# Two-Dimensional Convolution on the SCC

David W. Illig Chen Liu
Department of Electrical and Computer Engineering
Clarkson University
Potsdam, NY 13699, USA
{illigdw, cliu}@clarkson.edu

***Abstract*—Convolution is one of the most widely used digital signal processing operations. This work aims to distribute two-dimensional convolution operation across Intel's Single-Chip Cloud Computer (SCC), an experimental processor created by Intel Labs. This platform enables experiments with varying both the data sizes and the physical parameters of the platform such as voltage, frequency, and number of cores. The program can also be optimized subject to power and energy considerations. We find that implementing the convolution operation on the SCC can reduce the calculation time but results in a communication bottleneck. We find that calculations should be run at a lower frequency to reduce energy consumption, while communications should be run at a higher frequency to reduce execution time. Current applications are in the area of early vision using a Gaussian pyramid, while we aim to expand the study to additional image processing areas.**

***Keywords—Convolution; SCC; FFT; Parallelism; DVFS***

## I. INTRODUCTION

Convolution is a widely used operation in image processing, including applications such as smoothing, blur removing, filtering, and edge detection. In all of these operations, a filter kernel is convolved with an image matrix that is typically much larger than the kernel. Performing a convolution operation in a parallel fashion should allow for reduced execution time hence better performance.

The Single-Chip Cloud Computer (SCC) experimental processor [1-3] is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research. The SCC combines 48 P54C Pentium processor cores on a single chip. The 48 cores are placed in a tile formation, with two cores per tile. A single router is shared by the cores on a tile, which is connected to an on-die mesh network. The tiles are arranged in a 6×4 mesh, as shown in Figure 1. Each core has a private on-core L1 cache with 16 KB of data and 16 KB of instruction storage. Each core also has a private on-tile, unified L2 cache with 256 KB of storage. Each tile has a 16KB block of SRAM organized onto a shared address space visible to all cores on the chip. This memory is called the "message-passing buffer" (MPB), as it is a communication buffer that supports the movement of L1 cache lines between cores. The 64 GB off-chip DRAM is divided into memory regions that can be either private to each core or shared by all cores.

Intel provides a small, customized message-passing library for the SCC called RCCE [4, 5]. Program execution begins on a second machine called the management console PC (MCPC), which then issues commands to load programs onto individual cores on the SCC. The MCPC and the SCC cores all run versions of the Linux operating system. RCCE is built similarly to the message-passing interface (MPI) and OpenMP with functions customized for the SCC. RCCE provides utilities for communication, synchronization, memory management, and power management [4]. Power management is done by setting the frequency and voltage. As shown in Figure 1, the granularity for frequency changes is one tile, while the granularity for voltage changes is eight cores and is called a power domain. The 48 cores are divided into six power domains labeled from 0 to 5; note that the logical numbering of cores does not follow the power domain distribution. In total there are 69 frequency-voltage combinations, or "gears", possible on the SCC. This work will use a very small subset of gears for proof of concept purposes.


**Figure 1. SCC Layout (modified from [4])**

This work focuses on distributing the convolution operation across the 48 cores of SCC. The program can be optimized subject to power and energy constraints as well via SCC platform. Due to the novel status of the SCC, there is very little comparable work that has been published at present on the SCC. Wang *et al.* explored the parallelism of Fast Fourier Transform (FFT) on the SCC [6]. As our convolution approach will also use the FFT, there is some common ground. They found that they can achieve significant performance improvement by dividing the FFT task across multiple cores, up to 25 times speedup when using 32 cores. However, they do not perform any power or energy analysis. Additionally, their FFT algorithm requires that cores communicate during every transform operation due to a need to share data, while the approach we take

will distribute data such that cores work on independent data segments.

The parallelism of the FFT has been well studied on several other hardware platforms, many of which use more sophisticated hardware than the SCC. Sapir implemented a distributed FFT algorithm on a 16-core Epiphany Network-on-Chip (NOC) processor, finding that this particular NOC was limited to a maximum image size of 128×128 and that almost 90% of the execution time was spent on the butterfly stages of the FFT [7]. Although the SCC has more cores that each have more memory than those of the Epiphany NOC, there is still some similarity in that our approach decomposes the image matrix in a similar manner to Sapir's approach. Bahn *et al.* developed three parallel FFT algorithms that they tested on a simulated NOC using SystemC, concluding that the algorithm with the most balanced work load and least communication overhead allowed for fastest computation of the FFT [8]. Our approach is similar in that we attempt to minimize communication and ensure that all cores have the same amount of data to work on during parallel computation steps. Moreland *et al.* explored distributing the FFT on a commodity graphics processing unit (GPU) [9]. As the SCC is based on older Pentium processors, it is difficult to make a fair comparison to performance achieved by a GPU, though it may be of interest to adapt certain aspects of Moreland's algorithm for the SCC. Pippig developed an MPI-based library for computing the FFT of multidimensional matrices which was tested on three supercomputers at the Jülich Supercomputing Center, showing scalability for large number of cores superior to previous FFT libraries, while also analyzing the percentage of execution time spent on communication and computation steps [10]. While Pippig's use of supercomputers with thousands of cores and more than 0.5 GB of memory per core prevents any fair comparison to the SCC's capabilities, the execution time trends observed may provide insight into how the SCC might perform as number of cores or matrix size is increased.

The rest of this paper is organized as follows: Section II provides some background material on the convolution; Section III describes our algorithm and data collection methodology; Section IV presents experimental results; Section V discusses the results and experience of using our algorithm on the SCC as well as raising areas of future work and application; Section VI provides a conclusion summarizing the work.

## II. Background

This section will provide a brief overview of the mathematical foundations relevant to this work. We will give a high-level overview of the mathematical definitions and properties used to implement this study.

Two-dimensional convolution of $N \times N$ matrices $F$ and $G$ is defined by the following integral [11-13]:

$$H = F * G = \iint f(a,b)g(x-a, y-b)\, da\, db \qquad (1)$$

To calculate one entry in the convolution matrix $H$, due to the shifting operation of the convolution, the corresponding entry in the matrix $F$ must be multiplied by all $N^2$ elements of the matrix $G$. These products must be added together to obtain the final result. This process must be repeated for each of the $N^2$ elements of $F$ to compute the remaining entries of $H$ [14].

The convolution requirements can be greatly reduced by taking advantage of the properties of the Fourier Transform. The Fourier Transform of a two-dimensional matrix is given by the following integrals, with Equ. 2 giving the forward transform and Equ. 3 giving the inverse transform [13]:

$$G(a,b) = \iint g(x,y) \cdot \exp\big(-j2\pi(ax+by)\big)\, dx\, dy \qquad (2)$$

$$g(x,y) = \iint G(a,b) \cdot \exp\big(j2\pi(ax+by)\big)\, da\, db \qquad (3)$$

Throughout the remainder of the paper, we will use the symbol $FT\{X\}$ to represent taking the Fourier Transform of matrix $X$, and the symbol $FT^{-1}\{X\}$ will represent taking the Inverse Fourier Transform of the matrix $X$.

The Fourier Transform of a convolution is equivalent to the product of the Fourier Transforms of the two input matrices [12]. This will allow us to calculate the convolution of two matrices using the following equation:

$$H = F * G = FT^{-1}\{FT\{F\} \cdot FT\{G\}\} \qquad (4)$$

Using the definition of the Fourier Transform, this approach has a run-time of $O(N^2)$. However, the Fast Fourier Transform (FFT) can be used to improve the run-time performance to be $O(N \log N)$. For sufficiently large matrices, the overhead from computing the FFT will be insignificant compared to the performance improvement versus calculating convolution directly from its definition.

A final property used to improve performance in this study is that the Fourier Transform is a separable operation [13,15]. That is, a multi-dimensional Fourier Transform can be separated into a series of one-dimensional Fourier Transforms. This can be derived from Equ. 2, as the exponential function can be split into independent functions of x and y, allowing for integration along the x-dimension to be performed independently of the y-dimension. The benefits of this relation are two-fold. First, we need only implement a one-dimensional Fourier Transform to perform the convolution. Second, as each one-dimensional transform will work with less data than a multi-dimensional transform, we would expect the one-dimensional transforms to complete faster.

## III. Methodology

This section will discuss the methodologies and algorithm used in this work. The algorithm design will be presented, followed by discussion of how certain aspects of the algorithm will be implemented on the SCC. This will be followed by a brief overview of the measurement and calculation schemes used to assess this work.

### A. Algorithm Design

The algorithm developed for this study is an attempt to balance some known improvements for computing two-dimensional convolution alongside the specific attributes of the SCC. A pseudo-code representation of the algorithm is presented in Figure 2.

1. **Load** $A$ and $B$ from memory
2. **Partition** $A$ and $B$ across $N$ cores
3. Each core calculates **row FFT** of $1/N$ rows of $A$ and $B$
4. **Assemble** $C = FT\{A\}$ and $D = FT\{B\}$
5. Compute **transpose** $E = C^T$ and $F = D^T$
6. **Partition** $E$ and $F$ across $N$ cores
7. Each core calculates **row FFT** of $1/N$ rows of $E$ and $F$
8. Each core calculates $1/N$ rows of **complex multiplication** $G = E \cdot F$
9. Each core calculates **row IFFT** of $1/N$ rows of $G$
10. **Assemble** $H = FT^{-1}\{G\}$
11. Compute **transpose** $I = H$
12. **Partition** $I$ across $N$ cores
13. Each core calculates **row IFFT** of $1/N$ rows of $I$
14. **Assemble** $J = FT^{-1}\{I\}$
15. **Save** $J$ to memory

**Figure 2. Pseudo-code of Convolution Algorithm**

The algorithm consists of eight repeated steps, which we will refer to as the program phases; these have been highlighted in bold in Figure 2. Note that due to the use of the Fourier Transform, all matrices are complex data, which we represent using one matrix to store the real part and another to store the imaginary part. This program is a mixed workload consisting of some sequential steps and some parallel steps. We use Core 00 of SCC to control the process and perform the sequential operations of file input/output, partition, assemble, and transpose operations. In the partition operation, Core 00 uses message-passing to send sub-matrices to the other cores, while in the assemble operation the other cores send their submatrix results to Core 00. Forward and inverse Fourier Transforms as well as the complex multiplication are distributed across all cores involved in program execution. As discussed previously, we implement a one-dimensional Fourier Transform. By transposing the matrix after a row transform, we are able to use the same code to process the columns, as in [15]. By sending an entire row of data to each participating core, the core has all the data it needs to perform the transform and thus does not need to communicate with any other cores while performing the calculation. All phases use custom implementations of the desired operation. The FFT and IFFT implementations use an iterative version of the Cooley-Tukey algorithm [16], which was adapted for the SCC from a version presented in [17].

**Table 1. Gear Settings**

| Gear | Voltage | Frequency |
|---|---|---|
| Idle | 0.8 V | 200 MHz |
| Low | 0.8 V | 533 MHz |
| High | 1.1 V | 800 MHz |

In addition to the message-passing interface, other SCC features are taken advantage of in this study as well. The most obvious one is the distribution of calculation steps across multiple cores. Cores are assigned not based on their numerical numbering, but based on their physical location in order to minimize the number of power domains used in each operation, in reference to Figure 1. We utilize three gear settings for testing our program on the SCC, where each gear is a distinct combination of voltage and frequency settings. Cores that are not active are placed in a low-power, idle state. In addition, execution is tested using high and low power gears, as shown in Table 1. Power domains are set statically at the start of program execution.

### *B. Data Collection*

For this study, both execution time and average power data were collected for each program phase as well as for the entire program. Execution time was measured after every phase of the program as well as at the end of the SCC program, using the RCCE timing function [4]. Power measurement was done in the standard way for the SCC presented in [18].

From the execution time and average power measurements, energy consumed by the SCC can be estimated. The energy was calculated as the product of total execution time and average power:

$$E = T_{exec} P_{avg} \tag{5}$$

In addition to calculating the energy consumption with Equ. 5, we are also able to compute the energy-delay product (EDP). EDP is a metric that incorporates information about both the execution time and energy consumption of a program, which is a balanced measure of both the response time from the user point of view and energy saving from the system point of view. EDP is calculated as the product of energy and total execution time:

$$EDP = T_{exec} E = (T_{exec})^2 P_{avg} \tag{6}$$

Beyond analyzing the complete program, we were also able to calculate the energy and EDP consumed by each program phase by measuring the execution time and power consumed by each phase and applying Equ. 5 and 6 to each phase.

The program was tested for matrices of representative sizes of 64×64 and 128×128. These sizes were felt to be reasonable to represent large image processing filter kernels. The program was also tested using various numbers of cores to explore the tradeoffs between parallel calculation and the overhead from the increased communication required as the number of cores increases.

## IV. RESULTS

First, results for the entire program will be presented. This will be followed by a presentation of results for specific phases of the program. The program was tested with configurations using 1, 2, 4, 8, 16, 32, and 48 cores.

### *A. Results for Entire Program*

Execution time, power, energy, and EDP trends were calculated for the entire program as a function of the matrix size as well as the number of cores. The trends for the low gear setting are shown in Table 2 and Table 3 for the 64×64 and 128×128 matrices, respectively; while the higher gear data for the 64×64 matrix is shown in Table 4 with the 128×128 matrix results in Table 5. As would be expected, in an identical configuration the high gear has better execution time, while the low gear has the better power reading. The results also show evidence of a communication bottleneck, indicated by the trend for execution time to increase as the number of cores increases.

In the absence of communication delays, we would expect that increasing the number of cores would reduce execution time. This issue will be discussed in more detail in the following subsection.

Different trends are observed for energy and EDP. For the smaller matrix size of 64×64, the lower gear always consumes less energy. In terms of EDP, the lower gear is slightly better when using a single core, while the reduced execution time of the higher gear results in better EDP reading when using multiple cores. For the larger matrix size of 128×128, the lower gear has better energy performance except when all 48 cores are active, at which point the higher gear consumes less energy. For the 128×128 matrix, the EDP performance is typically better with the higher gear. An exception occurs for the 16-core case, where the EDP of the lower gear is 1.3 J·s less than the higher gear scenario. In general, the reduced execution time of the higher gear allows for optimization of EDP performance, while the reduced power consumption of the lower gear allows for optimization of energy consumption.

### *B. Results for Program Phases*

For each program phase, the percentage of total execution time spent in that phase was calculated. Performance trends for energy and EDP were also calculated for each phase. The program phases can be grouped into three categories:

- Communication phases consisting of partition and assemble;
- Calculation phases consisting of FFT, transpose, multiply, and IFFT;
- File input/output phases consisting of load and save.

**Table 2. Low gear performance on 64×64 matrix**

| Cores | $T_{exec}$ (s) | $P_{avg}$ (W) | $E$ (J) | $EDP$ (J·s) |
|---|---|---|---|---|
| 1 | 0.28 | 28.20 | 7.86 | 0.59 |
| 2 | 0.37 | 28.32 | 10.41 | 0.91 |
| 4 | 0.46 | 28.16 | 12.92 | 1.62 |
| 8 | 0.60 | 29.90 | 17.73 | 3.63 |
| 16 | 0.83 | 31.85 | 26.37 | 10.14 |
| 32 | 1.29 | 36.35 | 46.89 | 43.88 |
| 48 | 2.39 | 36.68 | 90.14 | 185.75 |

**Table 3. Low gear performance on 128×128 matrix**

| Cores | $T_{exec}$ (s) | $P_{avg}$ (W) | $E$ (J) | $EDP$ (J·s) |
|---|---|---|---|---|
| 1 | 1.12 | 28.14 | 31.59 | 11.86 |
| 2 | 1.43 | 28.27 | 40.72 | 15.33 |
| 4 | 1.79 | 28.56 | 50.19 | 25.53 |
| 8 | 2.30 | 29.86 | 68.81 | 54.76 |
| 16 | 2.69 | 31.73 | 85.66 | 132.61 |
| 32 | 5.22 | 36.41 | 190.08 | 619.82 |
| 48 | 12.25 | 36.78 | 456.69 | 3013.46 |

**Table 4. High gear performance on 64×64 matrix**

| Cores | $T_{exec}$ (s) | $P_{avg}$ (W) | $E$ (J) | $EDP$ (J·s) |
|---|---|---|---|---|
| 1 | 0.25 | 46.43 | 11.67 | 0.77 |
| 2 | 0.27 | 48.06 | 12.96 | 0.82 |
| 4 | 0.33 | 49.23 | 16.43 | 1.45 |
| 8 | 0.43 | 52.93 | 22.65 | 3.31 |
| 16 | 0.61 | 59.47 | 36.18 | 10.03 |
| 32 | 0.97 | 71.83 | 69.86 | 42.10 |
| 48 | 1.54 | 73.44 | 116.66 | 136.30 |

**Table 5. High gear performance on 128×128 matrix**

| Cores | $T_{exec}$ (s) | $P_{avg}$ (W) | $E$ (J) | $EDP$ (J·s) |
|---|---|---|---|---|
| 1 | 0.83 | 46.35 | 38.59 | 10.13 |
| 2 | 1.02 | 48.12 | 48.93 | 12.88 |
| 4 | 1.25 | 49.37 | 61.80 | 21.76 |
| 8 | 1.61 | 52.87 | 84.96 | 45.98 |
| 16 | 2.25 | 59.52 | 133.03 | 133.92 |
| 32 | 3.63 | 71.91 | 260.32 | 583.54 |
| 48 | 5.66 | 73.53 | 428.38 | 1816.03 |

As a representative example, the percentage of execution time for each phase for the high gear operating on the 128×128 matrix is shown in Table 6. The trends for the other configurations are very similar to this example, which would be expected as the percentage of execution time should be independent of the gear setting. Several interesting observations can be made from the data in Table 6. For a small number of cores, the file input/output operations occupy the majority of the execution time. As desired, the execution time spent on the calculation steps decreases as the number of cores is increased. The communication phases of partition and assemble dominate execution time when 4 or more cores are used for this application. In particular, the partition step consumes at least 50% of execution time for 8 or more cores, as execution is essentially halted until Core 00 has been able to distribute all of the data to the other cores. From another perspective, for 8 or more cores, the parallel calculation steps occupy 1% or less of the execution time. This suggests that the program is not utilizing the message-passing architecture as well as it could be, as the current implementation results in a communication bottleneck which dominates execution time. This is in contrast to Pippig's algorithm, where message-passing was used without any significant communication bottleneck [10]. Alternative communication approaches are discussed in Section V as a possible area of future work.

**Table 6. Percentage of Execution Time for Phases at High Gear**

| | **Number of Cores** | | | | | | |
|---|---|---|---|---|---|---|---|
| | ***1*** | ***2*** | ***4*** | ***8*** | ***16*** | ***32*** | ***48*** |
| Load | 19.25 | 14.01 | 10.95 | 8.52 | 6.26 | 3.80 | 2.47 |
| Partition | 10.31 | 26.48 | 37.77 | 49.45 | 63.15 | 77.19 | 85.78 |
| FFT | 5.38 | 2.20 | 0.92 | 0.66 | 0.16 | 0.06 | 0.03 |
| Assemble | 6.07 | 13.84 | 14.51 | 12.83 | 9.97 | 6.27 | 4.12 |
| Transpose | 4.58 | 3.73 | 2.99 | 2.32 | 1.67 | 1.02 | 0.66 |
| Multiply | 0.43 | 0.12 | 0.05 | 0.03 | 0.01 | <0.01 | <0.01 |
| IFFT | 3.15 | 1.28 | 0.55 | 0.50 | 0.11 | 0.05 | 0.02 |
| Save | 50.82 | 38.36 | 32.27 | 25.71 | 18.66 | 11.60 | 6.92 |

The energy and EDP trends are summarized for each of the three categories of phases: communication, calculation, and file input/output. As before, the high gear always optimizes execution time for a given phase, while the low gear always optimizes power consumption. The high gear energy and EDP have been normalized against the low gear energy and EDP for comparison purposes in this section. That is to say, relative values less than one indicate that the high gear has better performance, while values greater than one indicate that the low gear has the better performance. The relative energy and EDP trends for the 64×64 matrix are shown in Figure 3 and Figure 4, respectively, with the trends for the 128×128 matrix shown in Figure 5 and Figure 6, respectively.

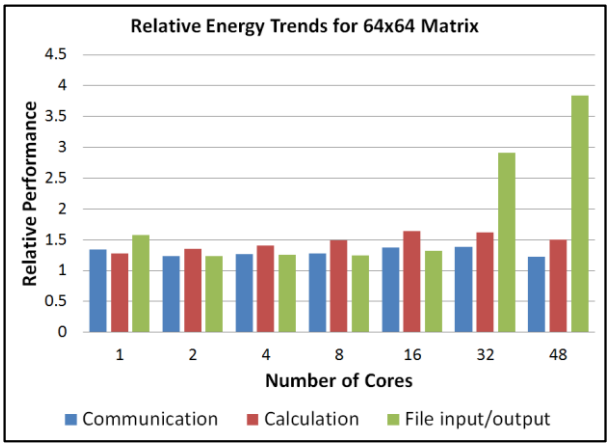


**Figure 3. Relative energy trends for 64×64 matrix**

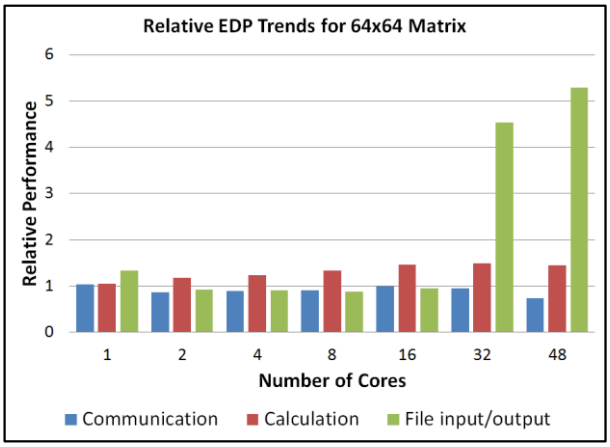


**Figure 4. Relative EDP trends for 64×64 matrix**

For the 64×64 matrix, energy consumption in every phase is minimized by using the lower gear, similar to the result observed across the entire program. In order to minimize EDP for a 64×64 matrix, from Figure 4 it appears that the low gear should be used for all calculation steps with the high gear used for all communication steps. This makes intuitive sense given that the communication steps are consuming the majority of the execution time and thus contribute more significantly to the EDP than any other phase. The file input/output phases for the 64×64 matrix follow a more complicate pattern, in which EDP is minimized by using the low gear for a single core as well as when 32 or 48 cores are used. This is believed to be due to the fact that while Core 00 is performing the file operations, the other cores are essentially doing nothing and should be set into the lowest possible gear in order to reduce power consumption.

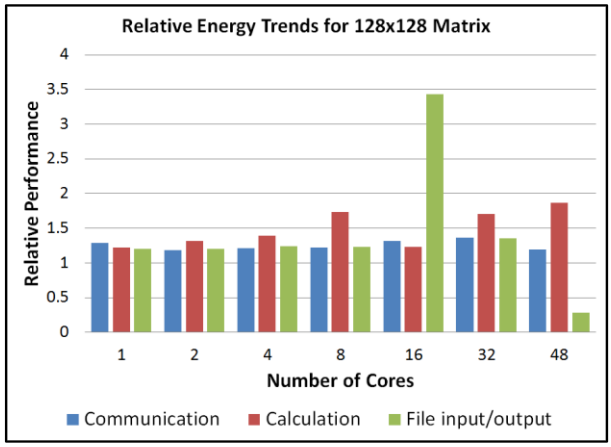


**Figure 5. Relative energy trends for 128×128 matrix**

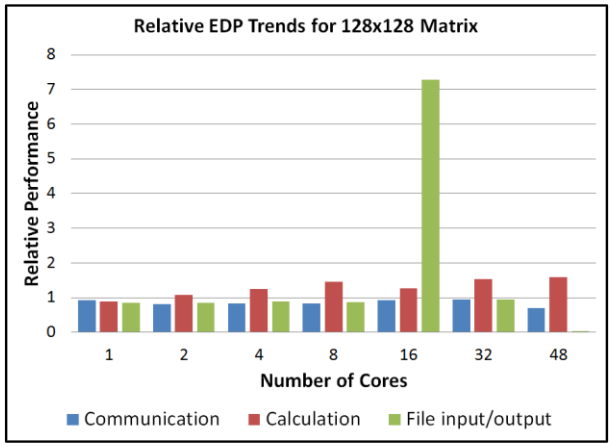


**Figure 6. Relative EDP trends for 128×128 matrix**

The results for the 128×128 matrix are similar, with energy almost always minimized by using the low gear as shown in Figure 5. An exception occurs for the file input/output phase when using 48 cores; this exception requires further investigation. The EDP trends for the 128×128 matrix shown in Figure 6 are also similar to the 64×64 matrix trend, with EDP of communication phases minimized in high gear and calculation phases minimized in the low gear. With the exception of the 16-core case, the EDP of the file input/output phases is also minimized in the high gear. This implies that for a larger matrix size, the reduced execution time from operating in the higher gear has more of an impact on the EDP than the increased power consumption. The 16-core case requires further investigation to determine why there is such a dramatic increase in the EDP.

## V. DISCUSSION

In this section implications of the preliminary results will be discussed. Following this, a few areas for potential future work will be described. Finally, a possible application for this study will be presented that is tolerant of the current limitations.

### A. Optimal Configurations

This subsection will briefly present the optimal configurations determined from the data collected for both the entire program and the different program phases. When looking at the entire program, in order to minimize energy consumption it is typically best to run in the low gear. An exception occurs when using 48 cores to process 128×128 matrices. On the other hand, when attempting to minimize EDP it is best to run in the high gear. The case of 16 cores operating on a 128×128 matrix offers an exception where the EDP of the high gear is approximately the same as the EDP of the low gear. These trends show that the increased power consumption of the high gear drives up its energy consumption, while the reduced execution time allows for a reduction in EDP.

The results for the individual program phases can also be used to optimize performance in terms of energy or EDP. In general, the low gear should be used on all phases in order to minimize the energy consumption. The EDP performance follows a more complicated pattern. The communication phases of partition and assemble should always be run in the high gear in order to reduce EDP; this suggests that the reduction in execution time of these phases is much more significant than their increased power consumption. The calculation phases require such a small percentage of the execution time that they can be run in the low gear in order to minimize EDP, showing that the power consumption is a more important consideration for the calculation phases than the execution time. Finally, the EDP trend for the file input/output phases follows a slightly more complicated trend. The low gear minimizes EDP for the file input/output phases using a 64×64 matrix when the program is executed on 32 or more cores. The high gear minimizes EDP for the 64×64 matrix when using fewer than 32 cores as well as for the larger 128×128 matrix, regardless of the number of cores. This suggests that the interaction between power consumption and execution time is more complicated for the file input/output phases than for the communication or calculation phases. Nonetheless, this information can be used to generate an optimal configuration in terms of energy or EDP for a specific matrix size and a specific number of cores.

### B. Future Work Ideas

The current program was developed as a proof-of-concept to explore the two-dimensional convolution problem and could potentially be improved. A significant area for future work would be to improve the communication structure used in this program. It was observed in the phase analysis that the communications phases become a bottleneck for a large number of cores. This is because Core 00 writes one piece of data at a time into its MPB, and then waits for the desired core to read this data. This sequential communication structure will greatly limit the potential execution time improvement of the parallel sections of the program as per Amdahl's Law [19]. Recently, an improved version of RCCE, named iRCCE [20], has been released. The iRCCE is the non-blocking communication extension to the RCCE library for SCC. It could reduce the communication overhead caused by RCCE during program execution, which is the subject of our next phase work. Another approach is to form a tree-structure among cores during data distribution. For example, Core 00 could initially distribute larger pieces of data to the master core in each power domain, which could then distribute to the remaining cores within each domain. Another option might be for each core to initially load its own input data and to write its own output file at the end of program execution. These separate output files could be combined together into a single output file later. This approach would remove the communication phases following the load operation and preceding the save operation, but would not offer any improvement to the other communication operations. Alternatively, a completely different approach could be taken, such as the lookup table based copy operation detailed in [21].

Other future work areas would help to move this work from a proof-of-concept state into something closer to an eventual application. In image processing, a filter kernel is typically a different size than the image to which it is applied, implying that one modification to the program would be to operate on matrices of different sizes. In addition, the filter kernel is typically applied to different regions of the image in order to achieve complete coverage, suggesting that the current work could be adapted to represent a function performing convolution of a filter kernel with a subsection of an image. An additional function could be written to iterate across the entire image, repeatedly performing the convolution operation designed in this work. Finally, the program's memory utilization could be improved by either performing more operations in place or using the RCCE dynamic memory allocation functions instead of static allocation [4].

### C. Possible Application Space

Although this study presently has a number of areas that could potentially be improved, it is still possible to see an application space for the current implementation. There is a common technique in image processing and computer vision in which a scaled representation of an image is generated, where each scaled representation has dimensions one-half of the previous scale [11]. This is also called creating a "Gaussian pyramid." This is done to generate images that contain some of the information of the original, larger image but can be processed more easily due to their reduced size. These are commonly used in applications termed "early vision", where some rudimentary processing or filtering is applied to determine regions of potential interest in the image. These regions can then be processed in more detail in the larger scales. As the current implementation has been shown to work properly on small matrices, it could be an ideal candidate for processing the lower scale images of a Gaussian pyramid in a distributed way.

## VI. Conclusions

This work presents a proof-of-concept for implementing two-dimensional convolution on Intel's Single-Chip Cloud Computer. The program was run using two gears to assess overall performance as well as individual performance of the eight program phases. It was found that the message-passing architecture results in a severe communications bottleneck as the number of cores is increased, such that the gains from distributed calculations are outweighed by the additional time spent on communications. It was also determined that the program should be run in the lowest gear possible in order to minimize energy consumption. On the other hand, the optimization of EDP requires consideration of the different program phases. For calculation phases, the program should be run in a low gear such that power consumption is reduced; while it should be run in a higher gear for the communication and file input/output phases such that execution time is reduced. Plans exist to explore the communications bottleneck and implement more advanced features. Even in the current state, this work could be applied to early vision analysis of low scale images in a Gaussian pyramid. Future work will aim to expand the application space for this study.

## Acknowledgement

The authors would like to thank Intel for providing the SCC platform to conduct this research, thank Gildo Torres for his technical support on SCC-related issues, and thank the anonymous reviewers for their constructive feedback. David Illig is supported by DoD SMART scholarship. This work is also partly supported by the National Science Foundation under Grant Number ECCS-1301953. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel, the Department of Defense, or the National Science Foundation.

## References

[1] Intel Labs, "Single-chip Cloud Computer," http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html (2009).

[2] J. Howard et al., "A 48-Core IA-32 Message-Passing Processor with DVFS in 45 nm CMOS," *ISSCC 2010* (2010).

[3] M. Baron, "The Single-Chip Cloud Computer: Intel Networks 48 Pentiums on a Chip," *Microprocessor Report* (2010).

[4] T. Mattson and R. van der Wijngaart, "RCCE: a Small Library for Many-Core Communication," 0th ed., Intel Corporation (2010).

[5] Intel Labs, "The SCC Programmer's Guide," 1st ed., Intel Corporation, (2012).

[6] Qing Wang and Nai-Jie Gu, "Research on the parallelism problem of FFT based on SCC," *MARC Symposium Wuxi* (2012).

[7] Y. Sapir, "Using a Scalable Parallel 2D FFT for Image Enhancement," Adapteva, http://www.adapteva.com/wp-content/uploads/2012/10/Using-a-Scalable-Parallel-2D-FFT-for-Image-Enhancement.pdf (2012).

[8] J.H. Bahn, J.S. Yang, W.-H. Hu, and N. Bagherzadeh, "Parallel FFT Algorithms on Network-on-Chips," *Journal of Circuits, Systems, and Computers* **18**: 255-269 (2009).

[9] K. Moreland and E. Angel, "The FFT on a GPU," *Graphics Hardware*, Eurographics Association (2003).

[10] M. Pippig, "PFFT: An Extension of FFTW to Massively Parallel Architectures," *SIAM J. Sci. Comput.* **35**: C213-C236 (2013).

[11] D.A. Forsyth and J. Ponce. *Computer Vision: A ModernApproach*. 2nd ed. New York: Pearson (2012).

[12] B.P Lathi. *Signal Processing & Linear Systems*. New York: Oxford University Press (1998).

[13] J.W. Woods. *Multidimensional Signal, Image, and Video Processing and Coding*. New York: Elsevier (2006).

[14] S.H. Ahn. "Example of 2D Convolution". *Digital Signal Processing*. http://www.songho.ca/dsp/convolution/convolution2d_example.html. (2011)

[15] I. Foster. "4.4 Case Study: Convolution". *Designing and Building Parallel Programs (Online)*. NSF Center for Research on Parallel Computation (1995).

[16] J.W. Cooley and J.W. Tukey. "An algorithm for the machine calculation of complex Fourier series". *Math. Comput.* **19**: 297–301 (1965).

[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. 3rd ed. Cambridge, Mass.: MIT Press (2009).

[18] T. Kubaska, "How to Read SCC Voltages with a RCCE Program," Intel Labs, http://communities.intel.com/docs/DOC-5463, Tech. Rep. (2010).

[19] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. New York: Morgan Kaufmann Publishers (2007).

[20] Carsten, C.; et al. "iRCCE: A Non-blocking Communication Extension to the RCCE Communication Library for the Intel Single-Chip Cloud Computer," Intel Many-core Applications Research Community, February 2011.

[21] M.W. van Tol, R. Bakker, M. Verstraaten, C. Grelck and C.R. Jesshope, "Efficient Memory Copy Operations on the 48-core Intel SCC Processor," *MARC Symposium Fraunhofer IOSB* (2011).