Description

GPU Implementation of Convolution Operation

March 6, 2023

1 Introduction

Analysis is provided for different optimizations attempted to improve the performance of the convolution operation performed on a GPU. For the complete project, a linear/one dimensional view is adopted for the block and grid dimensions. Each thread within the thread grid is mapped to a specific output element in the output array. The *d_output* array has 4 dimensions. Each of the indices within these 4 dimensions are mapped to the thread grid in a hierarchical manner. Since, the nested for loop contains 7 levels, mapping 4 of the indices within the loop corresponding to each output element brings the number of levels down to 3 in the parallel version. Hence, a 3 level nested for loop is run sequentially by each individual thread in a grid.

The nested for loop executed sequentially by each thread contains the r, s and the c indices. Since, the values of R, S and C are usually very small, sequential overhead is very limited for most problem size configurations. As explained in later subsections, for the case when $C = 832$, loop unrolling is performed to achieve maximum benefit.

For each one of the developed kernels, the indices q, p, k and n are mapped to the thread grid. For the submitted kernels, q varies fastest in the grid followed by p, then k and n. This permutation was found to achieve reasonably high performance. Although reorderings were attempted, these were only executed to analyze performance and the corresponding kernels have not been submitted.

A **general** kernel was developed which incorporated a switching mechanism based on the variation of problem size parameters. For the rest of the article, below notation shall be used to refer to individual problem size variations.

- **conf1**: ./cnn-gpu 1 3 64 112 112 3 3 2 2
- **conf2**: ./cnn-gpu 1 832 128 7 7 1 1 1 1
- **conf3**: ./cnn-gpu 128 3 64 112 112 3 3 2 2
- **conf4**: ./cnn-gpu 128 832 128 7 7 1 1 1 1

Since, the general kernel incorporates different levels of loop unrolling for different problem size variations, different values of grid size were initialized within the **cnn.assign.cu** file

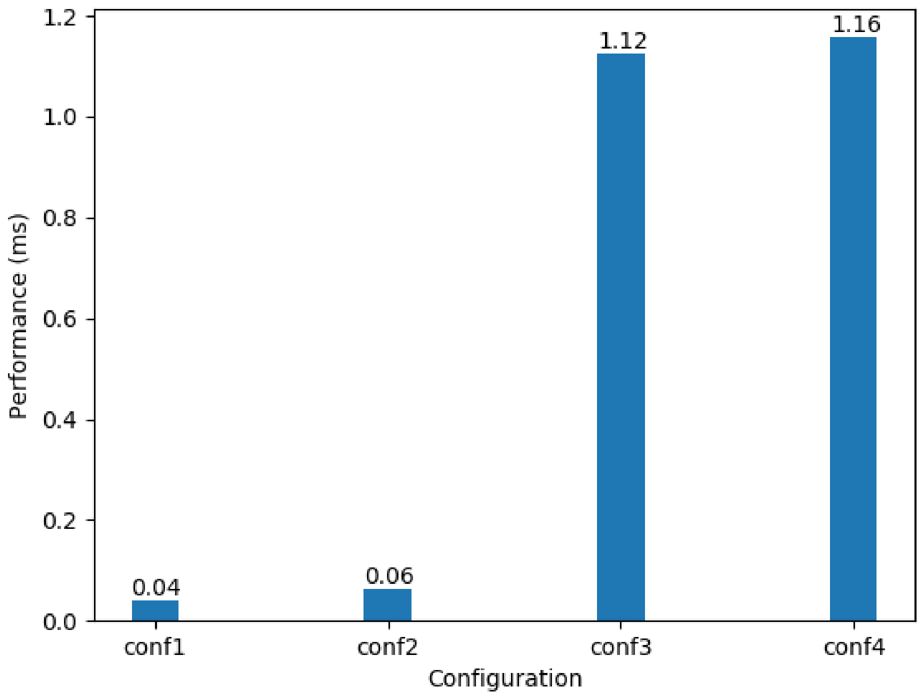


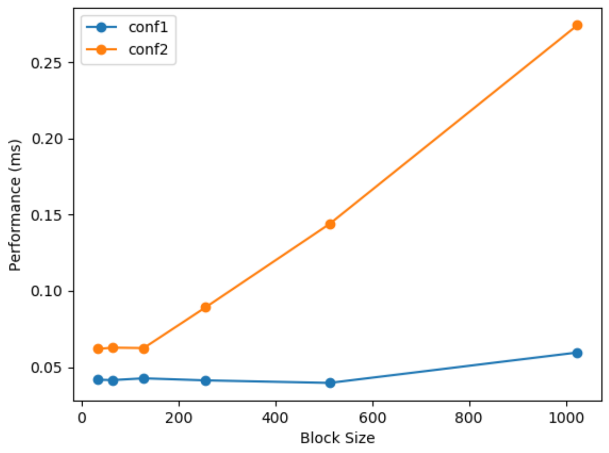
Figure 1: Performance Trends for the General Kernel for different Problem Size Configurations

based on the problem size parameters. Hence, this file is also submitted with the developed kernel. Since, the switching mechanism calls for a nested if statement within the general kernel, individual kernel files corresponding to the optimizations are also submitted. Please note that the individual kernel files may only work correctly for a specific problem size variation and provide high performance for this configuration but may not work correctly for all problems. Hence, for correct execution on all 4 problems, the general kernel must be used.

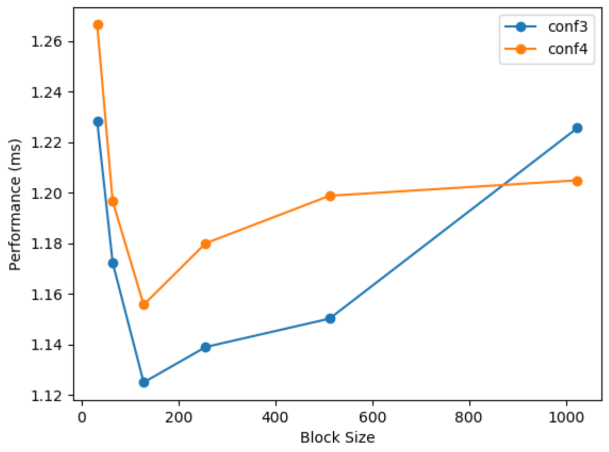
Figure 1 shows the performance obtained when the general kernel was run on the 4 problems. The general kernel performs better than the $cuDNN$ version for $conf1$ and $conf2$ configurations while it approaches the same performance as the $cuDNN$ version for the $conf3$ problem size variation. All performance data within the plots presented in various subsections is reported in $milliseconds$. The performance data was gathered on the $kp359$ node.

2 Blocksize Considerations

Attempts were made to identify a value of blocksize which provided the most optimal performance for different problem size parameters. Due to the incorporation of various optimizations, it was not possible to analytically reason for a specific value and hence, heavy



(a) $N = 1$



(b) $N = 128$

Figure 2: Performance Variation with *Blocksize* for the General Kernel

benchmarking was performed.

Figure 2 displays the performance variation of the general kernel when blocksize varies from 32 until 1024. Due to scale differences across timing results for the configurations where $N = 1$ and $N = 128$, separate plots were created for them. For results where $N = 1$, performance deterioration is observed with increases in the Blocksize parameter. For these configurations, maximum performance is observed at values around $Blocksize = 64$ or $Blocksize = 128$ after which performance either deteriorates rapidly or stagnates. For results where $N = 128$, steep performance improvements are observed until $Blocksize = 128$ after which performance deteriorates. In summary, across all four configurations, $Blocksize = 128$ was identified as the globally optimal blocksize value.

3 Effects of Individual Loop Unrolling

Loop unrolling was found to positively affect the performance for most configurations. However, it was observed that different problem size parameters called for different unroll configurations. The general kernel recognizes this observation by switching between different code versions depending on problem size parameters.

In summary, simultaneous n and k loop unrolling is performed when $N = 128$, only k loop unrolling when $N = 1$ and $C < 256$ and simultaneous unrolling of k and c loops when $C \geq 256$.

3.1 K Loop Unrolling

Individual k loop unrolling showed different performance trends for different problem size parameters. For cases where $N = 128$, performance improvements were observed while steep degradation/stagnation was observed for cases where $N = 1$. Figure 3 shows the performance trends of the general kernel with varying k loop unroll factors. These results were obtained at $Blocksize = 512$ but the trends were seen to be valid for other values of blocksize as well.

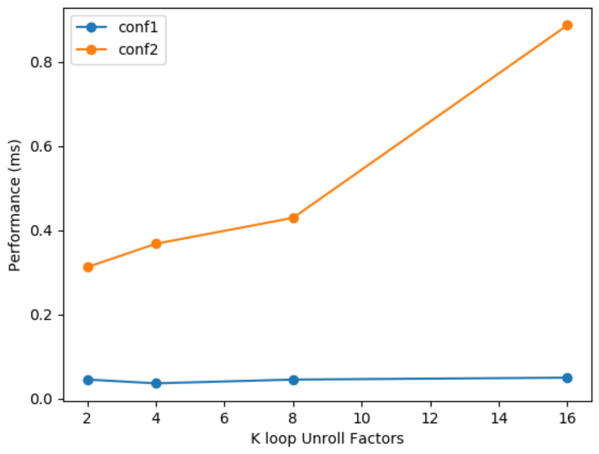
It must be observed that for cases where loop density is low, additional register utilization as a result of increased loop unrolling has the potential to reduce the number of simultaneously executing blocks on a streaming multiprocessor. Although this claim is not verified, but increased block resource utilization can cause lower performance for cases where loop density is not high.

3.2 C Loop Unrolling

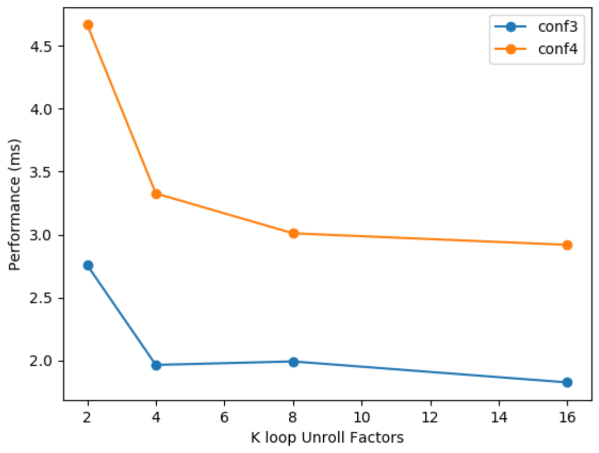
Individual c loop unrolling was not found to have a positive impact on kernel performance. Figure 4 shows the performance variation of the general kernel with different c loop unroll factors. It is observed that for most problem size parameters (not including $conf2$), performance without unrolling remained higher than the observed performance with any degree of c loop unrolling. It must also be noted that for cases where degradation/stagnation is observed, $C = 3$ and hence, loop unrolling might not lead to any benefits. For $conf2$ where initial performance improvement is observed, $C = 832$, validating the effectiveness of loop unrolling for cases where loop density is high.

4 Loop Permutation

As stated earlier, within each one of the developed kernels, every thread executes a three level nested for loop involving c, r and s iterations. It is possible to permute the order of these iterations for which comparisons are provided. Furthermore, it is also possible to permute the order in which indexes are mapped to the thread grid. Performance comparisons are stated for both of these observations.

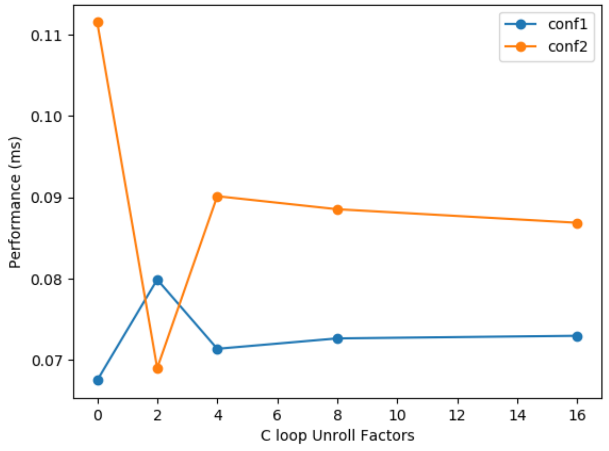


(a) $N = 1$

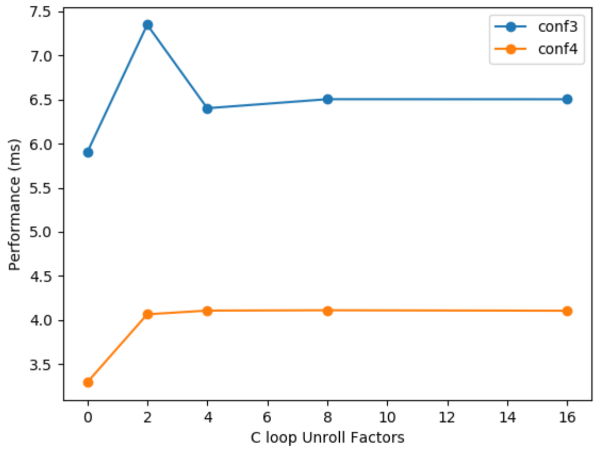


(b) $N = 128$

Figure 3: Performance Variation with Different K Loop Unroll Factors



(a) $N = 1$



(b) $N = 128$

Figure 4: Performance Variation with Different C Loop Unroll Factors

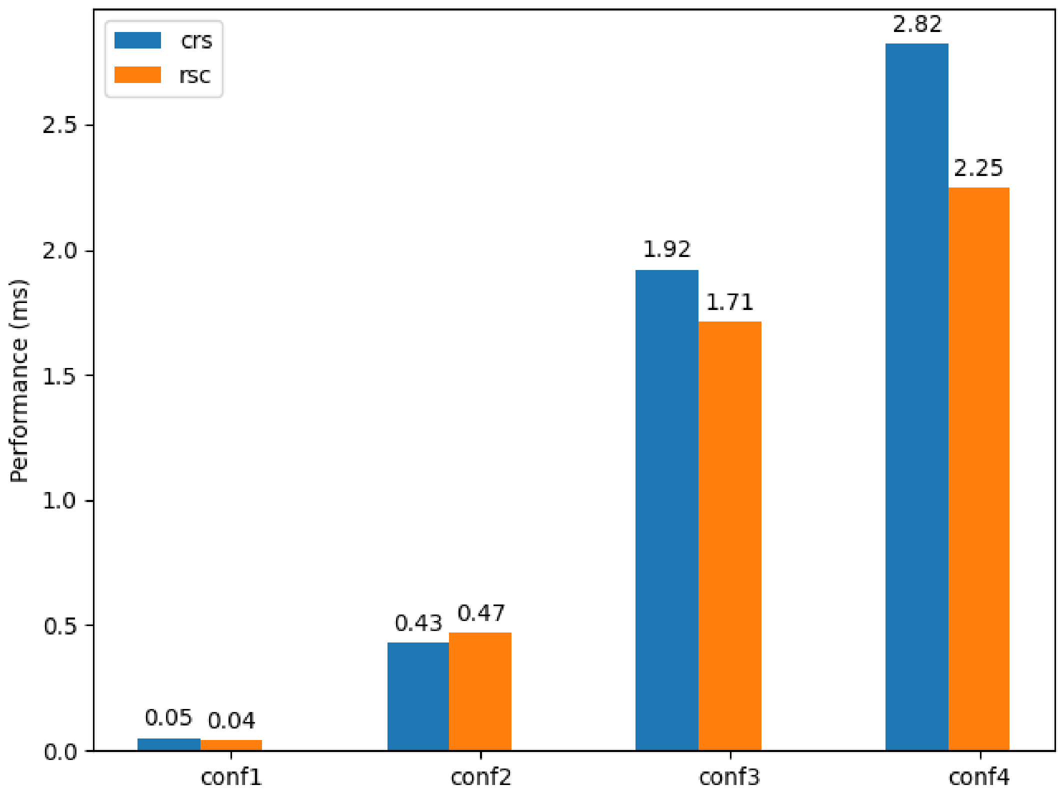


Figure 5: Performance Variations observed on performing Inner Loop Permutation (*CRS vs RSC* Loop Permutation)

4.1 CRS Vs RSC Loop Permutation

It was found that for most problem size configurations, the *rsc* loop permutation provided higher performance as compared to the *crs* permutation. Figure 5 shows performance comparisons between both of these permutations. Since, the *rsc* loop is executed sequentially by each thread, it was counterintuitive to make this observation due to high global memory access stride with respect to the c index.

When the *crs* permutation was combined with k loop unrolling, it was possible to achieve unroll factors as high as 16 without any issues. However, with the *rsc* permutation, unroll factors greater than 8 led to register allocation issues (more register allocation per thread than permitted). Even though high unroll factors could not be achieved with this permutation, performance gains obtained with this permutation with an unroll factor of 8 exceeded the performance gains obtained with any unroll factor for the *crs* permutation for the $conf3$ and $conf4$ problem size configuration.

4.2 N and K Index Permutation for Thread Mapping

Along with the permutation of sequential loop indices, it was also possible to reorder the mappings between the array indices and the overall thread grid. Figure 6 shows the performance comparisons observed when the n and k array indices were permuted across the thread

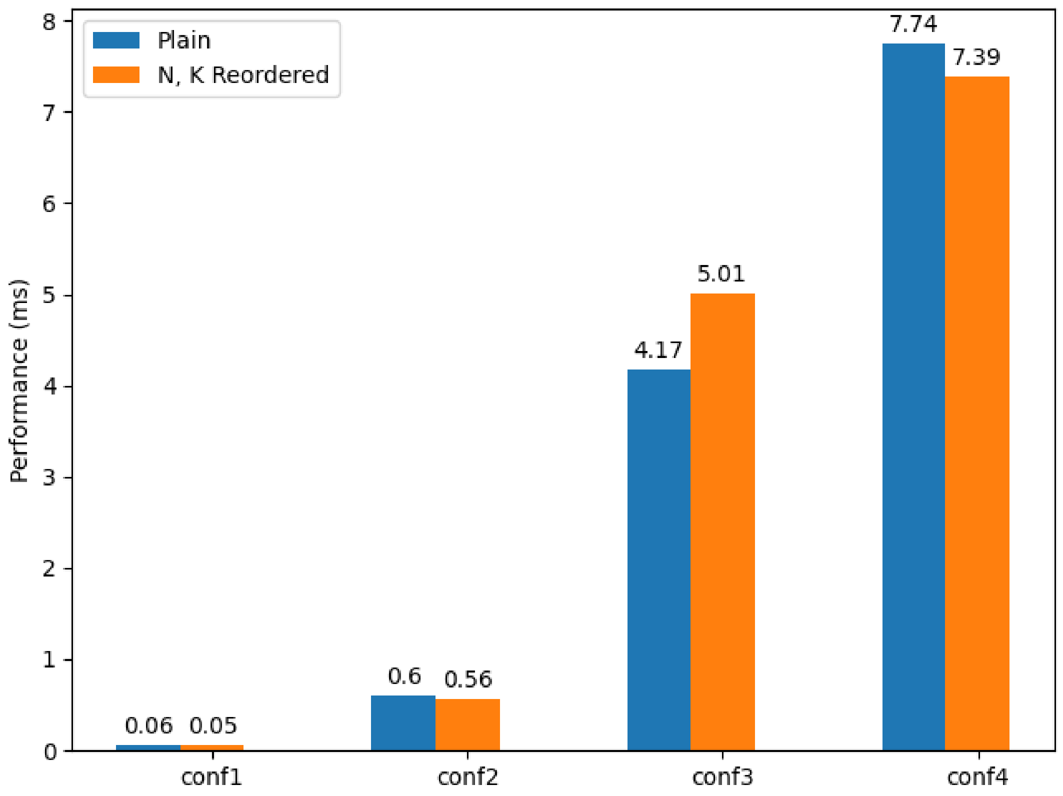


Figure 6: Performance Comparisons between q, p, n, k and q, p, k, n thread mapping

grid. No specific trend was observed and as a result, this optimization was not incorporated within the general kernel.

4.2.1 Shared Memory Considerations

It must be noted that although, reordering of thread index mappings were not associated with major performance improvements, but temporal locality for d_weight could only be achieved when n and k indices were permuted. As a result, for the q, p, n, k thread index mapping, it was possible to incorporate the usage of shared memory. It must also be noted that this incorporation of shared memory was only valid when $N * P * Q > Blocksize$ and the remainder obtained on dividing $Blocksize$ by $N * P * Q$ was equal to 0. Since, $P * Q$ usually resulted in an odd integer, the requirement boiled down to having N to be a multiple of the $Blocksize$ parameter. These conditions also put severe restrictions on the value of blocksize requiring it to not exceed the value N to make sure that N remained a multiple of it.

Shared memory utilization was performed by tiling the c loop within the sequential rsc permutation. A separate shared memory array was created which was filled by each one of the executing threads in a block with Tilesize kept equal to the value of Blocksize. Figure 7 shows the performance comparison between the *plain* kernel incorporating no use of shared memory and the shared memory kernel. To ensure a fair comparison, the *plain* kernel also kept the q, p, n, k thread index mapping. The shared memory kernel showed performance

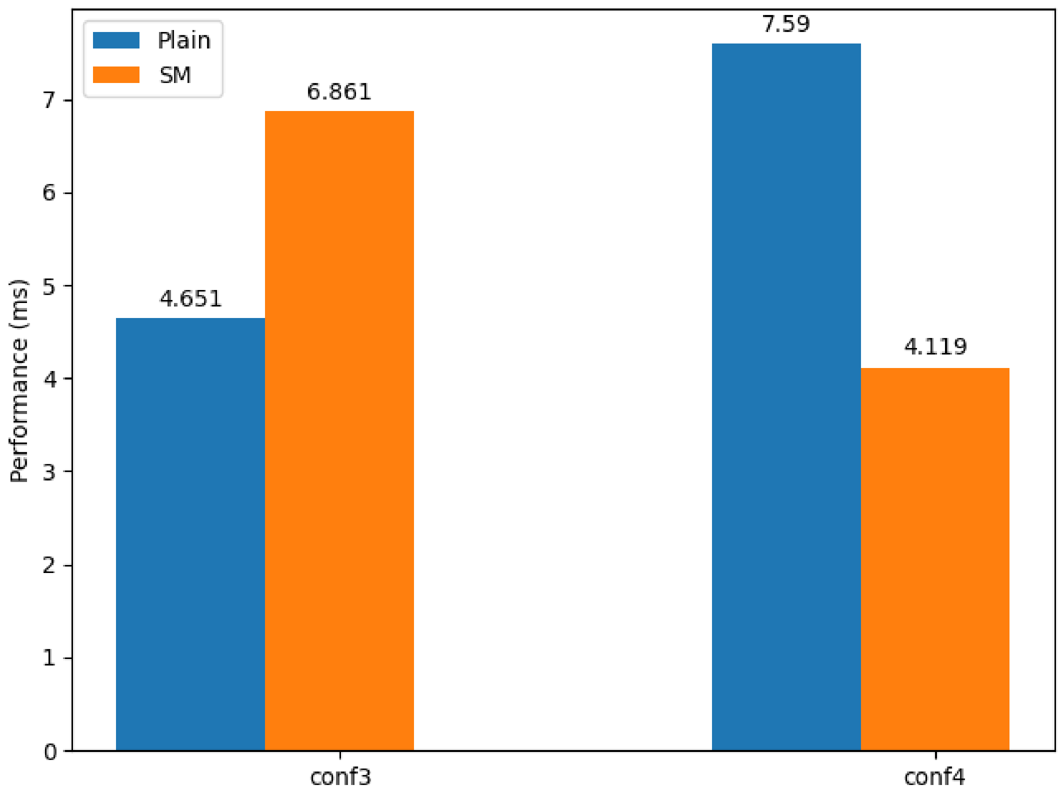


Figure 7: Performance Variations observed with the Inclusion of Shared Memory when $N = 128$

improvements for the case when $C = 832$. Since, $C = 3$ for the $conf3$ problem size variation, it was natural to observe a performance loss for this case. Furthermore, due to restrictions on values of N, this optimization could not be attempted for the case where $N = 1$. Hence, utilization of shared memory was only kept experimental and further analysis was not performed.

5 Unroll Jam Performance

Since k loop unrolling was found to be favorable for cases having high values of K, attempts were made to also perform simultaneous unrolling of n and k loops for the case where both of these parameters had high values ($N = 128$ and $K >= 64$). Figure 8 shows the performance comparison obtained by unrolling both n and k loops by a factor of 4 and the peak performance obtained with any degree of k loop unrolling. It is observed that simultaneous unrolling of these loops leads to steep performance gains having the potential to double the performance improvement for the $conf4$ problem size variation. As a result, this optimization was included in the general kernel for all cases where $N = 128$.

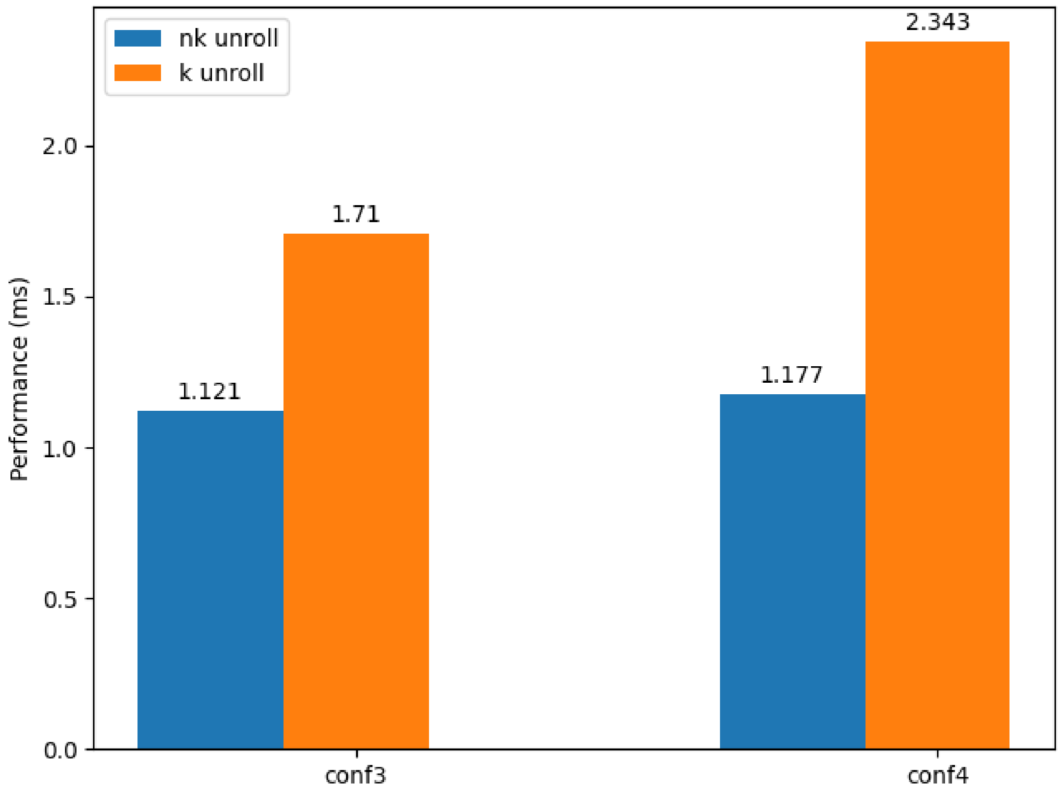


Figure 8: Peak Performance Comparisons between Individual k Loop Unrolling and Simultaneous unrolling of n and k loops when $N = 128$

6 Appendix

The file used for performing benchmarking on the CHPC cluster is reported within this subsection.

```python
import os
import subprocess
import re
from matplotlib import pyplot as plt
import numpy as np

fig, ax = plt.subplots()

cmd = list()

cmd.append("./cnn-gpu 1 3 64 112 112 3 3 2 2")
cmd.append("./cnn-gpu 1 832 128 7 7 1 1 1 1")
cmd.append("./cnn-gpu 128 3 64 112 112 3 3 2 2")
cmd.append("./cnn-gpu 128 832 128 7 7 1 1 1 1")

confs = ["conf1", "conf2", "conf3", "conf4"]

numeric_const_pattern = '[-+]? (?: (?: \d* \. \d+ ) | \
                 (?: \d+ \.? ) ) (?: [Ee] [+-]? \d+ ) ?'

def runallConfs(root_dir, outfile, compile_cmd, cmd):

    build_dir = root_dir + "/build"

    open(outfile, 'w').close()
    compile_cmdlist = compile_cmd.split(" ")
    subprocess.run(compile_cmdlist, cwd=build_dir)

    for item in cmd:

        cmdlist = item.split(" ")
        subprocess.run(cmdlist, stdout=open(outfile, 'a'),
                       stderr=open(outfile, 'a'), cwd=build_dir)


def plotallConfs(root_dir, outfile):

    build_dir = root_dir + "/build"

    rx = re.compile(numeric_const_pattern, re.VERBOSE)

    confs = ["conf1", "conf2", "conf3", "conf4"]
    vals = []

    with open(outfile, 'r') as fval:

        for idx, lineval in enumerate(fval.readlines()):

            if idx % 2 == 1:
                vals.append(float(rx.findall(lineval)[1]))

    print(vals)
    print(confs)
    bars = ax.bar(confs, vals, width=0.2)

    for bar in bars:
        yval = bar.get_height()
        plt.text(bar.get_x(), yval + .01, round(yval, 2))

    plt.xlabel(”Configuration”)
    plt.ylabel(”Performance␣(ms)”)
    plt.savefig(build_dir + ”/general.png”)
    plt.show()


def blockSizeBench(root_dir):

    linenumber = 0

    driverfile = root_dir + ”/driver.cu”
    build_dir = root_dir + ”/build”
    fullfile = []

    bsizes = [32, 64, 128, 256, 512, 1024]
    allvals = []

    for bsize in bsizes:

        with open(driverfile, ”r”) as fval:

            fullfile = list(fval.readlines())

            for idx, line in enumerate(fullfile):

                if re.match(”#define␣BLOCK_SIZE”, line):
                    linenumber = idx

        newlistval = ”#define␣BLOCK_SIZE␣” + str(bsize) + ”\n”

        newfullfile = fullfile[:linenumber] + \
            [newlistval] + fullfile[linenumber + 1:]

        with open(driverfile, ”w”) as fval:

            fval.writelines(newfullfile)

        outfile = build_dir + ”/out_” + str(bsize) + ”.txt”

        runallConfs(root_dir, outfile, ”make”, cmd)

        rx = re.compile(numeric_const_pattern, re.VERBOSE)

        vals = []
        with open(outfile, 'r') as fval:

            for idx, lineval in enumerate(fval.readlines()):

                if idx % 2 == 1:
                    vals.append(float(rx.findall(lineval)[1]))

        allvals.append(vals)

    plt.figure()

    allvals = np.array(allvals)

    for idx, conf in enumerate(confs):

        plt.plot(bsizes, allvals[:, idx], "-o", label=conf)

    plt.xlabel("Block Size")
    plt.ylabel("Performance (ms)")
    plt.legend()
    plt.savefig("Block_Size_Benchmark128.png")


def kunrollBench(root_dir):

    kfactors = [2, 4, 8, 16]
    build_dir = root_dir + "/build"

    allvals = []

    for kfactor in kfactors:

        foldername = root_dir + "/unroll_k" + str(kfactor)

        fname = foldername + "/cnn.assign.cu"

        compile_cmd = "nvcc -O3 -o cnn-gpu " + fname
        outfile = build_dir + "/outk" + str(kfactor) + ".txt"

        runallConfs(root_dir, outfile, compile_cmd, cmd[:2])

        rx = re.compile(numeric_const_pattern, re.VERBOSE)

        vals = []

        with open(outfile, 'r') as fval:

            for idx, lineval in enumerate(fval.readlines()):

                if idx % 2 == 1:
                    vals.append(float(rx.findall(lineval)[1]))

        allvals.append(vals)
    plt.figure()

    allvals = np.array(allvals)

    for idx, conf in enumerate(confs[:2]):

        plt.plot(kfactors, allvals[:, idx], "-o", label=conf)

    plt.xlabel("K loop Unroll Factors")
    plt.ylabel("Performance (ms)")
    plt.legend()
    plt.savefig("KLoopUnroll_Benchmark1_bsize=128.png")


def cunrollBench(root_dir):

    cfactors = [0, 2, 4, 8, 16]
    build_dir = root_dir + "/build"

    allvals = []

    for cfactor in cfactors:

        foldername = root_dir + "/unroll_c" + str(cfactor)

        fname = foldername + "/cnn.assign.cu"

        compile_cmd = "nvcc -O3 -o cnn-gpu " + fname
        outfile = build_dir + "/outc" + str(cfactor) + ".txt"

        runallConfs(root_dir, outfile, compile_cmd, cmd[:2])

        rx = re.compile(numeric_const_pattern, re.VERBOSE)

        vals = []

        with open(outfile, 'r') as fval:

            for idx, lineval in enumerate(fval.readlines()):

                if idx % 2 == 1:
                    vals.append(float(rx.findall(lineval)[1]))

        allvals.append(vals)

    plt.figure()

    allvals = np.array(allvals)

    for idx, conf in enumerate(confs[:2]):

        plt.plot(cfactors, allvals[:, idx], "-o", label=conf)
    plt.xlabel("C loop Unroll Factors")
    plt.ylabel("Performance (ms)")
    plt.legend()
    plt.savefig("CLoopUnroll_Benchmark1.png")


def generalBench(root_dir):

    build_dir = root_dir + "/build"

    outfile = build_dir + "/out.txt"

    runallConfs(root_dir, outfile, "make", cmd)


def rscpermute(root_dir):

    build_dir = root_dir + "/build"

    outfile = build_dir + "/out.txt"

    foldernames = [root_dir + "/unroll_k8", root_dir + "/unroll_k8_rsc"]
    allvals = []

    for idx, foldername in enumerate(foldernames):

        fname = foldername + "/cnn.assign.cu"

        compile_cmd = "nvcc -O3 -o cnn-gpu " + fname
        outfile = build_dir + "/outpermute" + str(idx) + ".txt"

        runallConfs(root_dir, outfile, compile_cmd, cmd)

        rx = re.compile(numeric_const_pattern, re.VERBOSE)

        vals = []

        with open(outfile, 'r') as fval:

            for idx, lineval in enumerate(fval.readlines()):

                if idx % 2 == 1:
                    vals.append(float(rx.findall(lineval)[1]))

        allvals.append(vals)

    permutes = ["crs", "rsc"]
    x = np.arange(len(confs))  # the label locations
    width = 0.25  # the width of the bars
    multiplier = 0

    fig, ax = plt.subplots(constrained_layout=True)

    allvals = np.array(allvals)
    for idx, permute in enumerate(permutes):
        offset = width * multiplier
        rects = ax.bar(x + offset, allvals[idx, :], width, label=permute)
        ax.bar_label(rects, padding=3)
        multiplier += 1

    ax.set_ylabel('Performance (ms)')
    ax.set_xticks(x + width, confs)

    plt.savefig(build_dir + "rsc_vs_crs.png")


def nk_vs_k(root_dir):

    build_dir = root_dir + "/build"

    outfile = build_dir + "/out.txt"

    foldernames = [root_dir + "/unroll_nk", root_dir + "/unroll_k8_rsc"]
    allvals = []

    for idx, foldername in enumerate(foldernames):

        fname = foldername + "/cnn.assign.cu"

        compile_cmd = "nvcc -O3 -o cnn-gpu " + fname
        outfile = build_dir + "/outpermute" + str(idx) + ".txt"

        runallConfs(root_dir, outfile, compile_cmd, cmd[2:])

        rx = re.compile(numeric_const_pattern, re.VERBOSE)

        vals = []

        with open(outfile, 'r') as fval:

            for idx, lineval in enumerate(fval.readlines()):

                if idx % 2 == 1:
                    vals.append(float(rx.findall(lineval)[1]))

        allvals.append(vals)

    unrollconfs = ["nk", "k"]
    x = np.arange(len(confs[2:]))  # the label locations
    width = 0.25  # the width of the bars
    multiplier = 0

    fig, ax = plt.subplots(constrained_layout=True)

    allvals = np.array(allvals)

    for idx, unrollconf in enumerate(unrollconfs):
        offset = width * multiplier
        rects = ax.bar(x + offset, allvals[idx, :], width, label=unrollconf)
        ax.bar_label(rects, padding=3)
        multiplier += 1

    ax.set_ylabel('Performance (ms)')
    ax.set_xticks(x + width, confs[2:])

    plt.savefig(build_dir + "nk_vs_k.png")


def Plain_vs_Reorder(root_dir):

    build_dir = root_dir + "/build"

    outfile = build_dir + "/out.txt"

    foldernames = [root_dir + "/Plain", root_dir + "/Reorder"]
    allvals = []

    for idx, foldername in enumerate(foldernames):

        fname = foldername + "/cnn.assign.cu"

        compile_cmd = "nvcc -O3 -o cnn-gpu " + fname
        outfile = build_dir + "/reorder" + str(idx) + ".txt"

        runallConfs(root_dir, outfile, compile_cmd, cmd)

        rx = re.compile(numeric_const_pattern, re.VERBOSE)

        vals = []

        with open(outfile, 'r') as fval:

            for idx, lineval in enumerate(fval.readlines()):

                if idx % 2 == 1:
                    vals.append(float(rx.findall(lineval)[1]))

        allvals.append(vals)

    unrollconfs = ["Plain", "Reorder"]
    x = np.arange(len(confs))  # the label locations
    width = 0.25  # the width of the bars
    multiplier = 0

    fig, ax = plt.subplots(constrained_layout=True)

    allvals = np.array(allvals)

    for idx, unrollconf in enumerate(unrollconfs):
        offset = width * multiplier
        rects = ax.bar(x + offset, allvals[idx, :], width, label=unrollconf)
        ax.bar_label(rects, padding=3)
        multiplier += 1

    ax.set_ylabel('Performance␣(ms)')
    ax.set_xticks(x + width, confs)

    plt.savefig(build_dir + "Plain_vs_Reorder.png")


def SM_vs_Plain(root_dir):

    build_dir = root_dir + "/build"

    outfile = build_dir + "/out.txt"

    foldernames = [root_dir + "/Reorder", root_dir + "/SM"]
    allvals = []

    for idx, foldername in enumerate(foldernames):

        fname = foldername + "/cnn.assign.cu"

        compile_cmd = "nvcc␣-O3␣-o␣cnn-gpu␣" + fname
        outfile = build_dir + "/SM" + str(idx) + ".txt"

        runallConfs(root_dir, outfile, compile_cmd, cmd[2:])

        rx = re.compile(numeric_const_pattern, re.VERBOSE)

        vals = []

        with open(outfile, 'r') as fval:

            for idx, lineval in enumerate(fval.readlines()):

                if idx % 2 == 1:
                    vals.append(float(rx.findall(lineval)[1]))

        allvals.append(vals)

    unrollconfs = ["Plain", "SM"]
    x = np.arange(len(confs[2:]))  # the label locations
    width = 0.25  # the width of the bars
    multiplier = 0

    fig, ax = plt.subplots(constrained_layout=True)

    allvals = np.array(allvals)

    for idx, unrollconf in enumerate(unrollconfs):
        offset = width * multiplier
        rects = ax.bar(x + offset, allvals[idx, :], width, label=unrollconf)
        ax.bar_label(rects, padding=3)
        multiplier += 1
    ax.set_ylabel('Performance␣(ms)')
    ax.set_xticks(x + width, confs[2:])

    plt.savefig(build_dir + "SM_vs_Plain.png")


if __name__ == "__main__":

    root_dir = "/uufs/chpc.utah.edu/common/home/u1444601/CS6235/HW4"
    build_dir = root_dir + "/build"
    outfile = build_dir + "/out.txt"
    # runallConfs( root_dir )
    # plotallConfs( root_dir, outfile )
    # blockSizeBench(root_dir)
    generalBench(root_dir)
    # kunrollBench(root_dir)
    # cunrollBench(root_dir)
    # rscpermute(root_dir)
    # nk_vs_k(root_dir)
    # Plain_vs_Reorder(root_dir)
    # SM_vs_Plain(root_dir)
```