing latency in CNN deployments. We implement five convolution algorithms using SYCL and integrate them into three popular CNN models: VGG16, Resnet101, and Inception V4. By replacing the standard PyTorch Conv2d function with our SYCL-based implementations, we evaluate the execution time of each convolution layer and the overall model on GPUs. Our extensive experiments benchmark the performance of these algorithms against the baseline implementations of the PyTorch and Pytorch Extension for Intel. The results demonstrate significant improvements in execution time, underscoring the potential of these algorithmic choices for achieving low latency in CNN deployments.**

***Index Terms*—Algorithmic Design Choices; Convolution Algorithms; Vision Models; Performance Portability**

I. Introduction

Convolution based models such as Convolutional Neural Networks (CNN) and recent Vision Transformers [1] and deep learning techniques that use convolution operations to progressively create higher level abstraction for image or video based applications. High impact applications such as facial recognition for security and access control [2], medical image processing [3], automated quality control of wafer design [4], material characterization [5], etc. are just a few examples of the successes of CNNs.

The versatility of Convolution based models can be attributed to the extensive research that has been conducted on developing Convolution algorithms — the most computationally intensive kernel of CNNs, targeting a variety of use cases [6]. For example, IM2COL [7], was developed to represent convolution layers as matrix multiplications to enable utilization of BLAS library and significantly enhance the portability of these models [8]. KN2ROW [9] was developed to reduce the memory requirements of IM2COL while depthwise convolution was developed to reduce both memory and computational requirements making it suitable for edge devices [9]. Scalar matrix multiplication was developed to speed up computations by avoiding matrix multiplications [10].

In DYNAMAP [6], authors demonstrated that the relative performance of convolution algorithms on FPGAs is dependent upon the input feature and kernel dimensions. Thus, to achieve extreme low latency deployment of a convolution based model, exploration of convolution algorithms to select one that obtains the best performance is needed. However, deep learning frameworks such as pytorch are packaged with a standard im2col based convolution implementation [11] and do not allow exploration of the benefits of algorithmic design choices. Thus, there is a need for a framework that can enable users to explore algorithmic design choices to obtain low-latency implementations for various deployment scenarios.

Simultaneously, another visible trend is the proliferation of AI accelerators as increasingly customized accelerators are being developed for various use cases [12]. From high performance data centers to low powered edge devices, GPUs and FPGAs spanning across this entire spectrum are available in the market [12]. Processing In Memory (PIM) based architectures are targeting low latency, low power use cases [13]. Additionally, startups that are developing custom ASIC based AI accelerators such as Cerebras (for high performance datacenters) [14] have proliferated in recent years. So, while libraries such as CuDNN do offer implementations of various convolution algorithms [15], as they are limited to Nvidia GPU platforms, they fail to target a majority of deployment scenarios. Developing a performance portable framework is challenging due to extensive and ever evolving research in convolution algorithms and the requirement of developing platform specific implementations for a vast variety of AI accelerators.

To address these issues, we develop a framework that enables exploration of algorithmic design choices over a variety of accelerator platforms. Specifically, the contributions of this work are:

- We build a high performant, portable library of convolution algorithms using SYCL/oneAPI [16] targeting both Nvidia and Intel GPUs.
- For scalar matrix multiplication and direct convolution algorithms, we explore the impact of parallelization. Furthermore, we develop a decision tree based performance

model to predict optimal parallelism for a platform.
- We conduct extensive experimentation on Nvidia and Intel GPUs to demonstrate the impact of algorithmic choices on model performance. We obtain 1.4x improvement in convolution algorithm execution and 1.3x in end to end model inference latency against pytorch baselines.
- We also develop a performance model that predicts the convolution algorithm that is expected to obtain the best performance for a given model on a given hardware.

II. Background and Related Works

A. Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of Deep Neural Networks that use Convolution Layers as the core technique for feature extraction. Each convolution layer is characterized by the following parameters [17]: Input Feature Dimensions: $H_{in} \times W_{in} \times C_{in}$, where $H_{in} \times W_{in}$ denotes the image sizes and C_{in} denotes the channel width, basically, the number of images; Output Feature Dimensions: $H_{out} \times W_{out} \times C_{out}$, where $H_{out} \times W_{out}$ denote the image sizes and C_{out} denotes the channel width; and Kernel Dimensions: $k \times k \times C_{in} \times C_{out}$. The operation performed by convolution layer can then be represented by the following equation:

$$\begin{aligned} O(i, j, c_{out}) = \\ & \sum_{c_{in}=1}^{C_{in}} \sum_{i=1}^{H_{in}} \sum_{j=1}^{W_{in}} \sum_{k1=1}^{k} \sum_{k2=1}^{k} I(i+k1, j+k2, c_{in}) \\ & * K(k1, k2, c_{in}, c_{out}) \end{aligned} \tag{1}$$

Here I, O, K denote the input feature image, output image, and kernel respectively. In practice, appropriate padding and striding are incorporated as suitable to obtain the output image.

B. Research on Convolution Algorithms

Extensive research has been conducted on the development of algorithms to optimize convolution layers for different objectives. IM2COL [7] was one of the earliest works that modeled the convolution operation as matrix multiplications to enable the use of BLAS libraries optimized for devices. KN2ROW [9] was developed as an improvement upon IM2COL to reduce its memory requirements. DEPTHWISE convolution was developed to reduce the computational complexity of the convolution layer and has been especially effective in edge settings [18]. Other algorithms such as direct convolution [19], scalar matrix multiplication [10], indirect convolution [20], deformable convolution [21], etc. have also been developed for various use cases.

C. Related Works on CNN deployment

A variety of works have focused on accelerating the deployment of CNNs on FPGAs, GPUs, and ASICs. However, most of these work focus on hardware specific optimizations to achieve best performance — execution time, energy-efficiency, etc. They do not focus on exploring design choices provided due to convolution algorithms. Moreover, they are limited to specific devices such as FPGA or GPUs and do not offer portability.

Work such as FPG-AI [22], BODA-RTC [23], Intel Pytorch Extensions are a few works that have focused on the portability aspect of CNN deployment. In FPG-AI, authors developed a performance portable CNN deployment framework for a variety of FPGA devices. They achieved this by developing a template Convolution IP and using design space exploration to customize the template for the target FPGA. In BODA-ETC [23], authors developed a deployment framework that implements the key operations of CNNs using openCL to enable deployment on a variety of GPU platforms. They only support the implementation of the convolution IM2COL algorithm. The limitations with these frameworks are that they achieve limited portability on either FPGA or GPU. Intel Extension for Pytorch [24] is a DNN optimization framework from Intel that currently support deployment of CPU and GPU platforms. We are not aware of this framework supporting FPGAs currently. However, given that the framework leverages SYCL/onAPI [16] as one of the underlying library, future iterations may support deployment on FPGA platforms. Here again, the key limitation is that these frameworks do not allow exploration of algorithmic design choices.

DYNAMAP [6] does offer users to explore algorithmic design choices, however, it is written in verilog and customized for FPGA platforms and cannot be employed on GPUs. In summary, to the best of our knowledge, there does not exist a framework that enables users to explore algorithmic design choices on a variety of a variety of target platforms, thereby, motivating the need for this project.

D. SYCL/oneAPI

SYCL programming language was developed by Khronos group to address the immense heterogeneity that is being exhibited by modern computing platforms due to the integration of accelerators such as FPGA, GPU, and custom AI processors with CPU platforms [16]. Intel offers an implementation of SYCL using its oneAPI frameworks and libraries. Using SYCL/oneAPI, a programmer can write the code once and deploy it on a variety of CPU+X platforms, where X can be any accelerator. While the program can run with a reasonable performance on a variety of platforms, the programmer also has the option to further optimize code for specific platforms without compromising its portability.

III. Performance Portable Convolution Algorithm Library

We developed SYCL based implementations of popular convolution algorithms – IM2COL, KN2ROW, SMM, Direct, and Depthwise Convolution. SYCL enables *portability* by enabling execution of the same source code on a variety of platforms with minimal to no code modifications. To obtain *performance*, we leverage device specific BLAS libraries for IM2COL, KN2ROW. Intel MKL BLAS is used for Intel platforms, while cuBLAS is used for NVIDIA GPU. For SMM, direct, and depthwise convolutions, that do not map to matrix multiplication, we develop parallel algorithms accompanied by a decision tree based performance model to obtain optimal parallelism parameters.

The algorithms implemented in our SYCL library as well as the corresponding implementation strategies are described below.

A. IM2COL

IM2COL [8], short for "image-to-column," is a technique used to optimize convolution operations in deep learning. The method gained prominence in 2014 with the rise of convolutional neural networks (CNNs) for image processing tasks. The central concept behind IM2COL is to transform the convolution operation into a matrix multiplication (GEMM - General Matrix Multiply) operation. This transformation allows leveraging highly optimized BLAS (Basic Linear Algebra Subprograms) libraries, which are designed to perform matrix multiplications very efficiently on modern hardware.

Implementation Strategy: The IM2COL approach reshapes the input image and the filter into column vectors, making the convolution operation equivalent to a matrix multiplication. This is done by:

- Unfolding each receptive field (the region of the input over which the filter is applied) of the input image into a column.
- Stacking these columns side by side to form a matrix.

For an input tensor of shape $(N, C_{\text{in}}, H_{\text{in}}, W_{\text{in}})$, where N is the batch size, the transformation can be visualized as:

$$\text{Input}_{\text{COL}} \rightarrow \text{Matrix of shape } (C_{\text{in}} \cdot H_{\text{in}} \cdot W_{\text{in}}, N \cdot H_{\text{out}} \cdot W_{\text{out}})$$

Here, H_{out} and W_{out} are the height and width of the output feature map.

We utilize Intel's oneAPI Math Kernel Library (oneMKL), which provides device specific highly optimized matrix multiplication routines. For Intel, it leverages oneMKL GEMM kernel. For Nvidia, it leverages Nvidia's cuBLAS kernel.

B. KN2ROW

Kn2row [25] was developed to mitigate the limitations of IM2COL. IM2COL algorithm results in a $k \times k$ increase in the input memory size which results in significant inefficiencies on devices with lower memory or larger kernel sizes. Similar to IM2COL, KN2ROW also relies on converting convolution operations into matrix multiplications. However, its approach differs from IM2COL in how it rearranges the elements of the convolution operation. Instead of focusing on the input image, KN2ROW focuses on the kernel (filter) and transforms it into row vectors.

Implementation Strategy: The KN2ROW data layout process proceeds as follows:

- Reshaping the filter tensor into a matrix where each row corresponds to a flattened version of the filter applied to each output channel.
- Reorganizing the input tensor so that the receptive fields match the rows of the transformed filter matrix.

For a filter tensor of shape $(K, C_{\text{in}}, H_{\text{in}}, W_{\text{in}})$ (where K is the number of output channels), the transformation can be visualized as:

$$\text{Filter}_{\text{ROW}} \rightarrow \text{Matrix of shape } (K, C_{\text{in}} \cdot H_{\text{in}} \cdot W_{\text{in}})$$

The input tensor is rearranged accordingly to form a matrix suitable for multiplication with the transformed filter. KN2ROW results in an expansion in the output memory size by a factor of $k \times k$, but this can be mitigated by accumulating the outputs when they are generated.

Here again, we utilize Intel's oneAPI Math Kernel Library (oneMKL), which provides device specific highly optimized matrix multiplication routines. For Intel, it leverages oneMKL GEMM kernel. For Nvidia, it leverages Nvidia's cuBLAS kernel.

C. Scalar Matrix Multiplication with Zero Packing (SMM)

Scalar Matrix Multiplication with Zero Packing (SMM) [26] is a technique designed in 2022 that intends to optimize convolution operations by avoiding matrix multiplications.

The central concept of SMM is to view the output of the convolution operation as a linear combination of input features with the kernel elements as weights. Using this view, matrix multiplications are replaced using matrix scaling and matrix addition operations. Compared to IM2COL and KN2ROW, SMM does not result in an increase in input or output memory size.

Another avenue of optimization in SMM is obtained by avoiding the zero values of kernels explicitly. By avoid these values, a number of matrix scaling and additions are avoided resulting in observable reduction in execution times.

Implementation Strategy: SMM algorithm loops through the input channels and processes each channel individually. We use `parallel_for` loops provided by SYCL to implement parallelism. `parallel_for` enables us to spawn "work-items", that implement Single Program Multiple Data parallelism. "work-items" can be grouped together into logical blocks of "work-groups". "work-groups" have access to the same shared cache leading to improved data reuse and performance.

In our implementation of SMM, we parallelize along the output channel dimension. For each output channel, an accumulator is used that accumulates the results of processing the input channels C_{in}. The input channels are processed sequentially to avoid write conflicts. For each input channel c_i, the following operations are performed:

- **Extract Sub-Matrices:** Extract k sub-matrices $T_j^{c_i}$ from the input image I, consisting of all the rows and k columns such that $T_j^{c_i} = I[c_i, 1 : H_{in}, j : j+k-1], \forall j \in \{0, 1, \ldots, k-1\}$. Each sub-matrix corresponds to one column of the kernel matrix.
- **Shifting and Multiplication:** For each element of column j of the kernel matrix, appropriately shift $T_j^{c_i}$ vertically to produce $T_{jl}^{c_i} \forall l \in \{0, 1, \ldots, k-1\}$ so that it aligns with the output. This step produces a total of

$k \times k$ sub-matrices, one for each pixel of the kernel. The sub-matrices are padded appropriately to match the output size of $H_{out} \times W_{out}$. Each sub-matrix $T_{jl}^{c_i}$ is scaled by the kernel element at index jl.
- **Accumulation:** The results of the multiplications are accumulated.

Optimizations: We incorporate the following optimizations to obtain high performance in SMM.

- **Kernel Layout for Contiguous Access:** To match the access pattern of the algorithm and ensure contiguous memory accesses, the kernel layout is organized as a multidimensional array of shape (c_i, k_w, k_h, c_o).
- **Sub-matrices Layout for Memory Efficiency:** While we mention above that the algorithm requires creation of $k \times k$ sub-matrices $T_{jl}^{c_i}$, in practice, submatrices $T_{jl}^{c_i} \forall l \in \{0, 1, \ldots, k-1\}$ are sub-sets of $T_j^{c_i}$ and so we only keep a single copy and use appropriate indexing. The steps of scaling of sub-matrices and accumulation to output is combined so that we do not need to store the temporary $k \times k$ scaled sub-matrices.
- **Parallelization:** In addition to output channel parallelization, we parallelize the elementwise scaling and accumulation of the sub-matrices. Specifically, we use p_{smm} threads and each thread computes the scaling and accumulation of a sub-block of each sub-matrices. As noted above, scaling and accumulation steps are combined together to avoid allocation of memory for temporary results. The $k \times k$ sub-matrices are still processed sequentially to avoid write conflicts. To utilize the shared cache provided by modern GPU architectures, the p_{smm} threads are grouped into a single "work-group" and we have C_{out} "work-groups" corresponding to the C_{out} output channels.

Determining Optimal Parallelism p_{smm}: The parallelism parameter p_{smm} can vary from 1 till the number of elements of the output, i.e., $H_{out} \times W_{out}$. The optimal value of p_{smm} is expected to depend upon the number of output channels, kernel size, as well as the device.

To obtain this optimal value, our framework includes a simple decision tree model to predict the optimal p_{smm} value. We trained the decision tree model by performing close to 400 runs by varying the number of output channels, kernel size, image dimensions, and the value of p_{smm} on a targeted hardware. We treated this as a classification problem with number of labels varying from 1 till the product of image dimensions. Using $k-$ fold validation, our decision tree obtained an F1 score of around 0.79 for both platforms, which is significantly higher than random (given the large number of classes). The corresponding R^2 score was around 0.83 for both the platforms, showing a good predictability.

Limitation of Performance Model: Currently, we require running several 100 executions on a target platform to develop a model. We plan to address portability of performance model across platforms with minimal runs in future.

D. Direct Convolution

Direct Convolution [27] is one of the fundamental and straightforward approaches to performing convolution operations in neural networks. Direct Convolution involves applying the convolutional filter directly to the input data. This method has been in use since the inception of convolutional neural networks (CNNs) in the late 1980s and early 1990s, with foundational work by Yann LeCun and others.

The key idea behind Direct Convolution is to apply the convolutional kernel (filter) directly to the input tensor without transforming the data into another format. This means performing the sliding window operation, where the kernel is applied to each spatial location of the input tensor, computing dot products between the kernel weights and the corresponding input region.

Implementation Strategy: Direct convolution simply implements the five loops of Equation 1 for each output channel. Here again we use `parallel_for` loops to parallelize. Similar to SMM, we compute output channels in parallel. Within each output channel, p_{direct} threads are used to compute the output pixels in parallel. Similar to SMM, the value of p_{direct} can vary between 1 and the number of pixels in the output image, i.e., $H_{out} \times W_{out}$. To utilize the shared cache provided by modern GPU architectures, the p_{direct} threads are grouped into a single "work-group" and we have C_{out} "work-groups" corresponding to the C_{out} output channels.

We again train decision tree models to estimate optimal value for p_{direct}. The decision trees obtain F1 and R^2 score of around 0.74 and 0.85 for both the platforms, which again demonstrates high predictibility.

E. Depthwise Separable Convolution (DEPTHWISE)

Depthwise Separable Convolution [28], also known as Depthwise Convolution, was introduced by the Xception [18] architecture in 2016 and later adopted by MobileNets. It reduces computational cost and parameters in CNNs by dividing standard convolution into two steps: depthwise convolution and pointwise convolution. Depthwise Separable Convolution splits the convolution operation into two simpler operations:

- **Depthwise Convolution:** This step performs spatial convolution independently for each input channel. Instead of convolving all channels together with a set of filters, each input channel is convolved with its own filter. This reduces the number of computations as no cross-channel mixing occurs at this stage.
- **Pointwise Convolution:** This step performs a 1x1 convolution to combine the output of the depthwise convolution. This step mixes the information across channels and is equivalent to applying a fully connected layer independently at each spatial location.

Implementation Strategy: Depthwise Convolution requires parallel computation of $C_{in} = C_{out}$ convolutions. We use the following thread hierarchy: C_{out} "work-groups" corresponding to the C_{out} output channels are utilized with each "work-groups" using p_{depth} threads to compute the output pixels in parallel. For pointwise convolution, direct convolution parallelism scheme is adopted. We again train decision tree models to estimate optimal value for p_{depth}. The decision trees obtain F1 and R^2 score of around 0.72 and 0.83, for both the platforms.

Note: Depthwise convolution is not functionally equivalent to the convolution operation defined by Equation 1 as it reduces the number of input-output channels that interact with each other. Thus, simply replacing a convolution operation by depthwise convolution during inference may reduce accuracy. A model needs to be retrained if using depthwise convolution algorithm. However, we demonstrate it in our results due to its popularity.

F. Performance Modeling for Algorithm Selection

We also developed a simple decision tree model to estimate the convolution algorithm that can obtain the best performance. The model takes input dimensions, and number of input, output channels and outputs the convolution algorithm to use. The decision trees obtain F1 score of around 0.744 for both the platforms.

IV. Framework Implementation

Our framework integrates the performance portable SYCL implementations of convolution algorithm into pytorch using cppextensions to make them seamlessly available via the popular python interface. A user can simply set the right flags to select the appropriate algorithm. Moreover, the performance models come packaged with the framework and users can train device specific models by performing some measurements as described above. After the performance models are trained, the framework can be utilized as follows:

Input: The framework takes a pre-trained PyTorch CNN model described in pytorch as input. The convolution layers in these models are implemented using `Conv2d()` function provided by PyTorch.

Output: For each model, a deployment-ready CNN model where each convolution layer is replaced with the optimal convolution algorithm is produced.

The framework is available on github[1].

Note: While we do not perform experiments on FPGA platforms as they were unavailable to us, our framework is still expected to support these platforms as our selected programming language SYCL also has support for FPGAs.

V. Experiments and Results

A. Setup

We evaluate the performance-portable library on three popular CNN models: VGG16 [29], ResNet101 [30], and InceptionV4 [31]. Our experiments span a range of hardware, including the Intel® Iris® Xe MAX GPU and Intel® Xeon® Platinum 8468V CPU as low-power options, and the NVIDIA V100, A100 GPUs, and Intel Max 1100 GPU for high-performance deep learning and HPC tasks.

As described in Section IV, the SYCL algorithms are integrated into pytorch and can be utilized by simply setting a flag. No code change is needed between the Intel and the Nvidia platform.

Baselines: We use the eager execution pytorch (default mode) [11] and Intel extensions of pytorch [24] as baselines. Pytorch, even though it is implemented in python is heavility optimized as it takes the computational graph of a deep neural network model and maps it to device specific BLAS operations (for example, cuDNN on Nvidia). Similarly, Intel extensions for pytorch in an optimized version of pytorch for Intel platforms. Thus, even modest improvements against these baselines is a significant result. Both these frameworks utilize a single convolution algorithm and so the performance improvements presented in the following sections can be attributed to the impact of choosing appropriate convolution algorithm.

B. Evaluation Objectives

Our experimental evaluations are geared towards answering the following questions:

1) How does parallelism impact the performance of SMM and Direct Convolution? (Section V-C)
2) What is the relative performance of convolution algorithms on different models on different hardware? (Sections V-D- V-E)
3) What are the tradeoffs in various convolution algorithms? (Section V-F)

C. Impact of Parallelism on SMM and Direct Convolution performance

Figure 1 shows the change in execution time of convolution for a few selected input and kernel dimensions with respect to the parallelism parameters p_{smm}, p_{direct}, denoted as work-group size in the label. In all four scenarios, we observe that the performance initially improves with increasing number of threads. However, after reaching a minima, it starts reducing again. We make the following observations:

- The improvement in performance from 16 to 32 threads is much more pronounced in SMM compared to direct across all the points. This may imply that direct convolution is bounded by memory even with a lower number of threads. In other words, in our implementations, SMM has a better data reuse than direct.
- For SMM, work-group size of 128 has the best performance on both Intel and Nvidia platform. For Direct, the optimal work-group size is again 128 for most of the points and is 256 for Point 2.

D. Relative Performance of Convolution Algorithms on CNN Model Inference

TABLE I: Relative Performance Table on VGG16

Hardware \ Algorithms	Kn2row	Im2col	Depthwise	Direct	SMM	PyTorch	Intel PyTorch Extension
Intel® Xeon® Platinum 8468V	3.7421	3.6583	3.9217	3.4938	**3.2134**	3.8762	-
Intel® Iris® Xe MAX	0.8742	0.8876	0.8704	0.8875	**0.7614**	1.0983	0.8981
Nvidia V100	0.8724	0.8751	0.8605	0.8701	**0.6832**	*1.0000*	-
Nvidia A100	0.2534	0.2658	0.2656	0.2689	**0.1942**	0.3017	-
Intel Max 1100 GPU	0.2815	0.2928	0.2801	0.3251	**0.2498**	0.3736	-

[1]https://github.com/KLab-AI3/hipc-24

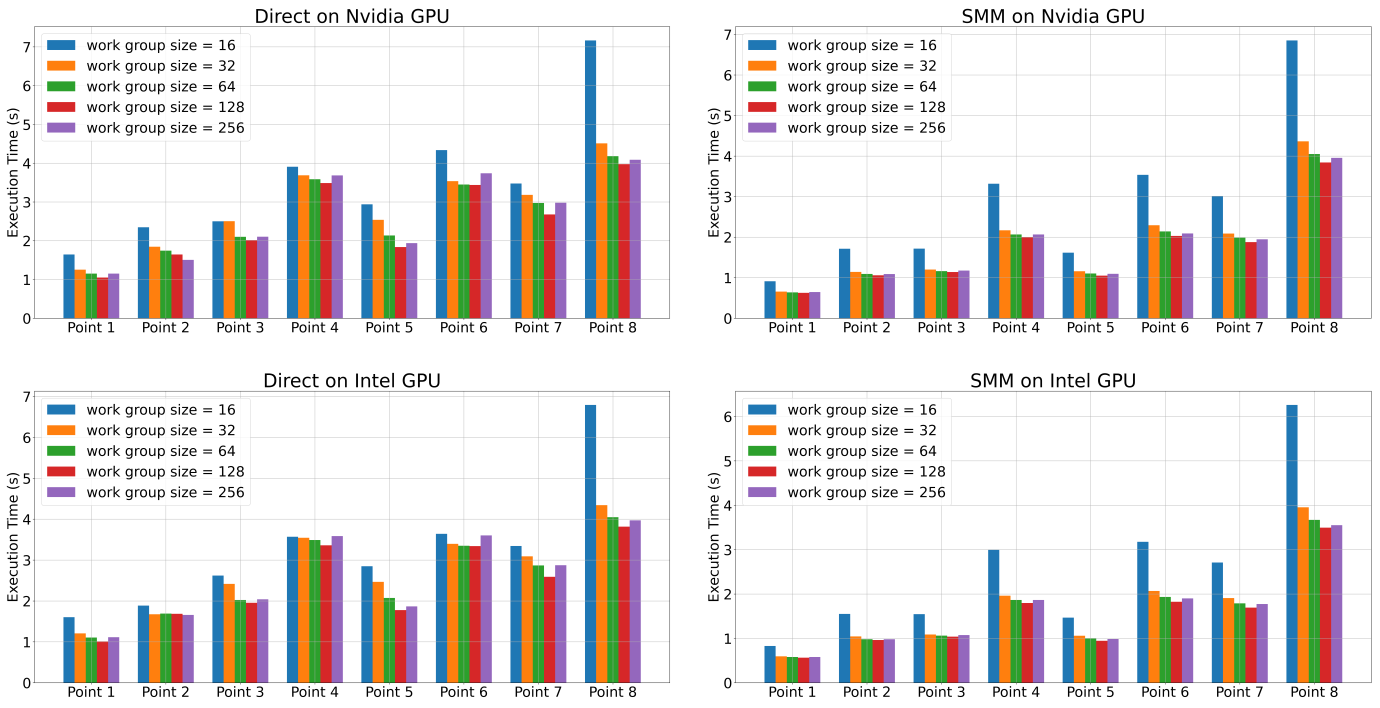

Fig. 1: Performance of Direct and SMM convolution on Nvidia and Intel platforms with varying parallelism. X-axis points: Point 1: 3x3 kernel, 128 input/output channels; Point 2: 5x5 kernel, 128 input/output channels; Point 3: 3x3 kernel, 128 input, 256 output channels; Point 4: 5x5 kernel, 128 input, 256 output channels; Point 5: 3x3 kernel, 256 input, 128 output channels; Point 6: 5x5 kernel, 256 input, 128 output channels; Point 7: 3x3 kernel, 256 input/output channels; Point 8: 5x5 kernel, 256 input/output channels. Image size is fixed at 1024x1024.

In Tables I, II, and III, we present the relative performance of the convolution algorithms compared to the execution time of the convolution layers of the model using PyTorch on NVIDIA V100 as the baseline (shown as 1.0). Lower relative execution times indicate better performance.

The relative performance of algorithms can be determined by the rows, whereas, columns denote relative performance of the same algorithm on the two hardware. However, due to the widely different capabilities of the two platforms, the difference in performance is to be expected.

For VGG16 on the Intel® Xeon® Platinum 8468V platform (Table I), the Kn2row, Im2col, Depthwise, and Direct algorithms all show performance improvements over PyTorch, with improvements of 1.04x, 1.06x, 1.05x, and 1.11x, respectively. The SMM algorithm achieves the best performance on this platform, providing a 1.21x improvement over PyTorch, making it the most efficient algorithm in terms of reducing execution time. For VGG16 on the Intel® Iris® Xe MAX platform (Table I), Kn2row, Im2col, Depthwise, and Direct outperform PyTorch with improvements of 1.26x, 1.24x, 1.26x, and 1.24x, respectively. Once again, the SMM algorithm provides the best performance, with a 1.44x improvement over PyTorch and a 1.18x improvement over Intel PyTorch Extensions. The Intel PyTorch Extensions provide a modest performance boost over PyTorch, but still fall short of the efficiency gains offered by the SMM algorithm and other custom algorithms. For VGG16 on the Nvidia V100 platform (Table I), Kn2row, Im2col, Depthwise, and Direct show performance improvements of 1.15x, 1.14x, 1.16x, and 1.15x, respectively, compared to PyTorch. The SMM algorithm delivers the best performance, achieving a 1.46x improvement over PyTorch, making it the most efficient choice for this platform. On the Nvidia A100 platform (Table I), the SMM algorithm continues to provide the best performance, with a 1.55x improvement over PyTorch. Kn2row, Im2col, Depthwise, and Direct show improvements of 1.19x, 1.13x, 1.13x, and 1.12x, respectively, over PyTorch. On the Intel Max 1100 GPU platform (Table I), the custom algorithms (Kn2row, Im2col, Depthwise, and Direct) demonstrate performance improvements over PyTorch, with improvements of 1.33x, 1.28x, 1.33x, and 1.15x, respectively. The SMM algorithm continues to be the fastest-performing algorithm, with a 1.49x improvement over PyTorch.

TABLE II: Relative Performance Table on Resnet101

Hardware \Algorithms	**Kn2row**	**Im2col**	**Depthwise**	**Direct**	**SMM**	**PyTorch**	**Intel PyTorch Extension**
Intel® Xeon® Platinum 8468V	3.6547	3.7458	4.0274	4.1325	**3.4549**	4.5476	-
Intel® Iris® Xe MAX	0.8073	0.8356	0.7768	0.8570	**0.7368**	1.0286	0.8770
Nvidia V100	0.7828	0.8123	0.7543	0.8332	**0.7139**	*1.0000*	-
Nvidia A100	0.2409	0.2438	0.2314	0.2577	**0.2061**	0.2854	-
Intel Max 1100 GPU	0.2832	0.2863	0.2935	0.3094	**0.2691**	0.3791	-

For Resnet101 on the Intel® Xeon® Platinum 8468V platform (Table II), Kn2row, Im2col, Depthwise, and Direct show performance improvements over PyTorch of 1.24x, 1.21x, 1.13x, and 1.10x, respectively. The best performance is achieved by the SMM algorithm, which provides a 1.32x improvement against PyTorch on this platform. This result highlights the efficiency of SMM on the CPU platform, reducing execution time compared to PyTorch significantly. For Resnet101 on the Intel® Iris® Xe MAX platform (Table II), Intel PyTorch Extensions perform better than PyTorch as expected. However, the SMM algorithm achieves the best performance, with a 1.4x improvement over PyTorch and a 1.19x improvement over Intel PyTorch Extensions. Other algorithms, such as Kn2row, Im2col, Depthwise, and Direct, obtain 1.27x, 1.23x, 1.33x, and 1.20x improvements against PyTorch, respectively, and 1.02x, 1.01x, 1.02x, and 1.02x improvements against Intel PyTorch Extensions. For Resnet101 on the Nvidia V100 platform (Table II), all the algorithms (Kn2row, Im2col, Depthwise, and Direct) outperform PyTorch by achieving improvements of 1.28x, 1.23x, 1.33x, and 1.20x, respectively. The best performance is obtained by the SMM algorithm, with a 1.4x improvement over PyTorch. For Resnet101 on the Nvidia A100 platform (Table II), the performance of the algorithms relative to PyTorch is even better. Kn2row, Im2col, Depthwise, and Direct achieve improvements of 1.18x, 1.17x, 1.23x, and 1.11x, respectively. The SMM algorithm performs the best with a 1.39x improvement over PyTorch. For Resnet101 on the Intel Max 1100 GPU platform (Table II), similar improvements are observed, with Kn2row, Im2col, Depthwise, and Direct showing 1.34x, 1.32x, 1.29x, and 1.22x improvements over PyTorch, respectively. Once again, the SMM algorithm shows the best performance with a 1.41x improvement over PyTorch.

TABLE III: Relative Performance Table on InceptionV4

Hardware \ Algorithms	Kn2row	Im2col	Depthwise	Direct	SMM	PyTorch	Intel PyTorch Extension
Intel® Xeon® Platinum 8468V	3.8723	3.9154	4.1072	4.1026	**3.4358**	4.7251	-
Intel® Iris® Xe MAX	0.8275	0.8080	0.7684	0.8428	**0.7478**	1.0352	0.8695
Nvidia V100	0.8002	0.7812	0.7444	0.8135	**0.7242**	*1.0000*	-
Nvidia A100	0.2171	0.2204	0.2281	0.2472	**0.1914**	0.2750	-
Intel Max 1100 GPU	0.2849	0.2860	0.2905	0.2974	**0.2725**	0.3683	-

For InceptionV4 on the Intel® Xeon® Platinum 8468V platform (Table III), Kn2row, Im2col, Depthwise, and Direct perform better than PyTorch with improvements of 1.22x, 1.21x, 1.15x, and 1.15x, respectively. The SMM algorithm shows the best performance, delivering a 1.38x improvement over PyTorch. This consistent trend highlights SMM's ability to significantly reduce execution time on this CPU platform. On the Intel® Iris® Xe MAX platform (Table III), the Kn2row, Im2col, Depthwise, and Direct algorithms offer improvements of 1.25x, 1.28x, 1.35x, and 1.23x, respectively, over PyTorch. The SMM algorithm provides the best performance on this platform, offering a 1.39x improvement over PyTorch and a 1.16x improvement over Intel PyTorch Extensions. The Intel PyTorch Extensions perform better than PyTorch but are still slower than the custom algorithms, especially SMM. For InceptionV4 on the Nvidia V100 platform (Table III), Kn2row, Im2col, Depthwise, and Direct achieve performance improvements over PyTorch by 1.25x, 1.28x, 1.34x, and 1.23x, respectively. The SMM algorithm shows the best performance on the V100, providing a 1.38x improvement over PyTorch. On the Nvidia A100 platform (Table III), the SMM algorithm continues to outperform all others, achieving the best performance with a 1.44x improvement over PyTorch. Kn2row, Im2col, Depthwise, and Direct also perform better than PyTorch, offering improvements of 1.27x, 1.25x, 1.21x, and 1.11x, respectively. On the Intel Max 1100 GPU platform (Table III), Kn2row, Im2col, Depthwise, and Direct algorithms show performance improvements of 1.29x, 1.29x, 1.27x, and 1.24x, respectively, over PyTorch. The SMM algorithm offers the best performance, with a 1.35x improvement over PyTorch.

E. Relative End to End Performance of Convolution Algorithms

In Tables IV, V, and VI, we present the relative performance of various convolution algorithms, using the execution time of the entire model with PyTorch on the NVIDIA V100 as the baseline (denoted as 1.0). Lower relative execution times correspond to improved performance.

The rows in these tables illustrate the relative performance of the different convolution algorithms, while the columns compare the performance of the same algorithm across two distinct hardware platforms. Given the considerable differences in the capabilities of these platforms, the variations in performance are expected.

TABLE IV: Relative End to End Performance of Convolution Algorithms Table on VGG16

Hardware \Algorithms	Kn2row	Im2col	Depthwise	Direct	SMM	PyTorch	Intel PyTorch Extension
Intel® Xeon® Platinum 8468V	3.7052	3.7618	3.8823	4.0964	**3.4027**	4.5065	-
Intel® Iris® Xe MAX	0.9706	0.9917	0.9685	1.0142	**0.7894**	1.1393	0.9985
NVIDIA V100	0.8549	0.8703	0.8500	0.8897	**0.6932**	*1.0000*	-
Nvidia A100	0.2939	0.3165	0.2964	0.3497	**0.2399**	0.3912	-
Intel Max 1100 GPU	0.3721	0.3842	0.3716	0.4064	**0.3492**	0.4628	-

On the Intel® Xeon® Platinum 8468V platform (Table IV), the algorithms show slower performance compared to the GPU platforms, but they still outperform PyTorch. Kn2row achieves a 1.22x improvement, Im2col achieves a 1.20x improvement, Depthwise achieves a 1.16x improvement, and Direct achieves a 1.10x improvement. The SMM algorithm remains the best performer, with a 1.32x improvement over PyTorch. On the Intel® Iris® Xe MAX platform (Table IV), the Intel PyTorch Extensions perform better than the standard PyTorch implementation. However, the SMM algorithm again demonstrates the best performance, with a 1.44x improvement over PyTorch and a 1.26x improvement over Intel PyTorch Extensions. The other algorithms show the following improvements relative to PyTorch: Kn2row achieves a 1.17x improvement, Im2col achieves a 1.15x improvement, Depthwise achieves a 1.18x improvement, and Direct achieves a 1.12x improvement. Compared to Intel PyTorch Extensions, the improvements are as follows: Kn2row achieves a 1.03x improvement, Im2col achieves a 1.01x improvement, Depthwise achieves a 1.03x improvement and Direct achieves a 1.02x improvement. For VGG16 on the Nvidia V100 platform (Table IV), the algorithms Kn2row, Im2col, Depthwise, and Direct show superior performance compared to PyTorch, with relative execution times being lower. Specifically, Kn2row achieves a 1.17x improvement, Im2col achieves a 1.15x improvement, Depthwise achieves a 1.18x improvement, and Direct achieves a 1.12x improvement. The SMM algorithm stands out with the highest performance gain, achieving a 1.44x improvement over PyTorch. For the Nvidia A100 platform (Table IV), the algorithms perform significantly better than PyTorch, with Kn2row achieving a 1.34x improvement, Im2col achieving a 1.26x improvement, Depthwise achieving a 1.36x improvement, and Direct achieving a 1.26x improvement. The SMM algorithm exhibits the highest improvement, with a 1.63x improvement over PyTorch. On the Intel Max 1100 GPU (Table IV), the performance also improves compared to PyTorch. Kn2row achieves a 1.25x improvement, Im2col achieves a 1.21x improvement, Depthwise achieves a 1.32x improvement, and Direct achieves a 1.23x improvement. The SMM algorithm remains the most efficient, with a 1.43x improvement over PyTorch.

TABLE V: Relative End to End Performance of Convolution Algorithms Table on Resnet101

Hardware \Algorithms	Kn2row	Im2col	Depthwise	Direct	SMM	PyTorch	Intel PyTorch Extension
Intel® Xeon® Platinum 8468V	3.6538	3.7431	4.0292	4.1305	**3.4521**	4.5498	-
Intel® Iris® Xe MAX	0.8045	0.8229	0.7665	0.8577	**0.7376**	1.0312	0.8854
NVIDIA V100	0.7800	0.7979	0.7434	0.8315	**0.7154**	*1.0000*	-
Nvidia A100	0.2416	0.2445	0.2321	0.2584	**0.2073**	0.2849	-
Intel Max 1100 GPU	0.2847	0.2871	0.2942	0.3101	**0.2695**	0.3788	-

On the Intel® Xeon® Platinum 8468V platform (Table V), Kn2row, Im2col, Depthwise, and Direct show slower performance than the GPU platforms, but they still outperform PyTorch. Kn2row achieves a 1.22x improvement, Im2col achieves a 1.18x improvement, Depthwise achieves a 1.14x improvement, and Direct achieves a 1.13x improvement. The SMM algorithm remains the fastest, providing a 1.32x improvement over PyTorch. In examining the performance of Resnet101 on the Nvidia V100 platform (Table V), we observe that the Kn2row, Im2col, Depthwise and Direct algorithms all demonstrate enhanced performance compared to PyTorch. Kn2row achieves a 1.28x improvement, Im2col sees a 1.25x enhancement, Depthwise records a 1.34x improvement, and Direct achieves a 1.20x improvement. In particular, the SMM algorithm outperforms all others, delivering a 1.40x improvement over PyTorch. For the Intel® Iris® Xe MAX platform (Table V), the Intel PyTorch Extensions outperform standard PyTorch performance. However, the SMM algorithm again emerges as the top performer, achieving a 1.40x improvement over PyTorch and a 1.26x enhancement over the Intel PyTorch Extensions. Other algorithms also show significant improvements: Kn2row achieves a 1.28x improvement, Im2col records a 1.25x enhancement, Depthwise sees a 1.34x improvement, and Direct achieves a 1.20x enhancement compared to PyTorch. When compared to Intel PyTorch Extensions, Kn2row improves by 1.03x, Im2col by 1.02x, Depthwise by 1.05x, and Direct by 1.02x. On the Nvidia A100 platform (Table V), Kn2row achieves a 1.30x improvement, Im2col achieves a 1.29x improvement, Depthwise achieves a 1.28x improvement, and Direct achieves a 1.23x improvement. The SMM algorithm continues to stand out, with a 1.48x improvement over PyTorch. For the Intel Max 1100 GPU (Table V), Kn2row achieves a 1.24x improvement, Im2col a 1.23x improvement, Depthwise a 1.29x improvement, and Direct a 1.21x improvement. The SMM algorithm remains the most efficient, delivering a 1.41x improvement over PyTorch.

TABLE VI: Relative End to End Performance of Convolution Algorithms Table on InceptionV4

Hardware \Algorithms	Kn2row	Im2col	Depthwise	Direct	SMM	PyTorch	Intel PyTorch Extension
Intel® Xeon® Platinum 8468V	3.8731	3.9162	4.1065	4.1032	**3.4375**	4.7268	-
Intel® Iris® Xe MAX	0.8608	0.8736	0.7715	0.8379	**0.7060**	1.0351	0.8724
NVIDIA V100	0.8315	0.8439	0.7453	0.8094	**0.6822**	*1.0000*	-
Nvidia A100	0.2178	0.2209	0.2294	0.2468	**0.1919**	0.2758	-
Intel Max 1100 GPU	0.2855	0.2864	0.2909	0.2983	**0.2734**	0.3689	-

On the Intel® Xeon® Platinum 8468V platform (Table VI), Kn2row, Im2col, Depthwise, and Direct demonstrate better performance than PyTorch, with Kn2row achieving a 1.21x improvement, Im2col a 1.19x improvement, Depthwise a 1.14x improvement, and Direct a 1.13x improvement. The SMM algorithm remains the best performer, with a 1.33x improvement over PyTorch. For InceptionV4 on the Nvidia V100 platform (Table VI), the algorithms Kn2row, Im2col, Depthwise, and Direct all show enhanced performance compared to PyTorch, with lower relative execution times. Specifically, Kn2row achieves a 1.20x improvement, Im2col achieves a 1.19x improvement, Depthwise achieves a 1.34x improvement, and Direct achieves a 1.24x improvement. The SMM algorithm stands out as the best performer, delivering a 1.47x improvement over PyTorch. On the Intel® Iris® Xe MAX platform (Table VI), Intel PyTorch Extensions show better performance than the standard PyTorch implementation. However, the SMM algorithm again demonstrates the highest performance, with a 1.47x improvement over PyTorch and a 1.24x improvement over Intel PyTorch Extensions. Other algorithms also exhibit notable enhancements: Kn2row achieves a 1.20x improvement, Im2col achieves a 1.18x improvement, Depthwise achieves a 1.34x improvement, and Direct achieves a 1.24x improvement compared to PyTorch. Relative to Intel PyTorch Extensions, the improvements are as follows: Kn2row improves by 1.01x, Im2col improves by 1.01x, Depthwise improves by 1.13x, and Direct improves by 1.04x. On the Nvidia A100 platform (Table VI), the algorithms show significantly better performance, with Kn2row achieving a 1.31x improvement, Im2col a 1.29x improvement, Depthwise a 1.35x improvement, and Direct achieving a 1.26x improvement. The SMM algorithm stands out with a 1.55x improvement over PyTorch. For the Intel Max 1100 GPU (Table VI), Kn2row achieves a 1.24x improvement, Im2col a 1.23x improvement, Depthwise a 1.29x improvement, and Direct achieves a 1.21x improvement. The SMM algorithm remains the highest performer, with a 1.42x improvement over PyTorch.

F. Tradeoffs in Convolution Algorithms

In this section, we evaluate the relative performance of the convolution algorithms by varying the image and kernel dimensions to evaluate under what conditions one algorithm performs better than the other.

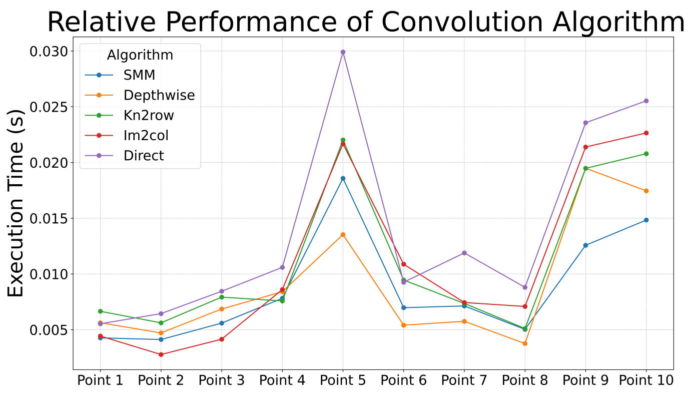

Fig. 2: Relative Performance of Algorithms for Varying Convolution Sizes.

Points represent configurations as follows:
Point 1: 50x50 image, 5x5 kernel, 64 input/output channels;
Point 2: 50x50 image, 5x5 kernel, 128 input, 4 output channels;
Point 3: 100x100 image, 5x5 kernel, 32 input, 64 output channels;
Point 4: 200x200 image, 3x3 kernel, 32 input/output channels;
Point 5: 200x200 image, 5x5 kernel, 16 input, 128 output channels;
Point 6: 250x250 image, 3x3 kernel, 32 input, 8 output channels;
Point 7: 250x250 image, 5x5 kernel, 2 input, 32 output channels;
Point 8: 250x250 image, 5x5 kernel, 16 input, 2 output channels;
Point 9: 400x400 image, 3x3 kernel, 64 input, 8 output channels;
Point 10: 500x500 image, 5x5 kernel, 1 input, 32 output channels.

Figure 2 illustrates the relative performance of five convolution algorithms: SMM, Depthwise, Kn2row, Im2col, and Direct across ten distinct convolution dimensions. Each dimension is defined by a combination of Input Size (Im), Kernel Size (Kn), Input Channels (Ic), and Output Channels (Oc). Analyzing these algorithms under various parameters is crucial for optimizing convolution operations in deep learning models. At point 2 (Im:50, Kn:5, Ic:128, Oc:4), Im2col shows the best execution time, indicating its efficiency with high-dimensional input data and smaller kernel sizes, while Direct exhibits significantly higher execution time, highlighting its inefficiency for such configurations. At point 5 (Im:200, Kn:5, Ic:16, Oc:128), SMM and Depthwise outperform others, with Depthwise particularly effective in handling moderate input sizes with high output channels, whereas Direct again shows poor performance. Point 6 (Im:250, Kn:3, Ic:32, Oc:8) sees SMM delivering the best performance, whereas Direct is the least efficient, suggesting SMM's adaptability to moderate input and output channels. At point 9 (Im:400, Kn:3, Ic:64, Oc:8), SMM excels with the highest input channels, reflecting its suitability for high-depth inputs, while Im2col and Direct lag behind, emphasizing their inefficiency under these conditions.

Based on the experiments we draw the following general conclusions for tradeoffs between the various convolution algorithms:

- The optimal algorithm is highly dependent upon the specific convolution algorithm. As discussed in Section V-F, even a non-linear decision tree model is not able to predict the best algorithm with near perfect accuracy.
- For smaller image dimensions, IM2COL performs the best (Points 1-3), but its performance worsens faster compared to other algorithms for larger image dimensions (Point 9-10).
- For majority of points (Points 4-10), SMM achieves the best execution time if we disregard Depthwise (as the model will need to be retrained to use it).
- Among the two matrix multiplication based algorithms, im2col performs better for lower image dimensions (Points 1-3) whereas kn2row performs better for higher image dimensions (Points 4-10).
- In general, the performance of direct convolution is the lowest.

These analyses underscore the importance of algorithm selection based on convolution parameters to enhance efficiency in deep learning frameworks, guiding practitioners towards more effective and efficient model deployments.

G. Relative Performance with ILSVRC-2012 Dataset

The ILSVRC-2012 (ImageNet) [32] dataset is a large-scale benchmark for image classification and object detection, containing 1.2 million training images, 50,000 validation images, and 100,000 test images across 1,000 categories. Its comprehensive labeling has made it foundational in advancing CNN and deep learning research.

TABLE VII: Relative Performance Table on VGG16 (ILSVRC-2012)

Hardware \ Algorithms	Kn2row	Im2col	Depthwise	Direct	SMM	PyTorch	Intel PyTorch Extension
Intel® Xeon® Platinum 8468V	3.7032	3.6289	3.8717	3.4538	**3.1632**	3.8258	-
Intel® Iris® Xe MAX	0.8513	0.8676	0.8614	0.8772	**0.7519**	1.0787	0.8778
Nvidia V100	0.8627	0.8651	0.8505	0.8606	**0.6728**	*1.0000*	-
Nvidia A100	0.2423	0.2558	0.2556	0.2584	**0.1851**	0.2909	-
Intel Max 1100 GPU	0.2719	0.2828	0.2702	0.3156	**0.2403**	0.3642	-

For VGG16 on the Intel® Xeon® Platinum 8468V platform (Table VII), all algorithms outperform PyTorch. Kn2row achieves a 1.03x improvement, Im2col 1.05x, Depthwise 1.00x, and Direct 1.11x. SMM stands out with a 1.21x improvement over PyTorch. On the Intel® Iris® Xe MAX platform, Intel PyTorch Extensions improve upon standard PyTorch, while SMM again performs best, achieving 1.43x over PyTorch and 1.17x over Intel PyTorch Extensions. Other algorithms offer 1.27x (Kn2row), 1.24x (Im2col), 1.25x (Depthwise), and 1.23x (Direct) improvements over PyTorch. For the Nvidia V100 platform, Kn2row, Im2col, Depthwise, and Direct show improvements of 1.16x, 1.15x, 1.18x, and 1.16x, respectively, over PyTorch. SMM is again the top performer, with a 1.49x improvement. On the Nvidia A100 platform, Kn2row, Im2col, Depthwise, and Direct achieve 1.20x, 1.14x, 1.14x, and 1.13x improvements, respectively, over PyTorch. SMM leads with a 1.57x improvement. On the Intel Max 1100 GPU, all algorithms show gains: Kn2row achieves 1.34x, Im2col 1.29x, Depthwise 1.35x, and Direct 1.15x over PyTorch, while SMM remains the top performer with a 1.51x improvement.

For Resnet101 on the Intel® Xeon® Platinum 8468V platform (Table VIII), Kn2row, Im2col, Depthwise, and Direct outperform PyTorch by 1.25x, 1.22x, 1.13x, and 1.10x, respectively, with SMM achieving a 1.32x improvement. On the Intel® Iris® Xe MAX platform, Intel PyTorch Extensions

TABLE VIII: Relative Performance Table on Resnet101 (ILSVRC-2012)

Hardware \Algorithms	**Kn2row**	**Im2col**	**Depthwise**	**Direct**	**SMM**	**PyTorch**	**Intel PyTorch Extension**
Intel® Xeon® Platinum 8468V	3.6051	3.6962	3.9772	4.0823	**3.4051**	4.5016	-
Intel® Iris® Xe MAX	0.7862	0.8161	0.7564	0.8366	**0.7271**	1.0083	0.8672
Nvidia V100	0.7731	0.8027	0.7445	0.8227	**0.7043**	*1.0000*	-
Nvidia A100	0.2312	0.2337	0.2209	0.2479	**0.1965**	0.2749	-
Intel Max 1100 GPU	0.2728	0.2768	0.2835	0.3014	**0.2603**	0.3697	-

improve upon PyTorch, but SMM performs best with a 1.39x improvement over PyTorch and 1.19x over Extensions. Other algorithms yield 1.28x, 1.23x, 1.33x, and 1.21x gains. For Nvidia V100, Kn2row, Im2col, Depthwise, and Direct improve upon PyTorch by 1.29x, 1.25x, 1.34x, and 1.22x, respectively, while SMM achieves a 1.42x gain. On Nvidia A100, Kn2row, Im2col, Depthwise, and Direct show 1.29x, 1.27x, 1.31x, and 1.11x improvements, with SMM reaching 1.51x. On Intel Max 1100 GPU, all algorithms improve over PyTorch, with Kn2row, Im2col, Depthwise, and Direct achieving 1.35x, 1.33x, 1.30x, and 1.22x, while SMM leads with 1.42x.

TABLE IX: Relative Performance Table on InceptionV4 (ILSVRC-2012)

Hardware \ Algorithms	**Kn2row**	**Im2col**	**Depthwise**	**Direct**	**SMM**	**PyTorch**	**Intel PyTorch Extension**
Intel® Xeon® Platinum 8468V	3.8226	3.8658	4.0569	4.0531	**3.3855**	4.6761	-
Intel® Iris® Xe MAX	0.8081	0.7874	0.7482	0.8331	**0.7382**	1.0149	0.8602
Nvidia V100	0.7903	0.7709	0.7341	0.8038	**0.7143**	*1.0000*	-
Nvidia A100	0.2067	0.2108	0.2179	0.2374	**0.1816**	0.2642	-
Intel Max 1100 GPU	0.2742	0.2763	0.2804	0.2877	**0.2631**	0.3572	-

For InceptionV4 on the Intel® Xeon® Platinum 8468V platform (Table IX), Kn2row, Im2col, Depthwise, and Direct show improvements over PyTorch by 1.23x, 1.22x, 1.15x, and 1.16x, respectively, with SMM achieving 1.38x. On the Intel® Iris® Xe MAX platform, Kn2row, Im2col, Depthwise, and Direct improve upon PyTorch by 1.28x, 1.29x, 1.38x, and 1.24x, respectively. SMM performs best with 1.39x over PyTorch and 1.18x over Intel PyTorch Extensions. For Nvidia V100, SMM achieves a 1.40x improvement over PyTorch, with Kn2row, Im2col, Depthwise, and Direct showing gains of 1.27x, 1.29x, 1.36x, and 1.24x. On Nvidia A100, SMM again leads with 1.45x over PyTorch, while Kn2row, Im2col, Depthwise, and Direct yield 1.30x, 1.25x, 1.20x, and 1.12x improvements. On Intel Max 1100 GPU, SMM reaches the highest performance with 1.43x over PyTorch, while Kn2row, Im2col, Depthwise, and Direct show gains of 1.35x, 1.33x, 1.32x, and 1.22x.

VI. Future Directions

Based on the results of our current study, we have identified several directions for future work to further enhance the efficiency and applicability of our SYCL-based convolution algorithms in Convolutional Neural Networks (CNNs).

Adding Support for Grouped Convolution: One immediate extension is to incorporate support for grouped convolutions. Grouped convolutions, which divide the input and output channels into smaller groups and perform convolutions independently on each group, have proven to be effective in reducing computation while maintaining accuracy. By integrating grouped convolutions into our framework, we can potentially achieve even greater reductions in latency, particularly for models like ResNeXt [33] and EfficientNet [34] that heavily utilize this technique.

Enhancing Accuracy in Performance Modeling: While our decision tree-based performance model has shown promising results in predicting optimal parallelism parameters, there remains room for improvement in accuracy. Future efforts will focus on refining the model to better capture the complex interactions between various hardware characteristics and convolution operations. Incorporating more sophisticated machine learning techniques, such as ensemble learning or neural networks, may provide more precise predictions and further optimize performance.

Supporting Advanced Convolution Techniques: Extending our framework to support advanced convolution techniques such as Fourier [35], Winograd [36], and deformable convolutions [21] is another important direction. Fourier and Winograd convolutions offer substantial computational savings for specific kernel sizes and input dimensions, while deformable convolutions provide enhanced modeling flexibility for tasks requiring precise spatial transformations. Implementing these techniques within our SYCL-based framework will broaden its applicability and effectiveness for a wider range of CNN architectures and applications.

Acknowledgements

This work has been supported by the U.S. National Science Foundation under award numbers 2411447 and 2425535. This work also made use of the High Performance Computing Resource in the Core Facility for Advanced Research Computing at Case Western Reserve University.

VII. Conclusion

In this work, we explored low-latency CNN deployment through SYCL-based implementations of five convolution algorithms integrated into VGG16, Resnet101, and Inception V4. By replacing the standard PyTorch Conv2d function, we evaluated layer and model execution times on multiple GPUs.

Experiments showed that algorithm choice critically affects performance across hardware. SMM consistently achieved the lowest execution times on Nvidia V100, Intel® Iris® Xe MAX, and demonstrated strong results on Nvidia A100 and Intel Max 1100 GPUs, underscoring its suitability for low-latency deployments.

Other algorithms, such as Kn2row and Im2col, showed notable improvements on Nvidia V100 and competitive results on GPU systems. Our decision tree-based model effectively predicted optimal parallelism parameters, further boosting SMM and Direct Convolution performance with minimal manual tuning. This result underscores the utility of machine learning-based modeling for identifying optimal configurations and will be explored in future.

This study highlights the value of algorithm selection and parallelism for efficient CNN deployments on diverse hardware. Future work will extend the framework to more platforms and refine our performance models, enabling more efficient, cross-device CNN optimization.

References

[1] Wenhai Wang, Jifeng Dai, Zhe Chen, Zhenhang Huang, Zhiqi Li, Xizhou Zhu, Xiaowei Hu, Tong Lu, Lewei Lu, Hongsheng Li, et al. Internimage: Exploring large-scale vision foundation models with deformable convolutions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14408–14419, 2023.

[2] Hamidov Oybek Ikromovich and Babakulov Bekzod Mamatkulovich. Facial recognition using transfer learning in the deep cnn. *Open Access Repository*, 4(3):502–507, 2023.

[3] Syed Muhammad Anwar, Muhammad Majid, Adnan Qayyum, Muhammad Awais, Majdi Alnowami, and Muhammad Khurram Khan. Medical image analysis using convolutional neural networks: a review. *Journal of medical systems*, 42:1–13, 2018.

[4] Takeshi Nakazawa and Deepak V Kulkarni. Wafer map defect pattern classification and image retrieval using convolutional neural network. *IEEE Transactions on Semiconductor Manufacturing*, 31(2):309–314, 2018.

[5] Nathaniel Tomczak and Sanmukh Kuppannagari. Automated indexing of tem diffraction patterns using machine learning. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2023.

[6] Yuan Meng, Sanmukh Kuppannagari, Rajgopal Kannan, and Viktor Prasanna. Dynamap: Dynamic algorithm mapping framework for low latency cnn inference. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 183–193, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan M. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *ArXiv*, abs/1410.0759, 2014.

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[9] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. High-performance low-memory lowering: Gemm-based algorithms for dnn convolution. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 99–106, 2020.

[10] Ofir Amir and Gil Ben-Artzi. Smm-conv: Scalar matrix multiplication with zero packing for accelerated convolution. *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 3066–3074, 2022.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[12] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Lincoln ai computing survey (laics) update. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2023.

[13] Gian Singh, Ankit Wagle, Sunil Khatri, and Sarma Vrudhula. Cidan-xe: Computing in dram with artificial neurons. *Frontiers in Electronics*, 3:834146, 2022.

[14] Mandy La and Andrew Chien. Cerebras systems: Journey to the wafer-scale engine. *University of Chicago, Tech. Rep*, 2020.

[15] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan M. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *ArXiv*, abs/1410.0759, 2014.

[16] Beau Johnston, Jeffrey S. Vetter, and Josh Milthorpe. Evaluating the performance and portability of contemporary sycl implementations. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 45–56, 2020.

[17] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[18] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[19] Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. In *International Conference on Machine Learning*, pages 5776–5785. PMLR, 2018.

[20] Marat Dukhan. The indirect convolution algorithm. *arXiv preprint arXiv:1907.02129*, 2019.

[21] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 764–773, 2017.

[22] Tommaso Pacini, Emilio Rapuano, and Luca Fanucci. Fpg-ai: A technology-independent framework for the automation of cnn deployment on fpgas. *IEEE Access*, 11:32759–32775, 2023.

[23] Matthew W Moskewicz, Forrest N Iandola, and Kurt Keutzer. Boda-rtc: Productive generation of portable, efficient code for convolutional neural networks on mobile computing platforms. In *2016 IEEE 12th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–10. IEEE, 2016.

[24] Intel. Pytorch optimizations from intel. https://www.intel.com/content/www/us/en/developer/tools/oneapi/optimization-for-pytorch.html, 2023. Accessed:2024-01-28.

[25] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. Low-memory gemm-based convolution algorithms for deep neural networks. *arXiv preprint arXiv:1709.03395*, 2017.

[26] Amir Ofir and Gil Ben-Artzi. Smm-conv: Scalar matrix multiplication with zero packing for accelerated convolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3067–3075, 2022.

[27] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[28] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[29] Sheldon Mascarenhas and Mukul Agarwal. A comparison between vgg16, vgg19 and resnet50 architecture frameworks for image classification. In *2021 International conference on disruptive technologies for multi-disciplinary research and applications (CENTCON)*, volume 1, pages 96–99. IEEE, 2021.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[33] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[34] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[35] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851*, 2013.

[36] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016.