Efficient Nearest-Neighbor Data Sharing in GPUs

NEGIN NEMATOLLAHI, Department of Computer Engineering, Sharif University of Technology, Iran
MOHAMMAD SADROSADATI and HAJAR FALAHATI, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran

MARZIEH BARKHORDAR, Department of Computer Engineering, Sharif University of Technology, Iran

MARIO PAULO DRUMOND, EPFL University, Switzerland

HAMID SARBAZI-AZAD, Department of Computer Engineering, Sharif University of Technology, Iran and School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Iran

BABAK FALSAFI, EPFL University, Switzerland

Stencil codes (a.k.a. nearest-neighbor computations) are widely used in image processing, machine learning, and scientific applications. Stencil codes incur nearest-neighbor data exchange because the value of each point in the structured grid is calculated as a function of its value and the values of a subset of its nearest-neighbor points. When running on Graphics Processing Unit (GPUs), stencil codes exhibit a high degree of data sharing between nearest-neighbor threads. Sharing is typically implemented through shared memories, shuffle instructions, and on-chip caches and often incurs performance overheads due to the redundancy in memory accesses.

In this article, we propose Neighbor Data (NeDa), a direct nearest-neighbor data sharing mechanism that uses two registers embedded in each streaming processor (SP) that can be accessed by nearest-neighbor SP cores. The registers are compiler-allocated and serve as a data exchange mechanism to eliminate nearest-neighbor shared accesses. NeDa is embedded carefully with local wires between SP cores so as to minimize the impact on density. We place and route NeDa in an open-source GPU and show a small area overhead of 1.3%. The cycle-accurate simulation indicates an average performance improvement of 21.8% and power reduction of up to 18.3% for stencil codes in General-Purpose Graphics Processing Unit (GPGPU) standard benchmark suites. We show that NeDa's performance is within 13.2% of an ideal GPU with no overhead for nearest-neighbor data exchange.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**;

Additional Key Words and Phrases: Nearest-neighbor computations, data sharing, data re-usage, GPUs

Authors' addresses: N. Nematollahi, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran; email: negin.mahani@gmail.com; M. Sadrosadati, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran; email: m.sadr89@gmail.com; H. Falahati, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran; email: hajar.falahati@gmail.com; M. Barkhordar, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran; email: marzieh.barkhordar@gmail.com; M. P. Drumond, EPFL University, Lausanne, Switzerland; email: mario.drumond@epfl.ch; H. Sarbazi-Azad, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran, School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran; email: azad@sharif.edu; B. Falsafi, EPFL University, Lausanne, Switzerland; email: babak.falsafi@epfl.ch.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

1544-3566/2020/12-ART6

<https://doi.org/10.1145/3429981>

ACM Reference format:

Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, Mario Paulo Drumond, Hamid Sarbazi-Azad, and Babak Falsafi. 2020. Efficient Nearest-Neighbor Data Sharing in GPUs. *ACM Trans. Archit. Code Optim.* 18, 1, Article 6 (December 2020), 26 pages.
<https://doi.org/10.1145/3429981>

1 INTRODUCTION

Stencil codes are widely used in many important classes of data-parallel computing including applications ranging from image processing to machine learning and scientific computing [23, 44]. Stencil image processing filters eliminate the noise of the images [13], enhance images (e.g., to modify the image contrast) [16, 49], or compress them [51]. In machine learning, feature extraction (e.g., edge detection [17]) and convolutional neural networks exhibit nearest-neighbor data sharing like stencil codes [4]. Scientific computing uses stencil codes to solve partial differential equations [77] (e.g., computational physics and weather forecast [50]). As such, stencil codes and nearest-neighbor data sharing are often key targets for performance optimization in parallel computing [5, 36].

Like most data-parallel applications, stencil codes are typically run on Graphics Processing Units (GPUs) because of the latter's high computational density and silicon efficiency. Unfortunately, the performance of stencil codes running on GPUs is fundamentally limited by inherent nearest-neighbor data sharing; stencil codes exhibit a nearest-neighbour communication pattern, in which an output value is calculated based on its neighboring values in a structured grid [9, 64]. When running on GPUs, stencil codes exhibit a high degree of data sharing between nearest-neighbor threads [64].

To facilitate nearest-neighbor data sharing in a streaming multiprocessor, GPU programmers implement communication through a number of built-in mechanisms including the texture cache, the L1 data cache, the shared memory, and the shuffle instructions. Texture caches provide a good hit rate for nearest-neighbor data access patterns but incur a high access latency [38], limiting their effectiveness in providing fast nearest-neighbor data sharing. In contrast, L1 data caches are faster in zero-load scenarios [21, 26], but exhibit a lower hit rate for stencil codes. Shared memory backed with shuffle instructions are faster, and facilitates inter-thread data sharing but is limited to a thread block. On the other hand, the access latency of on-chip memories is significantly increased for memory-intensive stencil codes [37, 41, 76]. As a result, the thread-level parallelism cannot effectively hide these long memory access latencies.

In this article, we make the observation that carefully embedding two registers in each Streaming Processor (SP) that can be accessed by nearest-neighbor SP cores paves the way for a fast data sharing mechanism. The added registers can be interconnected through narrow local links placed around an SP's microarchitectural structures, thereby minimizing the impact on design density. We propose a new data sharing mechanism, called Neighbor Data (NeDa), between the nearest-neighbor SP cores inside a streaming multiprocessor. NeDa is a shared register storing the data value to be processed by a thread currently running on the SP and uses a modified thread scheduler to enable threads that share data to run together and be synchronized. We define a new compiler primitive in Compute Unified Device Architecture (CUDA) that transfers data values between nearest-neighbor SP cores. We place and route NeDa in an open source GPU and show an area overhead of 1.3%. Furthermore, our cycle-accurate simulations show an average performance improvement of 21.8% and power reduction of up to 18.3% over standard implementation of stencil codes. We show that NeDa's performance is within 13.2% of an ideal GPU with zero overhead for data exchange between nearest-neighbor SP cores.

Fig. 1. A conceptual view of 2D stencil code, called jacobi, which shows input structured grid, input window, output structured grid, and the path of the input window movement.

This article makes the following contributions:

- We show the suboptimality of current GPU hardware and software mechanisms to provide fast data exchange between nearest-neighbor SP cores.
- We propose NeDa, a hardware-software co-designed mechanism to provide fast and power-efficient data exchange between nearest-neighbor SP cores in GPUs, accelerating stencil codes.
- We place and route NeDa in an open source GPU and show an impact of 1.3% on GPU die area.
- We show that NeDa enhances the performance of standard stencil codes by an average of 21.8%, while reducing GPU power consumption by up to 18.3%. Moreover, NeDa's performance is within 13.2% of an ideal GPU architecture with zero overhead for data exchange between SP cores.

2 BACKGROUND

Stencil codes are common in many important classes of data-parallel applications such as image processing, machine learning, and scientific pipelines, widely employed for removal of noise from images, feature extraction, and solving partial differential equations [16, 77]. In stencil codes, the value of each point in a structured grid is updated as a function of the values of a subset of its nearest neighbors [22, 40]. Figure 1 presents an instance of 2D stencil code called jacobi. The code jacobi uses an input window that is convolved over an input structured grid and produces a weighted sum of a subset of the input window data values. The figure also shows the movement path of the input window. A stencil operator computes an output data value from five nearest-neighbor data values of the input window, with neighboring output data values sharing parts of their input windows.

In the GPU's single-instruction multiple-threads (SIMT) programming model, different output data values are mapped to threads running in different SP cores. SP cores are grouped in streaming multiprocessors. The SIMT threads are grouped in warps, and warps are grouped in thread blocks. In a warp, all of the threads run in a SIMD fashion, and all of the warps in a thread block are allocated to a single streaming multiprocessor. GPU kernels are composed of several thread blocks that are distributed across the streaming multiprocessors in a GPU. Hence, stencil codes can share

Fig. 2. The overview of a GPU architecture and the major components inside each streaming multiprocessor.

input both within and across streaming multiprocessors. In general, the shared data within a thread block can be re-read through shared memory (see blue lines in Figure 2), while the shared data across thread blocks need to be re-read from global memory in GPUs [42, 58] (see red lines in Figure 2).

GPUs schedule threads at two levels. The warp scheduler picks warps to execute within a streaming multiprocessor while the thread block scheduler picks thread blocks to distribute them across streaming multiprocessors [1, 56]. Both GPU schedulers play an important role in facilitating data sharing between stencil threads. It is desirable that neighboring stencil threads run as close together as possible, both in space (i.e., in the same streaming multiprocessors) and in time (i.e., neighboring thread blocks executing at the same time in different streaming multiprocessors).

Conventional GPUs offer support for data communication between threads through several on-chip memory structures such as the L1 data cache, texture cache, and shared memory. Hence, all of the communication among threads that share input data values occur through these memory structures. Figure 2 depicts the hierarchical organization of streaming multiprocessors, SP cores, and the on-chip memories in a GPU architecture. In this architecture, streaming multiprocessors are arranged around a shared L2 cache. Inside each streaming multiprocessor, there are SP cores all sharing the streaming multiprocessor on-chip memories and warp schedulers. Note that there is no direct link between SP cores [47].

3 MOTIVATION

Stencil codes is a major performance bottleneck in image processing, machine learning, and scientific applications, dominating total execution time. To illustrate this point, we profiled a set of common stencil codes in image processing, machine learning, and scientific applications on a Maxwell GTX 980 [47] GPU using Nvidia Profiler (NVPF) [48] profiler. For more information about the benchmarks we used, please refer to Section 5. The results illustrated in Figure 3 show that stencil codes take, on average, more than 94% of the total execution time of common image processing, machine learning, and scientific applications. Thus, it is crucial to improve the stencil codes performance to improve the overall performance of these applications.

One of the significant bottlenecks of stencil codes running on GPUs is the inefficient sharing of input data between nearest-neighbor threads. To show how prevalent such accesses are in typical stencil codes, we simulate the execution of several stencil codes in General-Purpose Graphics

Fig. 3. Stencil codes execution time normalized to the total application execution time.

Fig. 4. Nearest-neighbor data values accesses normalized to the total memory accesses in standard stencil codes.

Processing Unit (GPGPU)-Sim V3.2.2 [2] and present the fraction of the memory accesses that are shared between nearest neighbors in Figure 4. We identify a memory access as being made to a nearest neighbor whenever a thread accesses data that was previously accessed by another thread that is processing an adjacent output data. For more information about the benchmarks we used, please refer to Section 5. We observe that all of the applications exhibit a high degree of nearest-neighbor data sharing, resulting in an average of 62% of memory accesses being made to nearest-neighbor data values.

To facilitate the nearest-neighbor data sharing among threads in a streaming multiprocessor, GPU architectures offer a number of solutions, including the L1 data cache, texture cache, shared memory, and shuffle instructions. L1 data cache accelerates the nearest-neighbor data sharing among the threads in the streaming multiprocessor. However, as the stencil codes usually have 2D locality, the data values of nearest-neighbor threads usually reside in different cache blocks, reducing the hit rate. Texture cache is widely employed to implement stencil codes like image processing filters and graphics applications [9, 18]. However, the access latency of the texture cache [38] greatly reduces its effectiveness to provide fast nearest-neighbor data exchange in stencil applications. Shared memory in each streaming multiprocessor, which provides data sharing among

Table 1. Raw Latency (Cycle) of the Register, L1 Data Cache, Texture Cache, Shared Memory, and Shuffle Instruction for NVIDIA Maxwell GTX 980

Register	Shuffle instruction	Shared memory	L1 data cache	Texture cache
2	6	56	70	84

threads inside a thread block, can be used for implementing stencil codes. Unfortunately, shared memory only enables data sharing inside each thread block. Moreover, using the shared memory for data sharing has its own weaknesses, including stalls due to the bank conflicts [14, 24], limited capacity, which can limit the thread-level parallelism [29, 39], synchronization overhead incurred by safe sharing [67], and high programming complexity [66]. Shuffle instruction shares data among a warp's threads at a register level and is supported by devices with CUDA compute capability of 3.x or higher [46]. Unfortunately, the demand for inter-warp communication cannot be satisfied using shuffle instructions, and, hence, programmers should exploit both shuffle instructions and on-chip memories to capture intra-/inter-warp data sharing.

To compare the zero-load latency of baseline GPU mechanisms (i.e., the register, shared memory, L1 data cache, texture cache, and shuffle instruction) in providing inter-thread communication, we profiled a set of micro-benchmarks on a Maxwell GTX 980 [47] GPU using NVPROF [48] profiler. These kernels differ in their mechanism to get their input data (e.g., register, shared memory, L1 data cache, texture cache, and shuffle instruction). Table 1 shows the raw latency of the register, L1 data cache, texture cache, shared memory, and shuffle instruction, for NVIDIA Maxwell GTX980 [47] GPU. Based on the results of our micro-benchmarking experiments, the minimum access latency (i.e., in zero-load scenarios) of L1 data cache and texture cache is around 70 and 84 cycles, respectively, for NVIDIA Maxwell GTX 980 [47] GPU. Prior work also reports similar numbers [21]. We also measure these latencies in high-load scenarios. We observe that L1 data cache and texture cache access latencies can be increased to 629 and 633 cycles, respectively. These results clearly show how the access latency of on-chip memories is significantly increased for memory-intensive applications. As a result, the thread-level parallelism cannot effectively hide these long memory access latencies. The optimizations with shared memory can potentially improve the performance of memory-bound stencil computations, but shared memory effectiveness is limited by the constraints imposed by the limited on-chip memory capacity [37]. In other words, GPUs are designed to hide memory latencies. But this capability is effective as long as the memory bandwidth is not saturated. In stencil codes, which are typically memory-bandwidth bound [37, 41, 76], the memory access latency cannot be fully hidden through thread-level parallelism.

To overcome the suboptimality of the inherent GPU data sharing mechanism, programmers have to work hard to make optimal use of hardware and minimize the overhead of data exchange [53]. It is challenging for a programmer to code in optimal mode. Because in addition to mastering software techniques, the programmer must also have sufficient knowledge of hardware architecture. Besides a little mistake can result in suboptimal performance. Thus, programming convenience along with high performance can attract many developers of parallel programs [66]. We believe that mitigating the overhead of data exchange can greatly improve the programming ease.

To measure how far the performance of the standard GPGPU benchmarks is from ideal performance, in providing inter-thread communication in stencil codes, we perform a set of experiments using GPGPU-Sim V3.2.2 [2] comparing the performance of various standard stencil codes, called Standard to the ideal case (Ideal), which provides nearest-neighbor data exchange with zero latency. For more information on the benchmarks used, see Section 5. We use Instruction Per Cycle (IPC) as the performance metric. Figure 5 shows the IPC of different Ideal stencil codes, normalized to the IPC values of the Standard. We make a key observation. The IPC of the Ideal case is

Fig. 5. Total GPU IPC using Ideal normalized to Standard.

around 40% higher than Standard, showing the gap between the ideal case and the standard implementation of the stencil codes. Hence, by developing hardware/software support for fast nearest-neighbor data exchange, a high performance programming model can be released that has been freed from time-consuming and costly optimization, since nearest-neighbor communication can be handled at the local register level.

4 NEAREST-NEIGHBOR DATA SHARING

We now present NeDa, a new data communication mechanism between nearest-neighbor SP cores. NeDa addresses the shortcomings of the inter-SP communication mechanisms described in Section 3. We describe the communication infrastructure in Section 4.1, the architecture and compiler support in Section 4.2, the scheduler support in Section 4.3, and the communication mechanism between nearest-neighbor SP cores across streaming multiprocessors in Section 4.4. We present a running example in Section 4.5.

4.1 Communication Infrastructure Support

Figure 6 illustrates NeDa's communication infrastructure in a streaming multiprocessor. Figure 6(a) depicts the links between nearest-neighbor SP cores within a streaming multiprocessor. NeDa provides direct horizontal and vertical communication links between the nearest-neighbor SP cores in each streaming multiprocessor. We carefully analyze the area and power overhead of NeDa's communication infrastructure in Section 6.7. Note that there is no pre-existing interconnection network between SPs. In our baseline (see Figure 2), all inter-SP communication happens through the shuffle instructions, the shared memory, or the memory hierarchy.

4.2 Architecture and Compiler Support

We add two registers to each SP: (1) one of these two registers is to keep the data value required by the nearest neighbor, called Local-Reg; and (2) the other one is a temporary buffer used for moving Local-Reg value of an SP to another neighbor SP, which are not connected directly in the network, called Neighbor-Reg. Local-Reg and Neighbor-Reg are depicted in Figure 6(b). We map Local-Reg and Neighbor-Reg to the R1 and R2 architectural registers. We analyze the overhead of reducing the number of general-purpose architectural registers in Section 6.7. When the destination register

Fig. 6. NeDa microarchitecture: (a) shows the overview of NeDa communication infrastructure inside a streaming multiprocessor, and (b) shows the architectural support for the read-neighbor instruction.

Fig. 7. An example of read-neighbor instruction application to read the Local_Reg register value of the nearest-neighbor SP.

is R1, the operand collector proceeds to write this data to the SPs' Local_Reg using the existing link between operand collectors and SP cores.

In the baseline GPU architecture, stencil codes threads fetch nearest-neighbor data values from the memory hierarchy and then access them through the register file. In the NeDa architecture, nearest-neighbor data values are accessed through the Local_Reg registers instead. To read the Local_Reg from nearest-neighbor SP cores, we introduce a new instruction, called read-neighbor, shown in Figure 7. In the figure, the $x_{distance}$ and $y_{distance}$ parameters represent the coordinate difference between the source SP and the destination SP in X and Y dimensions, respectively. For example, $x_{distance} = +1$, $y_{distance} = 0$ and $x_{distance} = -1$, $y_{distance} = -1$ show right and left-down

Fig. 8. Move instruction syntax as a new PTX instruction.

Fig. 9. Examples of read-neighbor instruction along with the corresponding register transfers using move instruction: (a) register transfer from up nearest-neighbor SP, (b) register transfer from left nearest-neighbor SP, and (c) register transfer from down-right nearest-neighbor SP.

nearest neighbors, respectively. The memory address parameter is used for accessing the memory when the specified neighbor does not exist. An SP may default to accessing the memory if it does not have the nearest neighbor, specified by the coordinate difference arguments.

To compile read-neighbor instruction, we introduce a new move instruction at Parallel Thread Execution (PTX) level, shown in Figure 8. In the figure, the destination-reg operand is the destination Neighbor-Reg register. The source-reg operand can be either Neighbor-Reg or Local-Reg registers of the nearest-neighbor SP. The direction attribute (i.e., up, down, right, and left) specifies which nearest-neighbor SP should be accessed.

Figure 6(b) shows the architectural support for the move instruction. We use a 4-to-1 multiplexer to select the correct nearest-neighbor register and a 2-to-1 multiplexer to select between Local-Reg or Neighbor-Reg registers (i.e., R1 or R2).

The read-neighbor instruction is translated to one or a sequence of move instructions depending on $x_{distance}$ and $y_{distance}$ values. Figure 9 shows some examples of read-neighbor instruction translation. Figure 9(a) shows an example of read-neighbor instruction with $x_{distance} = 0$, $y_{distance} = +1$ arguments, i.e., each SP needs to transfer the value of its up nearest-neighbor Local-Reg (i.e., R1) to its own Neighbor-Reg register (i.e., R2). This register transfer is performed with a single move.down instruction from R1 register in up nearest-neighbor SP to R2 in the des-

termination SP. Figure 9(b) shows another example of read-neighbor instruction with $x_{distance} = -1$, $y_{distance} = 0$ arguments, i.e., each SP needs to transfer the value of its left nearest-neighbor Local-Reg register (i.e., R1) to its own Neighbor-Reg register (i.e., R2). This register transfer is performed with a single `move.right` instruction from R1 register in left nearest-neighbor SP to R2 register in the destination SP.

Figure 9(c) shows another example of read-neighbor instruction with $x_{distance} = +1$, $y_{distance} = -1$ arguments, i.e., each SP wants to transfer the value of its down-right nearest-neighbor Local-Reg register (i.e., R1) to its own Neighbor-Reg register (i.e., R2). This register transfer is performed with two move instructions: (1) a `move.left` instruction from R1 register in down-right nearest-neighbor SP to R2 register in down nearest-neighbor SP, and (2) a `move.up` instruction from R2 register in down nearest-neighbor SP to R2 register in the destination SP.

In NeDa, we handle all the details of the interconnection network at the software-level. All the operations in this network are managed by PTX move instructions. All the SIMD threads in a batch¹ fill their local registers in lock-step. Then, each SP core reads a local register data of a neighboring SP by executing a move instruction in the same direction (e.g., all threads in the batch read their left neighbor at the same time by executing `move.right` R2, R1 in lock-step). Then, each SP core consumes the read data as an input data value for the stencil function. This mechanism ensures the timeliness of the data. Moreover, there is no conflict since at the moment only one move instruction with the same attribute (e.g., left, right, up, and down) is running in a batch; hence, it does not occur that an SP core reads data in multiple directions.

4.3 Scheduler Support

NeDa adds a novel scheduling policy to the GPU in order to provide fast communication and implicit synchronization between the on-chip SP cores, which are allocated to nearest-neighbor threads (i.e., threads in charge of processing the nearest-neighbor data values). We modify both the thread block and warp schedulers to force nearest-neighbor threads to run on nearest-neighbor SP cores. Figure 10(a) shows an example configuration of 4-warp-size thread blocks and their mapping to a streaming multiprocessor. Figure 10(b) illustrates NeDa's scheduling policy for both thread blocks and warps, and their distribution within and across streaming multiprocessors. While the baseline scheduler assigns consecutive blocks to consecutive streaming multiprocessors, we assign two consecutive blocks to the same streaming multiprocessor (e.g., we assign block#1 and block#2 to streaming multiprocessor#1, and block#3 and block#4 to streaming multiprocessor#2). This thread block assignment effectively exploits data sharing between nearest-neighbor warps of consecutive blocks.

We also modify thread-to-SP allocation, mapping logical nearest-neighbor threads to nearest-neighbor SP cores. For applications with regular parallelism, programmers can identify which pieces of data are assigned to each warp. The data index is determined based on the thread index. High data sharing occurs between neighbor indices. Warps with high data sharing should be scheduled together. To make sure that this happens, we modify the warp scheduler to ensure the warps operating on nearest-neighbor data values are scheduled for executing together. To this end, four nearest-neighbor warps are grouped into a batch (called local batch). For example, as shown in Figure 10(c), for 4-warp-size blocks configuration, warp numbers 1, 2, 5, and 6, and warp numbers 3, 4, 7, and 8 form two different batches. The compiler can determine which warps belong to a batch using the information about the program threads (e.g., thread index) and the hardware platform (how the SP cores are placed in a physical design). In general, compilers benefit from

¹Four nearest-neighbor warps are grouped into a batch of threads, as a larger warp. For more information, please see Section 4.3.

Fig. 10. NeDa scheduler support: (a) shows an example of the block configuration, (b) shows block distribution among streaming multiprocessors, and (c) shows nearest-neighbor warp (i.e., batch) scheduling inside a streaming multiprocessor.

some basic hardware information that becomes updated once per GPU generation. However, the required hardware information for the compiler support in NeDa is small; we provide the compiler with the hardware information to figure out the communication infrastructure, once, and determine the batches. Batches are scheduled in a round-robin fashion. When a single warp in a batch stalls, the entire batch is descheduled and another active batch (i.e., a batch with four active warps) is scheduled for the execution.

Applications that NeDa targets usually have regular parallelism, and, hence, batch scheduling is effective for them. Several pieces of prior work also show that applications with regular parallelism can benefit from larger warp sizes [32, 43, 55]. This is because the warps in the batch (1) have similar behavior, and they stall almost at the same time; and (2) they exploit potentially existing memory access localities among neighbor threads and coalesce them to fewer off-core requests. Our experimental results show that using batch scheduling (without applying NeDa mechanism) can improve overall GPU performance by up to 5% for stencil applications, compared to the baseline two-level round-robin warp scheduling.² For the applications that usually experience irregular parallelism, such as graph applications, batch scheduling can impose some performance overhead. Our experimental results show up to 12% (4%, on average) performance overhead for the evaluated

²A well-known and simple baseline scheduler in scientific research.

Fig. 11. The impact of batch scheduler on total GPU IPC normalized to two-level round-robin scheduler.

workloads from Polybench [19], Rodinia [6], Parboil [62], Shoc [10], and NVIDIA CUDA Software Development Kit (SDK) [45] benchmark suites (Figure 11). To deal with this issue, we devise a reconfigurable scheduler that can switch between batch scheduling and normal two-level round-robin scheduling when the kernel is launching. Determining the scheduler type depends on the type of the kernel (i.e., if it is a stencil kernel, it uses the batch scheduler). The programmer uses a special instruction (i.e., `read-neighbor` in CUDA programming model or `move.down`, `move.up`, `move.left`, and `move.right` in PTX) in the stencil code. By compiling the program statically, the compiler figures out the keyword, detects the kernel type, and sets the scheduling policy for the kernel.

We evaluate the hardware cost of our new scheduler. We synthesize the hardware description language (HDL) model of the modified scheduler using Synopsys Design Compiler for 45nm technology node. We observe that our new scheduler consumes 0.1% and 0.2% more area and power consumption compared to the baseline two-level round-robin warp scheduler. To estimate the hardware overhead of the GPU architecture with batch scheduler and two-level round-robin scheduler, we implement it in the MIAOW [3] open-source GPU, and place and route the design on an Field Programmable Gate Arrays (FPGA). We observe that the GPU architecture with both scheduling policies impose negligible area and power overhead (less than 0.05%) over the baseline GPU architecture.

4.4 Nearest-Neighbor Streaming Multiprocessor Data Sharing Support

An SP may default to reading a nearest-neighbor data value from on-chip caches if it is located at a streaming multiprocessor's borders (i.e., some of its required nearest neighbors does not exist in the same streaming multiprocessor). We minimize on-chip caches accesses by adding local wires between nearest-neighbor border SP cores of different streaming multiprocessors. In this way, a border SP can read a data value from the corresponding nearest-neighbor SP, defined by $x_{distance}$ and $y_{distance}$ parameters in the `read-neighbor` instruction, even if the required data value resides in the neighbor streaming multiprocessor. We connect the border SP cores of nearest-neighbor streaming multiprocessors. We refer to this mechanism as NeDa+. Figure 12 demonstrates NeDa+ communication infrastructure embedded in a GPU architecture, indicating exactly how border SP cores of two nearest-neighbor streaming multiprocessors communicate with each other via NeDa+ communication wires (i.e., blue links). Note that we still need to access on-chip caches to read nearest-neighbor data values for some of border SP cores, when the required neighbor does not exist.

To improve the effectiveness of NeDa+, local batches with more data sharing should run together in nearest-neighbor streaming multiprocessors. To this end, we coordinate the batch schedulers

Fig. 12. NeDa+ links between the nearest-neighbor streaming multiprocessors.

in nearest-neighbor streaming multiprocessors to ensure that the local batches operating on the nearest-neighbor data values are scheduled for executing together on the GPU. We define a new batch, called global batch, as a group of multiple local batches. Global batches are scheduled in a round-robin fashion. When a warp in a global batch stalls, the corresponding global batch is descheduled and another active global batch (i.e., all local batches are active in a global batch) is scheduled for the execution. When there is no active global batch (i.e., all global batches are stalled), we select an inactive global batch randomly to sacrifice it and schedule the local batches inside this global batch. The schedulers in nearest-neighbor streaming multiprocessors start scheduling these active local batches. Then, two scenarios can happen: (1) At some point during scheduling local batches, a global batch from the remaining inactive global batches becomes active. In this situation, we switch the scheduling to the global batch scheduling. (2) All local batches are stalled and still; there is no active global batch. In this case, we select another global batch randomly and continue local batch scheduling from the local batches of this new selected global batch. When nearest-neighbor streaming multiprocessors are running local batches, we disable local wires between streaming multiprocessors to reduce their leakage power consumption. When a global batch becomes active again, nearest-neighbor streaming multiprocessors will run global batch. Note that our mechanism has created two levels of batched threads (i.e., local and global batches) for coordinating within and across the streaming multiprocessor cores. When a global batch is issued, the coordination between the allocated streaming multiprocessors to that batch is guaranteed by relying on the lock-step execution in the batch.

4.5 A Running Example

We use NeDa to implement a simple example of 2D stencil codes. Note that it is just a running example of NeDa for better understanding. Algorithm 1 and Algorithm 2 show the pseudo codes

for implementing a sample jacobi-kernel using Standard and NeDa, respectively. The jacobi-kernel utilizes the center data value and the data values of its up, down, right, and left nearest neighbors to produce a weighted sum of them as the output for the corresponding center data value. In the Standard code, each thread (1) loads all of the required data values in the nearest-neighbor window one after another, and (2) realizes a stencil code operation on them to generate the output (lines 5, 6, and 7). In the NeDa code, each thread (1) loads its Local-Reg register, and (2) performs read-neighbor instruction to read Local-Reg register value of the nearest-neighbor SP cores (lines 6, 7, 8, and 9), instead of doing repetitive memory accesses to obtain data values.

ALGORITHM 1: jacobi-kernel: Standard

```

1: int j = blockIdx.x × blockDim.x + threadIdx.x;
2: int i = blockIdx.y × blockDim.y + threadIdx.y;
3: ⋮
4: // Each thread reads its corresponding data value, and the required nearest-neighbor data values from memory.

5: B[i + 0][j + 0] = C * (A[i + 0][j + 0] +
6: A[i + 0][j - 1] + A[i + 0][j + 1] +
7: A[i - 1][j + 0] + A[i + 1][j + 0]);
8: ⋮

```

ALGORITHM 2: jacobi-kernel: NeDa

```

1: int j = blockIdx.x × blockDim.x + threadIdx.x;
2: int i = blockIdx.y × blockDim.y + threadIdx.y;
3: ⋮
4: // Each thread reads its corresponding data value from memory, and saves it in the corresponding Local-Reg register.
   Then each thread reads its required data values from nearest-neighbor SP's Local-Reg registers using read-neighbor
   instruction.

5: B[i + 0][j + 0] = c * (A[i + 0][j + 0] +
6: read_neighbor(-1, 0, A[i + 0][j - 1]) +
7: read_neighbor(+1, 0, A[i + 0][j + 1]) +
8: read_neighbor(0, -1, A[i - 1][j + 0]) +
9: read_neighbor(0, +1, A[i + 1][j + 0]));
10: ⋮

```

5 METHODOLOGY

5.1 Simulation

We evaluate NeDa using GPGPU-Sim V3.2.2 [2] and GPUWattch [34], modeling a Maxwell-like GPU configuration [47]. Table 2 shows the details of our simulation parameters. We modify different parts of GPGPU-Sim (e.g., the front-end, the compiler tool, the decode stage, and the execution units) to provide the architectural support of NeDa (see Section 4.2 for more details).

5.2 Benchmark

To compare against prior GPU work on stencil codes, we evaluate NeDa against the last implementations of the standard stencil codes from Polybench [19], Rodinia [6], Parboil [62], Shoc [10], and NVIDIA CUDA SDK [45] benchmark suites as shown in Table 3.

Table 2. Simulation Configuration

Parameter	Value
Number of streaming multiprocessors	16
Core clock	1216 MHz
Scheduler	Two-level round-robin
Number of scheduler	4 per streaming multiprocessor
Number of warps	64 per streaming multiprocessor
Register file size	256KB per streaming multiprocessor
Shared memory size	96KB per streaming multiprocessor
Unified L1 data/texture cache	4-way, 48KB, 128B line
Number of Texture unit	8 per streaming multiprocessor
L1 instruction cache	4-way, 2KB, 128B line
LLC	8-way, 2MB, 128B line
Memory Model	6 GDDR5 MCs, FR-FCFS, 924 MHz
GDDR5 Timing (in nanoseconds)	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{RCD} = 12$, $t_{RRD} = 6$

Table 3. Stencil Codes Benchmark Suite Description

Benchmark	Source	Domain
fdtd-2D	Polybench [19]	Scientific computation
jacobi-1D	Polybench [19]	Scientific computation
jacobi-2D	Polybench [19]	Scientific computation
hotspot	Rodinia [6]	Scientific computation
hotspot-3D	Rodinia [6]	Scientific computation
stencil	Parboil [62]	Scientific computation
stencil-2D	Shoc [10]	Scientific computation
convolution-2D	Polybench[19]	Machine learning
convolution-3D	Polybench[19]	Machine learning
srad-v1	Rodinia [6]	Image processing
srad-v2	Rodinia [6]	Image processing
bilateral	NVIDIA CUDA SDK [45]	Image processing
image denoising	NVIDIA CUDA SDK [45]	Image processing
sobel	NVIDIA CUDA SDK [45]	Image processing

5.3 Comparison Points

To show the effectiveness of our proposal, we compare the following designs: (1) Standard implementation of the stencil codes using Polybench [19], Rodinia [6], Parboil [62], and NVIDIA CUDA SDK [45] benchmark suites; (2) NeDa (direct nearest-neighbor data exchange among SP cores inside a streaming multiprocessor); (3) NeDa+ (direct nearest-neighbor data exchange among SP cores inside a streaming multiprocessor and from nearest-neighbor streaming multiprocessors); and (4) Ideal case (called Ideal) that provides nearest-neighbor data exchange with no latency overhead.

5.4 Comparison Metrics

We compare different designs in terms of overall GPU IPC and power consumption.

Fig. 13. Total GPU IPC using NeDa, NeDa+, and Ideal normalized to Standard.

6 EVALUATION

6.1 Performance Analysis

Figure 13 shows the relative performance of NeDa, NeDa+, and Ideal implementations, normalized to Standard. We make five observations. First, NeDa improves GPU performance by 21.8% compared to Standard, on average. Second, NeDa's performance is within 13.2% of Ideal implementation. Third, NeDa+ improves GPU performance by 38.2% compared to Standard, on average. Fourth, hotspot has low performance improvement using NeDa, NeDa+, and Ideal implementations, as hotspot is a compute-intensive application. Fifth, image-denoising and sobel have high performance improvement using NeDa, NeDa+, and Ideal implementations, as image-denoising and sobel have high (around 90%) nearest-neighbor shared data accesses (i.e., each point needs all of its nearest-neighbor points). Note that we report the performance of the whole applications. Stencil kernels are the most time-consuming kernels in our evaluated applications.

NeDa improvement on performance is mainly due to three reasons: (1) providing direct data exchange through a light-weight and fast communication infrastructure among nearest-neighbor SP cores in a streaming multiprocessor with register access latency; (2) reducing the number of accesses to the register file and memory subsystem leads to contention reduction in the register file and memory subsystem, greatly reducing their access latency (see Section 6.3 for more detail); and (3) reducing different GPU stalls, including memory stalls by 26%, execution stalls by 21%, and data dependence stalls by 17% (see Section 6.4 for more detail).

6.2 Power Analysis

Figure 14 compares total GPU power consumption of NeDa and NeDa+, normalized to Standard. We make three observations. First, NeDa and NeDa+ reduces the GPU power consumption by 9.1% and 13.4%, compared to Standard, on average, respectively. Second, jacobi-2d has low power reduction, while decreasing on-chip cache connection around 27%, as jacobi-2d reduce the contention of L1D cache much more than the power-hungry register file (for more details, see Section 6.3). Note that the register file power consumption is around 4× higher than L1D cache power consumption and accounts for a big portion of the overall GPU power consumption. Third, sobel has high power reduction using NeDa, and NeDa+, as sobel has the most contention reduction for power-hungry register file among other stencil codes.

Fig. 14. Total GPU power consumption using NeDa, and NeDa+ normalized to Standard.

Fig. 15. Total target memory accesses in NeDa and NeDa+ normalized to Standard.

NeDa improvement on GPU total power consumption is due to reducing the number of accesses to power-hungry register file and memory subsystem; instead, NeDa provides nearest-neighbor data sharing from the local registers in SP cores, which consume lower power consumption. We observe that the dynamic power of the local register is 88% lower than the dynamic power of the register bank. Note that NeDa slightly increases static power consumption due to the added hardware.

6.3 Contention Reduction

6.3.1 On-chip Target Memory. Figure 15 compares total target memory (see Figure 7) accesses of NeDa, and NeDa+, normalized to Standard. We observe that NeDa and NeDa+ reduces the total target memory accesses by 33.4% and 48.3%, compared to Standard, on average, respectively.

6.3.2 Register File. Figure 16 compares total register file accesses of NeDa, and NeDa+, normalized to Standard. We observe that NeDa and NeDa+ reduce the total register file accesses by 14% and 22.4%, compared to Standard, on average, respectively.

6.4 Stall Reduction

Figure 17 compares different GPU stalls of NeDa, and NeDa+, normalized to Standard. We observe that NeDa /NeDa+ reduce different GPU stalls, including data dependence stalls by

Fig. 16. Register file accesses in NeDa and NeDa+ normalized to Standard.

Fig. 17. Different GPU stalls, including (a) dependence stalls, (b) memory stalls, and (c) execution stalls, in NeDa and NeDa+ normalized to Standard.

17%/27% (Figure 17(a)), memory stalls by 26%/39% (Figure 17(b)), and execution stalls by 21%/33% (Figure 17(c)).

6.5 Impact of Batch Size on Performance

There is a tradeoff in identifying global batch size. Larger batch sizes can improve the opportunity of supplying nearest-neighbor pieces of data through neighboring streaming multiprocessors. On the other hand, using larger batch sizes decreases the number of total global batches, and, hence, more likely, we need to switch to scheduling local batches (i.e., NeDa) as there is no active global

Table 4. Bandwidth Numbers of the Inter-SP Network, SP-L1 Data Cache, and SP-shared Memory Paths on the Maxwell-like GPU Architectures

Bandwidth number	Inter-SP network	SP-L1 data cache path	SP-Shared memory path
Theoretical bandwidth	2,048	256	2,047
Measured bandwidth	512	15.7	241

Fig. 18. Total GPU performance while sweeping global batch sizes from 1 to 16, normalized to size 1.

batch. We identify global batch size through performing a sensitivity analysis on global batch size. To this end, we measure the performance of several stencil applications in GPGPU-Sim V3.2.2 [2] while sweeping global batch sizes from 1 to 16. Note that NeDa+ with global batch size 1 works the same as NeDa. We observe that global batch size 4 offers the best performance compared to other global batch sizes. Thus, we use size 4 in NeDa+ experiments.

6.6 Bandwidth Analysis

The theoretical peak bandwidth numbers for the inter-SP network, SP-cache, and SP-shared memory paths are 2,048 B per cycle (there are 128 SP cores per streaming multiprocessor, 4 links per SP core, and 4 B can be transferred through each link per cycle), 256 B per cycle, and 2,047 B per cycle [21] in a Maxwell GTX 980 [47] GPU architecture, respectively (Table 4). To measure the bandwidth utilization for SP-cache, and SP-shared memory paths in stencil codes, we profiled a set of common stencil codes on a Maxwell GTX 980 [47] GPU using a NVPROF [48] profiler. The measured bandwidth numbers for SP-cache and SP-shared memory paths are 15.7 B per cycle and 241 B per cycle in a Maxwell GTX 980 [47] architecture, respectively. The gap between the theoretical peak bandwidth and the measured bandwidth is due to several issues such as memory stalls and bank conflicts. The measured bandwidth for inter-SP network in the stencil codes is 512 B per cycle (there are 128 SP cores per streaming multiprocessor, 1 active link per SP core, and 4 B can be transferred through each link per cycle); this is due to the fact that our network management policy does not allow the simultaneous transfer of data through four links around each SP core. To conclude, NeDa provides higher effective bandwidth in memory-bound stencil codes and, hence, improves the performance of them over the baseline communication mechanism.

6.7 Hardware Overheads

NeDa adds the following resources: (1) a modified warp scheduler (i.e., batch scheduler) per streaming multiprocessor, (2) two 32-bit registers, (3) a 4-to-1 multiplexer, (4) a 2-to-1 multiplexer per SP, and (5) a 32-bit bidirectional link between each two nearest-neighbor SP cores in each streaming multiprocessor (i.e., vertical and horizontal links). In addition, NeDa+ adds a 32-bit bidirectional link between each two nearest-neighbor border SP cores in different streaming multiprocessors, and a coordinator for neighboring batch schedulers, which are in charge of executing a global batch.

To estimate the area overhead of our proposal, we implement NeDa and NeDa+ in the MIAOW [3] open source GPU, and place and route the design on an FPGA. We observe that NeDa and NeDa+ (interconnection network and scheduling mechanism) impose area overhead by 1.3% and 2%, respectively, over the baseline GPU architecture. Note that, NeDa's interconnection network contributes to 96% of the total area overhead.

We synthesize the HDL model of all added resources for NanGate 45nm open cell library using Synopsys Design Compiler to calculate the power consumption of each added resource. We then import the calculated numbers to GPUWattch [34] to measure the power overhead of our proposal. We observe that our mechanism imposes a negligible power overhead, around 0.5%, when it is not used for nearest-neighbor data sharing. Note that NeDa can reduce GPU power consumption when it is employed for nearest-neighbor data sharing (more detail in Section 6.2).

To analyze the overhead of reducing the number of general-purpose architectural registers, we perform a set of experiments using GPGPU-Sim V3.2.2 [2] comparing the performance of the CUDA-based stencil codes with maximum register count equal to 255 and 253 (i.e., the number of general-purpose architectural registers). We observe that there is no performance penalty when reducing the number of general-purpose architectural registers by two. For more information on the benchmarks used, see Section 5.

7 RELATED WORK

To the best of our knowledge, this is the first work to propose a direct inter-core (i.e., SP core) nearest-neighbor data exchange mechanism in GPUs. We briefly discuss prior work.

Core-to-core communication. In this section, we compare NeDa to other core-to-core communication networks proposed for GPUs and CPUs. Several pieces of related work focus on core-to-core communication between streaming multiprocessors in GPUs, mainly to connect L1 data caches [12, 20, 70]. Ibrahim et al. [20] observed that a significant amount of the misses of the L1 data cache required by one streaming multiprocessor can result in a hit in the L1 data caches of other streaming multiprocessors. They proposed to organize the data movement across streaming multiprocessor cores efficiently, to exploit the remote-core bandwidth. However, NeDa is the first work that proposes a point-to-point communication infrastructure between nearest-neighbor SP cores at the local register level in the GPUs.

Several pieces of related work focus on core-to-core communication in general-purpose architectures [11, 27, 59, 60, 75]. Tiler Tile-Gx72 [11] supports explicit core-to-core communication in hardware using register files. In contrast to the hardware Network-on-Chip (NoC) mechanism in Tiler, which network routers are included in each tile, NeDa communication infrastructure is managed by PTX instructions and is dedicated to nearest-neighbor communications. Moreover, the local registers in NeDa have lower latency and power consumption in contrast to the registers in the large main register file of Tiler architecture. Although there is significant previous work on core-to-core communication for multi-core CPUs [11, 59, 60], nearest-neighbor core-to-core (i.e., SP core) communication has not been investigated in GPU designs. To have an efficient nearest-neighbor data exchange mechanism for GPUs, it is essential to optimize the core-to-core communication techniques by thoroughly considering the GPU context, such as the microarchitecture, SIMD execution model, and thread-level parallelism characteristics.

Inter-thread communication. The CUDA programming model [46] features shuffle instructions that provide register-level data exchange between threads within a warp. The existence of warp-limited shuffle instructions in CUDA [46] shows the necessity for inter-warp register-level communication in GPUs. There are some standards that support message passing in multicore systems, such as Message Passing Interface (MPI) [54]. MPI programming model includes point-to-point message passing, one-sided communications, collective communications, and communicator

concepts. NeDa communication model is not as complicated as the MPI, and supports a special case of transfers (i.e., regular read operations from nearest-neighbor SP cores in lock-step) using an interconnection network managed at software-level. As we have already explained in Section 4.2, conflict cannot happen in the NeDa interconnection network; hence, there is no need for an arbitrator.

Additionally, there have been some proposals for hardware message passing and synchronization mechanisms in multicore systems [61, 72]. Voitsechov et al. [68] proposed direct inter-thread communications in Coarse-Grained Reconfigurable Arrays (CGRAs), where intermediate values are communicated directly through the compute fabric. To conclude, they do not introduce an inter data exchange solution for the current GPU architecture. To have an efficient inter-thread nearest-neighbor data exchange mechanism for GPUs, it is essential to optimize data sharing techniques by thoroughly considering the GPU context.

Improving data sharing in GPUs. Works in this section are categorized into two fields. First field aims to increase data re-usage at cache level using various cache management policies (e.g., bypassing [74], buffering [8], and pinning [31]). The Locality Descriptor [65] is primarily designed to convey locality semantics to leverage cache and NUMA locality in GPUs. This mechanism expresses data locality in GPUs, enabling the programmer to better manage GPU memory resources using different techniques such as Cooperative Thread Array (CTA) scheduling, cache management, and memory placement.

Second field uses dedicated memories for safe data sharing in GPUs [67]. Recent research proposed transactional memory [67], which determines the bounds of a critical section using the concept of transaction, decreasing the programming effort required to support explicit synchronization in shared memory. GPU-Local Transactional Memory (LocalTM) [67] proposed a lightweight and efficient transactional memory for GPU shared memory mechanism. NeDa aims to provide a solution for fast data sharing between nearest-neighbor SP cores in GPUs at register level, and this mechanism is not limited to a warp or a thread block. To benefit other forms of sharing (e.g., non-neighbor), each of the above techniques can be used along with NeDa orthogonally. Wang et al. [71] proposed to bring global operations closer to the cores to exploit data sharing in a local manner in GPUs. They adopt software message passing into the GPU for synchronizing between thread blocks by distributing work to the shared memories, particularly for irregular applications. By defining two groups of client and server thread blocks, the client thread blocks send a message to the appropriate server thread blocks whenever they want to access the critical sections. However, the main goal of NeDa is to propose a light-weight communication infrastructure between threads with regular nearest-neighbor access patterns. NeDa does not need a complicated message passing system for synchronization, and supports implicit synchronization between threads in a batch using the proposed batch scheduler. Nearest-neighbor threads in a batch communicate data with each other directly rather than calling server thread blocks.

Locality-aware scheduling. To improve cache locality, in these methods, the threads are categorized into groups, at the level of warps [33, 43] or thread blocks [7, 30, 35], and given a priority. When the execution of one group is stalled, the other category is given priority. This method effectively reduces cache contention and ultimately improves the performance. Two-level round-robin scheduling policy [43] splits all executing warps into fetch groups and prioritizes warps from a single fetch group until they arrive at a stalling point. Then, the next fetch group is chosen. The scheduling policy within a fetch group is round-robin, and switching between fetch groups is done in a round-robin fashion. Their scheduling policy is the closest to ours, but they do not depend on the nearest-neighbor batch forming process and lock-step execution in a batch. In NeDa, the compiler divides the nearest-neighbor warps into batches. The batch scheduler guarantees safe nearest-neighbor data sharing enforcing lock-step execution of threads in a batch. In Ref. [63],

consecutive thread blocks are assigned to the same streaming multiprocessor to effectively use the data locality among them. NeDa inspired this thread block scheduling to enhance nearest-neighbor data sharing between thread blocks.

Efficient register space utilization in GPUs. Works in this section aim to share the physical register file space [25, 73]. In Ref. [25], the authors propose a software-hardware mechanism named Register Mutual Exclusion (RegMutex) to share a subset of physical registers between warps during the GPU kernel execution. RegMutex increases register utilization by sharing the physical register space (has nothing to do with nearest-neighbor data sharing), while NeDa reuses the physical register space efficiently, along with its corresponding data for a group of SP cores residing in a neighborhood window. In Ref. [73], the authors propose Coordinated Register Allocation and Thread-level parallelism (CRAT), a compiler-based performance optimization framework. CRAT first enables effective register allocation. Given a register per-thread limit, CRAT allocates the registers by analyzing the lifetime of variables. After that, CRAT employs a prediction model to find the best tradeoff between the single-thread performance and thread-level parallelism. CRAT has nothing to do with nearest-neighbor data sharing in GPUs, which is the focus of NeDa architecture. To conclude, prior work [25, 73], in register space utilization, aims to share the physical register file space (in the main register file), not the data value, in order to have an efficient register file utilization and increase the thread-level parallelism. However, NeDa aims to share the data values in the local registers (i.e., shared registers are distributed between the SP cores).

Register file caching. Register file caching, in general, caches registers in a small and fast register file cache in order to prevent from accessing slow and power-hungry register files. Register file caching for GPUs is used in several pieces of related work to reduce register file power consumption [15, 28], and enable high-capacity yet long-latency register files [57]. However, register file caching has nothing to do with nearest-neighbor data exchange in GPUs, which is the focus of NeDa architecture. Register file caching has not been used for nearest-neighbor data sharing before. NeDa is the first work that provides nearest-neighbor data sharing through accessing local registers (which cache the shared data value), and filters a fraction of the main register file accesses similar to register file caching in GPUs.

Stencil codes acceleration. Developing high-performance applications for stencil codes have been addressed in similar ways using optimization techniques in various efforts [52, 53]. To the best of our knowledge, there are no microarchitectural optimizations proposed for GPUs to support stencil codes or nearest-neighbor data exchange. Prior GPU work on stencil codes focuses on generating efficient applications to leverage the GPU memory hierarchy. To compare against prior GPU work on stencil codes, we evaluated our architecture against highly optimized implementations of the standard stencil codes from available suites. We believe that NeDa achieves superior performance without introducing complexity, eschewing the need for time-consuming and error-prone optimizations, and simplifying the code greatly (i.e., improving the programming ease) by mitigating the overhead of data exchange. Moreover, prior work on accelerators for stencil codes describes architectures that are incompatible with GPUs [69, 78]. We are first to provide a low-overhead solution that optimizes stencil codes on GPUs.

8 CONCLUSION

We propose NeDa, a light-weight data exchange mechanism for GPUs to effectively satisfy the demand of nearest-neighbor data exchange in stencil codes. NeDa improves the GPU performance and power consumption by (1) reducing the number of memory accesses in stencil codes, and (2) allowing threads to directly communicate through a light-weight communication infrastructure among nearest-neighbor SP cores. We place and route NeDa in an open-source GPU and show a small impact of 1.3% on the area. Our experimental results show an average

performance improvement of 21.8%, and up to 18.3% power consumption reduction compared to standard implementation of stencil codes. Moreover, we believe that NeDa greatly improves the programming ease by mitigating the overhead of data exchange.

REFERENCES

- [1] Homa Aghilinasab, Mohammad Sadrosadati, Mohammad Hossein Samavatian, and Hamid Sarbazi-Azad. 2016. Reducing power consumption of GPGPUs through instruction reordering. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. 356–361.
- [2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE, 163–174.
- [3] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, et al. 2015. Enabling GPGPU low-level hardware explorations with MIAOW: An open-source RTL implementation of a GPGPU. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2 Article 21 (2015).
- [4] Siqi Bao and Albert C. S. Chung. 2018. Multi-scale structured CNN with label consistency for brain MR image segmentation. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization* 6, 1 (2018), 113–117.
- [5] Nanchini Chandramoorthy, Giuseppe Tagliavini, Kevin Irick, Antonio Pullini, Siddharth Advani, Sulaiman Al Habsi, Matthew Cotter, John Sampson, Vijaykrishnan Narayanan, and Luca Benini. 2015. Exploring architectural heterogeneity in intelligent vision systems. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 1–12.
- [6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE, 44–54.
- [7] Li-Jhan Chen, Hsiang-Yun Cheng, Po-Han Wang, and Chia-Lin Yang. 2017. Improving GPGPU performance via cache locality aware thread block scheduling. *IEEE Computer Architecture Letters* 16, 2 (2017), 127–131.
- [8] Hyojin Choi, Jaewoo Ahn, and Wonyong Sung. 2012. Reducing off-chip memory traffic by selective cache management scheme in GPGPUs. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. ACM, 110–119.
- [9] Jason Clemons, Chih-Chi Cheng, Iuri Frosio, Daniel Johnson, and Stephen W. Keckler. 2016. A patch memory system for image processing and computer vision. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–13.
- [10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 63–74.
- [11] Halit Dogan, Masab Ahmad, Jose Joao, and Omer Khan. 2018. Accelerating synchronization in graph analytics using moving compute to data model on Tiler TILE-Gx72. In *Proceedings of the 2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 496–505.
- [12] Saumay Dublisch, Vijay Nagarajan, and Nigel Topham. 2016. Cooperative caching for GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–25.
- [13] Mamdouh F. Fahmy and Omar M. Fahmy. 2018. Efficient bivariate image denoising technique using new orthogonal CWT filter design. *IET Image Processing* 12, 8 (2018), 1354–1360.
- [14] Mrugesh Gajjar and Ismayil Guracar. 2016. Efficient rate conversion filtering on GPUs with shared memory access pattern scrambling. In *Proceedings of the IEEE International Workshop on Signal Processing Systems (SiPS'16)*. IEEE, 285–290.
- [15] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. IEEE, 235–246.
- [16] Michaël Gharbi, Jiawen Chen, Jonathan T. Barron, Samuel W. Hasinoff, and Frédo Durand. 2017. Deep bilateral learning for real-time image enhancement. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 118.
- [17] Claudia I. Gonzalez, Patricia Melin, Juan R. Castro, and Oscar Castillo. 2017. Edge detection methods and filters used on digital image processing. In *Edge Detection Methods Based on Generalized Type-2 Fuzzy Logic*. Springer, 11–16.
- [18] Benjiman L. Goodman, Adam T. Moerschell, and James S. Blomgren. 2017. Texture state cache. US Patent 9,811,875.
- [19] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–10.

- [20] Mohamed Assem Ibrahim, Hongyuan Liu, Onur Kayiran, and Adwait Jog. 2019. Analyzing and leveraging remote-core bandwidth for enhanced performance in GPUs. In *Proceedings of the 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 258–271.
- [21] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*.
- [22] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.
- [23] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified lifting of stencil computations. *ACM SIGPLAN Notices* 51, 6 (2016), 711–726.
- [24] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2014. Managing GPU concurrency in heterogeneous architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, 114–126.
- [25] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. 2018. Regmutex: Inter-warp GPU register time-sharing. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 816–828.
- [26] Hyesoon Kim, Richard Vuduc, Sara Bagsorkhi, Jee Choi, and Wen-mei Hwu. 2012. Performance analysis and tuning for general purpose graphics processing units (GPGPU). *Synthesis Lectures on Computer Architecture* 7, 2 (2012), 1–96.
- [27] Martha Mercaldi Kim, John D. Davis, Mark Oskin, and Todd Austin. 2008. Polymorphic on-chip networks. In *Proceedings of the 2008 International Symposium on Computer Architecture*. IEEE, 101–112.
- [28] John Kloosterman, Jonathan Beaumont, D. Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. 2017. Regless: Just-in-time operand staging for GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 151–164.
- [29] Robin Kobus, Christian Hundt, André Müller, and Bertil Schmidt. 2017. Accelerating metagenomic read classification on CUDA-enabled GPUs. *BMC Bioinformatics* 18, 1 (2017), 1–10.
- [30] Gunjae Koo, Hyeran Jeon, Zhenhong Liu, Nam Sung Kim, and Murali Annavaram. 2018. Cta-aware prefetching and scheduling for GPU. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 137–148.
- [31] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access pattern-aware cache management for improving data utilization in GPU. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 307–319.
- [32] Ahmad Lashgar, Amirali Baniasadi, and Ahmad Khonsari. 2013. Warp size impact in GPUs: Large or small? In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 146–152.
- [33] Shin-Ying Lee, Akhil Arunkumar, and Carole-Jean Wu. 2015. CAWA: Coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 515–527.
- [34] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWatch: Enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 487–498.
- [35] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-aware CTA clustering for modern GPUs. *ACM SIGOPS Operating Systems Review* 51, 2 (2017), 297–311.
- [36] Mostafa Mahmoud, Bojian Zheng, Alberto Delmás Lascorz, Felix Heide, Jonathan Assouline, Paul Boucher, Emmanuel Onzon, and Andreas Moshovos. 2017. IDEAL: Image denoising accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 82–95.
- [37] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing stencil computations for NVIDIA Kepler GPUs. In *Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna*. 89–95.
- [38] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86.
- [39] Andreas Meister and Gunter Saake. 2016. Challenges for a GPU-accelerated dynamic programming approach for join-order optimization. In *GvD*. 86–81.
- [40] Hiroko Midorikawa, Hideyuki Tan, and Toshio Endo. 2014. An evaluation of the potential of flash SSD as large and slow memory for stencil computations. In *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 268–277.
- [41] Nobuhiro Miki, Fumihiko Ino, and Kenichi Hagihara. 2019. PACC: A directive-based programming framework for out-of-core stencil computation on accelerators. *International Journal of High Performance Computing and Networking* 13, 1 (2019), 19–34.
- [42] Amirhossein Mirhosseini, Mohammad Sadrosadati, Behnaz Soltani, Hamid Sarbazi-Azad, and Thomas F. Wenisch. 2017. POSTER: Elastic reconfiguration for heterogeneous NoCs with BiNoCHS. In *Proceedings of the 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 164–165.

- [43] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 308–317.
- [44] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, and Hamid Sarbazi-Azad. 2018. Neda: Supporting direct inter-core neighbor data exchange in GPUs. *IEEE Computer Architecture Letters* 17, 2 (2018), 225–229.
- [45] NVIDIA. 2018. CUDA SDK code samples. Retrieved from https://docs.nvidia.com/cuda/archive/9.1/pdf/CUDA_Samples.pdf.
- [46] NVIDIA. 2018. Cuda9.0 programming guide. Retrieved from <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [47] NVIDIA. 2018. GeForce GTX 980 Whitepaper—NVIDIA File Downloads. Retrieved from https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.
- [48] NVIDIA. 2018. Profiler user guide. Retrieved from <http://docs.nvidia.com/cuda/profiler-users-guide/index.html>.
- [49] Stefan Olsson. 2017. Image processing method for detail enhancement and noise reduction. US Patent 9,595,087.
- [50] Pablo José Pavan, Matheus da Silva Serpa, Victor Martínez, Edson Luiz Padoin, Jairo Panetta, and Philippe O. A. Navaux. 2018. Strategies to improve the performance and energy efficiency of stencil computations for NVIDIA GPUs. In *Proceedings of the 17th Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance 2018)*, Vol. 17. SBC.
- [51] Aaditya Prakash, Nick Moran, Solomon Garber, Antonella Dillo, and James Storer. 2017. Semantic perceptual image compression using deep convolution networks. In *Proceedings of the Data Compression Conference (DCC), 2017*. IEEE, 250–259.
- [52] Prashant Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2019. On optimizing complex stencils on GPUs. In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 641–652.
- [53] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Mahesh Ravishankar, Vinod Grover, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2018. Domain-specific optimization and generation of high-performance GPU code for stencil computations. In *Proceedings of the IEEE* 106, 11 (2018), 1902–1920.
- [54] David Richie, James Ross, Song Park, and Dale Shires. 2015. Threaded MPI programming model for the Epiphany RISC array processor. *Journal of Computational Science* 9 (2015), 94–100.
- [55] Timothy G. Rogers, Daniel R. Johnson, Mike O'Connor, and Stephen W. Keckler. 2015. A variable warp size architecture. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 489–501.
- [56] Mohammad Sadrosadati, Seyed Borna Ehsani, Hajar Falahati, Rachata Ausavarungnirun, Arash Tavakkol, Mojtaba Abae, Lois Orosa, Yaohua Wang, Hamid Sarbazi-Azad, and Onur Mutlu. 2019. ITAP: Idle-time-aware power management for GPU execution units. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 1 (2019), 1–26.
- [57] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. 2018. LTRF: Enabling high-capacity register files for GPUs via hardware/software cooperative register prefetching. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 489–502.
- [58] Mohammad Sadrosadati, Amirhossein Mirhosseini, Shahin Roozkhosh, Hazhir Bakhishi, and Hamid Sarbazi-Azad. 2017. Effective cache bank placement for GPUs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 31–36.
- [59] Pradip Kumar Sahu and Santanu Chattopadhyay. 2013. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture* 59, 1 (2013), 60–76.
- [60] Mostafa S. Sayed, Ahmed Shalaby, Mohamed El-Sayed, and Victor Goulart. 2012. Flexible router architecture for network-on-chip. *Computers & Mathematics with Applications* 64, 5 (2012), 1301–1310.
- [61] Shreesha Srinath, Berkin Ilbeyi, Mingxing Tan, Gai Liu, Zhiru Zhang, and Christopher Batten. 2014. Architectural specialization for inter-iteration loop dependence patterns. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 583–595.
- [62] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).
- [63] Abdulaziz Tabbakh, Murali Annavam, and Xuehai Qian. 2017. Power efficient sharing-aware GPU data management. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 698–707.
- [64] Didem Unat, Xing Cai, and Scott B. Baden. 2011. Mint: Realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the International Conference on Supercomputing*. ACM, 214–224.

- [65] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B. Gibbons, and Onur Mutlu. 2018. The locality descriptor: A holistic cross-layer abstraction to express data locality in GPUs. ISCA.
- [66] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–14.
- [67] Alejandro Villegas, Rafael Asenjo, Angeles Navarro, Oscar Plata, and David Kaeli. 2018. Lightweight hardware transactional memory for GPU scratchpad memory. *IEEE Transactions on Computers* 67, 6 (2018), 816–829.
- [68] Dani Voitsechov and Yoav Etsion. 2018. Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society.
- [69] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2019. Multi-FPGA accelerator architecture for stencil computation exploiting spacial and temporal scalability. *IEEE Access* 7 (2019), 53188–53201.
- [70] Jianfei Wang, Li Jiang, Jing Ke, Xiaoyao Liang, and Naifeng Jing. 2019. A sharing-aware L1. 5D cache for data reuse in GPGPUs. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 388–393.
- [71] Kai Wang, Don Fussell, and Calvin Lin. 2019. Fast fine-grained global synchronization on GPUs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 793–806.
- [72] Yipeng Wang, Ren Wang, Andrew Herdrich, James Tsai, and Yan Solihin. 2016. Caf: Core to core communication acceleration framework. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 351–362.
- [73] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 395–406.
- [74] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, 76–88.
- [75] Payman Zarkesh-Ha, George BP Bezerra, Stephanie Forrest, and Melanie Moses. 2010. Hybrid network on chip (HNoC) local buses with a global mesh architecture. In *Proceedings of the 12th ACM/IEEE International Workshop on System Level Interconnect Prediction*. 9–14.
- [76] Guangwei Zhang and Yinliang Zhao. 2016. Modeling the performance of 2.5 D blocking of 3D stencil code on GPUs. In *IEEE High Performance Extreme Computing Conference, HPEC*.
- [77] Tuowen Zhao, Samuel Williams, Mary Hall, and Hans Johansen. 2018. Delivering performance-portable stencil computations on CPUs and GPUs using Bricks. In *Proceedings of the 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 59–70.
- [78] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 153–162.

Received March 2020; revised September 2020; accepted October 2020