# TaskStream: Accelerating Task-Parallel Workloads by Recovering Program Structure

Vidushi Dadu
vidushi.dadu@cs.ucla.edu
University of California, Los Angeles
USA

Tony Nowatzki
tjn@cs.ucla.edu
University of California, Los Angeles
USA

## ABSTRACT

Reconfigurable accelerators, like CGRAs and dataflow architectures, have come to prominence for addressing data-processing problems. However, they are largely limited to workloads with regular parallelism, precluding their applicability to prevalent task-parallel workloads. Reconfigurable architectures and task parallelism seem to be at odds, as the former requires repetitive and simple program structure, and the latter breaks program structure to create small, individually scheduled program units.

Our insight is that if tasks and their potential for communication structure are first-class primitives in the hardware, it is possible to recover program structure with extremely low overhead. We propose a task execution model for accelerators called TaskStream, which annotates task dependences with information sufficient to recover inter-task structure. TaskStream enables work-aware load balancing, recovery of pipelined inter-task dependences, and recovery of inter-task read sharing through multicasting.

We apply TaskStream to a reconfigurable dataflow architecture, creating a seamless hierarchical dataflow model for task-parallel workloads. We compare our accelerator, Delta, with an equivalent static-parallel design. Overall, we find that our execution model can improve performance by 2.2× with only 3.6% area overhead, while alleviating the programming burden of managing task distribution.

## CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; **Data flow architectures**; *Heterogeneous (hybrid) systems.*

## KEYWORDS

Irregularity, tasks, load-balance, accelerators, generality, dataflow, reconfigurable, streaming

**ACM Reference Format:**
Vidushi Dadu, Tony Nowatzki. 2022. TaskStream: Accelerating Task-Parallel Workloads by Recovering Program Structure. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507706



*ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland*

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507706

## 1 INTRODUCTION

With improvements to general purpose processors slowing, reconfigurable accelerators (aka. dataflow accelerators [30, 41, 43, 45, 61–63], or CGRAs [16, 17, 26, 34, 35, 57, 60]) have become an increasingly favorable option for meeting the needs of data-processing workloads. Recently, multicore versions of these designs have seen commercial traction, particularly for use in datacenters (e.g. [4, 5, 39, 68, 69]). The key attraction is their broad applicability across workloads, due to their general computation datapaths and memory access patterns.

While promising for generality, most reconfigurable architectures remain fairly limited to regular computations. While many forms of irregularity can be supported[1], we focus on irregular parallelism, also known as *task parallelism*: this occurs when a program's work is created and scheduled to execution resources dynamically, based on runtime computations.

There are at least three clear benefits to supporting task parallelism in reconfigurable accelerators. First, many workloads have inherent data-dependences in forming parallel work (e.g. create tasks for all outgoing edges of a graph's vertex), so this enables broader applicability. Second, sometimes the amount of work per task can only be determined at runtime (e.g. number of elements matched in a join), so dynamic assignment can balance load. Third, many irregular workloads have multiple task types, where each type stresses the system differently in terms of compute, memory, network, or other resources (e.g. memory-bound graph aggregation and compute-bound multiplication in Graph Convolution Networks - GCNs); running different task types in parallel can balance shared resource usage.

Yet task parallelism is generally not supported on reconfigurable accelerators, and instead, resources are assigned at coarse grain. This is understandable, as such architectures are designed to exploit *structure* – i.e. patterns in the execution that occur through time (temporal structure) or across units simultaneously (spatial structure). For example, configuring a dedicated communication path between two instructions exploits the temporal structure of repeated communication; a vector operation exploits spatial structure. However, because tasks are traditionally assigned to resources independently, the structure across tasks is lost, leading to severe overheads: naïvely assigning tasks to cores would cause excess reconfiguration; relying on traditional shared-memory inter-task communication incurs high latency and traffic overheads.

Our goal is to enable efficient task parallelism on reconfigurable accelerators. In principle, the information required to optimize task-parallel programs to avoid inter-task communication could be

[1] E.g. supporting irregular memory [19, 48], or irregular control [62, 67].

made available: tasks could be annotated with information that describes which data they use, and the hardware could take advantage of structured communication patterns, like pipelining and multicasting. Performing this analysis in software would likely not be profitable, especially in an accelerator system where tasks are short. Our solution is to expose task-management and structured-access as first-class primitives of the hardware's execution model.

**TaskStream Model:** With these insights, we propose the TaskStream execution model, which augments a reconfigurable accelerator's ISA with primitives for dynamic task management and structured access. In TaskStream, tasks and their dependences are represented in a graph, and edges are annotated with opportunities for structure recovery. Overall, TaskStream provides reconfigurable accelerators with efficient support for tasks through three key features: First, it provides a high-throughput task creation interface from instructions in the datapath. Second, it provides load balance support, informed by programmer's knowledge of task type distribution and task size (work-per-task). Third, it supports dynamic data streaming to recover inter-task communication structure, and dynamic task batching of reused inputs to recover temporal and spatial locality structure.

**Implementation and Evaluation:** To implement TaskStream, we augment a decoupled-spatial reconfigurable accelerator [66] with TaskStream abstractions in software and hardware (called Delta). Programmers express their programs with a hierarchical dataflow representation, where TaskStream nodes are coarse grain tasks, each containing dataflow instructions that define the task semantics. We evaluate performance with a cycle-level simulator.

We chose five challenging task-parallel workloads with unique opportunities for structured communication: K-nearest neighbor (kNN), an ML-oriented database query, sparse matrix-multiply, Cholesky decomposition and Graph Convolution Networks (GCN).

Overall, we achieve 81.3× speedup over multicore CPUs. Over an efficient static-parallel CGRA baseline (without TaskStream), we achieve 2.2× speedup, with load-balancing optimizations alone yielding only 1.3× performance.

**Contributions:**

- Novel model for task parallelism for reconfigurable accelerators, which enables a new class of structure-recovery optimizations.
- Hardware/software co-design to support three different classes of communication patterns efficiently: 1. lightweight task creation, 2. inter-task streaming dependences with co-scheduling of dependent tasks, 3. inter-task reuse by exploiting spatial or temporal structure with batching.
- Evaluation against static and naïve task-parallel models, demonstrating the value of structure recovery in irregular workloads.

**Paper Organization:** We first motivate and describe TaskStream in Section 2. In Section 3, we apply TaskStream to a dataflow accelerator to create a hierarchical dataflow representation, describe the mapping of our evaluated workloads, and discuss limitations and possible extensions. Section 4 describes the accelerator hardware implementation, and then Sections 5 and 6 provide methodology and evaluation. Finally, section 7 describes the relationship to existing software and hardware systems that optimize for task locality.

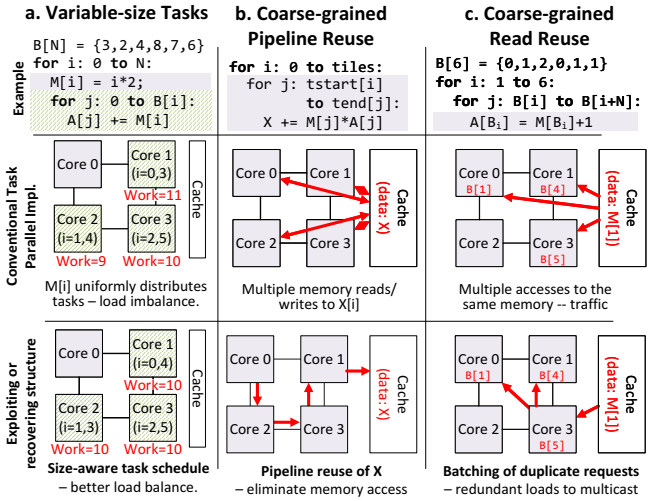


**Figure 1: Opportunities in Naïve Task Parallelism**

## 2 TASKSTREAM EXECUTION MODEL

We first motivate the principles of TaskStream by discussing opportunities to exploit certain forms of program structure. Then, we will present our proposed TaskStream model, abstract from any particular architecture implementation.

### 2.1 Opportunities for Structure Recovery

The context for our proposed system is a tiled multicore architecture, in which a task scheduler assigns tasks to cores. We assume a mesh-based network on chip (NoC), but optimizations apply to other topologies. To elaborate our optimizations, we discuss three program idioms from Figure 1, where locality structure is lost due to exploiting task parallelism.

**Variable-sized Tasks:** A variety of task-parallel workloads have task types[2] whose amount of work is either data-dependent or progressively changing over its instances. Figure 1a shows an example where inner loop tasks have a data-dependent length, based on B[i]. A naive task parallel model would assign the inner loop tasks irrespective of the work involved in a task. Work-stealing is possible, but requires extra inter-core communication latency and bandwidth.

The opportunity here is to distribute tasks with the knowledge of the work involved. In the example, core 1 gets the smallest and second-largest task (i.e. with total work = 3+7 = 10), so that all cores get similar total work. This model is synergistic with accelerators, which have quite predictable execution times.

**Coarse-grain Pipeline Reuse:** A common behavior in data processing algorithms is ordered dependences between one task and another, for example where one task produces an array which the other uses in the same order. Figure 1b demonstrates a global reduction example where each core gets a tile of data. In the NSAïve task parallel implementation, all cores need to perform updates on the reduction variable through memory.

[2]A "task type" is the static definition of a task, including computation and memory accesses, while the dynamic instantiation of a task is a task instance.

The opportunity here is to identify the ordered reuse, and pipeline or *stream* the data from a producer to one or more consumer tasks. This transforms the memory traffic into direct network traffic, reduces shared-memory overhead from coherence, and also allows overlapped execution of tasks for more concurrency. In the example, the pipelined reduction can be performed without accessing memory (except for writing the final value).

**Coarse-grain Read Reuse:** Another common idiom is when different subsets of tasks read the same data. If such tasks are not scheduled together in time or space, the opportunity to exploit this form of reuse can be lost. Figure 1c demonstrates this with an algorithm that traverses and modifies a compressed sparse row (CSR)-like data structure, and is representative of common algorithms that rely on range-based indirection. Here the duplicates in B are expected to create multiple tasks with shared read data, providing an opportunity for reuse. A naïve task parallel model schedules tasks without respecting locality, so tasks that access the same data may not be scheduled on the same core or at the same time. The reuse cannot be exploited to save network traffic and cache/memory bandwidth.

Such coarse-grain reuse can be exploited by identifying tasks that access the same data, and reordering them to execute at the same time on different cores; the responses can then be multicast to significantly reduce network traffic and memory bandwidth usage. We call this optimization *task batching*.

An alternate opportunity, used in a variety of other contexts [6, 12, 18, 25, 27, 74], is to use a "spatial hint" to assign tasks that access the same data to the same cores. While this reduces memory access for data that fits in private cache, it also restricts the allowed scheduling locations, which could restrict load balancing optimizations. We compare against `spatialhint` in evaluation.

## 2.2 TaskStream Model

TaskStream is a task-parallel execution model that adds sufficient information to identify and exploit structure-recovery opportunities. In principle, TaskStream can be an extension for a variety of architectures, but its simplicity and approach to load balance makes it well-suited to reconfigurable accelerators (Section 3 discusses integration with a dataflow-based accelerator model). We first discuss the basics, and then cover how each optimization is applied.

**TaskStream Basics:** A program in TaskStream is represented as a set of nodes, one for each task type, and edges for inter-task dependences. Edges are typed, and each type indicates the potential for structure-recovery: creation (standard), streaming (for pipeline reuse), and batching (read reuse). Tasks can be in one of three states: *1. Created:* the arguments for a task instance are constructed on the originating core; *2. Scheduled:* the task is bound to execution resources, *3. Executing:* task computation is in progress.

Tasks are created when they receive a set of values for all incoming creation (standard) edges. Next, a task is scheduled to storage/execution resources (e.g. buffer/core), after which it is assigned a TaskID that represents this location; the TaskID may be returned to the parent if a streaming communication will be established. Tasks may only be scheduled to a core which is configured for its task type. To convey the configuration information, each task node is annotated with a `coreMask`: a bitmap that describes the legal

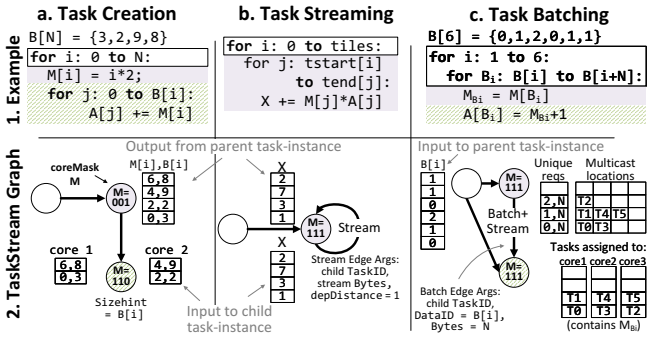


**Figure 2: TaskStream Graph Abstractions**

mapping locations. Some task types can be co-located on the same core, provided there are sufficient resources. Tasks that are not yet *ready* to execute may be waiting on streaming or batched data, and we call these tasks *pending*.

One phase of the program completes when all tasks are completed. A program may consist of multiple phases. Examples of a TaskStream program phase are shown in Figure 2, where task types are distinguished by color (and shading), and we discuss next.

**Task Creation & Work-aware Load Balance:** Figure 2a demonstrates the basics, as well as annotations for load balancing. A single task creation edge connects the outer-loop multiplication task, and inner-loop accumulation. When two task nodes are connected by a *task creation* edge, it means that some outputs from the source node task are used to activate the creation edge and will be inputs to the destination node task. In Figure 2, the interface for activations is represented by task buffers, and the data order indicates the order of producing and receiving tasks. The outer-loop task gets core 1 (`coreMask:001`) while accumulation gets the remaining 2 cores (`coreMask:110`) because it has a ratio of `B[i]` times more work compared to the outer-loop.

TaskStream provides annotations to aid load balancing at task scheduling time. Task creation edges may be annotated with a `sizehint`, which is a task argument that describes the relative amount of work for the task. This enables a simple size-aware scheduling policy, where a new task is assigned to the core with the least cumulative work. In the example, `B[i]` is the number of iterations of the inner loop, and is therefore used for that task's `sizehint`. The scheduler could then assign tasks of size 3, 8 to core 1 and tasks of size 2, 9 to core 2, resulting in a balanced load.

**Task Streaming:** To facilitate dynamic pipelining between tasks, edges may be of *task streaming* type. When two nodes are connected by such an edge, the output at the source node task triggers a streaming communication with the assigned children (stored as child `TaskID`). For a streaming edge, the programmer can specify a dependence distance (`depDistance`), which allows developing a streaming relationship between task-instances separated by a fixed number of tasks. In the example, the `depDistance` is 1. To close the communication, an *end-of-stream* message is sent when the required number of `bytes` have been streamed in, this parameter must be specified by the producer. To set up the communication, *start-of-stream* handshaking messages are exchanged to ensure that the children are ready, and producers have their scheduling information. The data is streamed in between these messages.

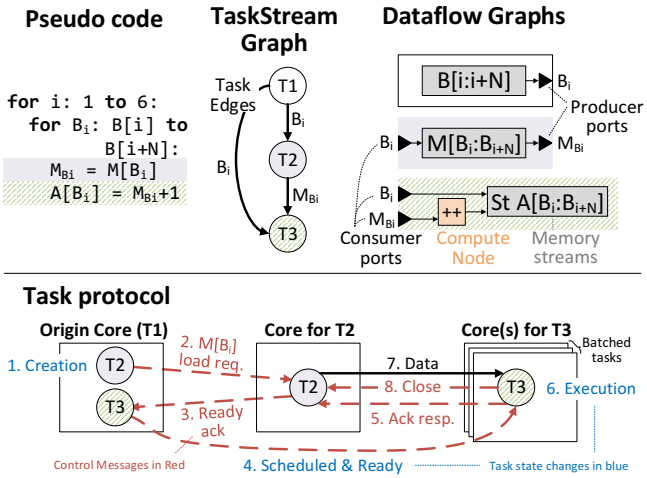


**Figure 3: TaskStream + Dataflow** (T2 state changes omitted.)

Figure 2b demonstrates the task streaming edge: here the dependent instances of the reduction task type are scheduled in mutually exclusive locations – this is required to ensure that tasks involved in streaming are concurrently scheduled. When data is available at the parent task, the *start-of-stream* message is sent to the destination node and data will be streamed in. When finished, *end-of-stream* messages will free resources.

**Task Batching:** To enable multicasting of shared reads, we implement a *task batching edge*. This edge requires three parameters when it is activated: `DataID` indicates whether the reads are to the same data, `TaskID` indicates the dependent dynamic task, and `bytes` indicates the length of these reads. The task scheduler can use this information to record which tasks are dependent on the same reads, and reorder them to schedule them together. The advantage is that data can be multicast to all co-scheduled tasks.

Figure 2c demonstrates the task batching edge: here we split the program into a CSR-traversal task and an addition task type. The outputs from `B[i]` are batched, resulting in only 3 unique requests instead of 6. For each unique request, the TaskIDs of the corresponding batched tasks are shown in the "Multicast locations" table. We are able to perform this reordering by exploiting the fact that the addition tasks are commutative and can be executed in any order without affecting correctness.

## 3 TASKSTREAM FOR RECONFIGURABLE ACCELERATORS

Applying TaskStream to a reconfigurable accelerator naturally creates a hierarchical-dataflow representation: one higher-level dataflow of task management and communication, and one lower-level dataflow of instruction execution. Here we describe the integrated abstractions of such a representation, the process of programming, and mapping of evaluated workloads. Finally, we discuss the limitations and possible extensions of our programming model.

### 3.1 Hierarchical TaskStream Dataflow

**Decoupled Dataflow Background:** We integrate TaskStream with a decoupled-dataflow model for accelerators (e.g. [18, 19, 30, 41, 45, 62]). We provide a simplified description here.

In this model, computation and memory instructions are represented as nodes, edges represent *ordered* dependences between instructions, and nodes fire on data arrival. Memory nodes are referred to as *memory streams*, as they encode patterns of memory access (e.g. A[i] for i = 0 to N). Computation is decoupled from memory to allow for efficient prefetching, and memory and compute nodes communicate with each other through *"ports"*. Ports are analogous to registers in a VonNeumann architecture, but have FIFO semantics. Instruction nodes and ports are scheduled to dedicated hardware resources by the compiler.

To execute a computation, the memory and compute nodes are first configured. Execution begins when inputs are supplied through ports, which can be consumed by memory nodes as parameters of streams, or as inputs to the computation nodes.

**Hierarchical Integration:** TaskStream can be integrated on top of dataflow abstractions: each task is represented as a set of instruction-level dataflow nodes, and edges between tasks carry TaskStream semantics (i.e. creation, streaming, batching). Figure 3 depicts this hierarchical approach for an example program.

When a task is scheduled, its inputs are delivered to corresponding ports, which triggers instruction-level execution. Tasks are then executed in pipelined fashion in the order they are scheduled. The instructions and memory prefetching of different tasks is interleaved, which is possible because of the tight composition of the instruction and task-level dataflow. Streaming inter-task communication also uses ports (shown as triangles in Figure 3); producer ports at the parent deliver data to the consumer ports at the child task. A consumer port must be "acquired" by a parent task before communication can begin, as explained later.

Figure 3 shows an example streaming communication, where the output port from the *T1* task, `B[i]`, can trigger the `M[j]` load and the addition task *(T2* and *T3)*. The task arguments can be reordered along the TaskStream edge using the spatial and temporal scheduling policy, and hardware resource limitations will limit the maximum reordering distance.

**Task Protocol:** Figure 3 also shows the task protocol for the example, demonstrating all three task operations: scheduling, streaming and batching. We refer to the figure as we detail the protocol.

*Task Protocol – Scheduling:* After a task is created, it is scheduled both spatially and temporally (step 1 and 2 in Figure 3). For spatial scheduling, TaskStream checks whether any task argument is annotated with `sizehint`, and the task is sent to the core with the least cumulative work until now, and this value is incremented. If no argument is annotated as `sizehint`, round-robin ordering is used. To minimize the response traffic, tasks which only access memory (e.g. *T2* in Figure 3) are scheduled differently; they are instead scheduled where the data is located (e.g. by determining the shared cache bank of the start address).

Task instances (identified by their arguments) may either be held in a ready state if all arguments are available, or in a pending state otherwise. For example, in Figure 3, the *T2* task is ready after receiving `B[i]`, however the *T3* task will be in the pending state, as it is still waiting on `M[j]`. In the pending case, the producer must provide an explicit "acknowledgment" (ack) of data readiness to trigger the task.

*Task Protocol – Streaming:* Whenever there is data at the producer port of a task streaming edge, an ack is sent to the child tasks along with the producer port information (Figure 3, step 3). This ack should trigger a check as to whether the child task can be concurrently scheduled; this requires that the current task has finished and the consumer port is free. When both conditions are met: the child task is set ready and scheduled (Figure 3, step 4), the consumer port is set busy (i.e. it is acquired), and the ack response is sent back to start streaming (Figure 3, step 4, 5). After the last data is sent, the producer sends another ack to close the communication and free the remote port (Figure 3, step 8).

*Task Protocol – Batching:* Batched tasks will be held temporarily, and those identified to have the same `DataID` will be scheduled simultaneously across multiple cores; the responses of batched requests are multicast to all co-scheduled tasks. Batched requests also supply a `bytes` argument that indicates the length of the data to be streamed. In Figure 3, step 3, ready acks are sent to all batched tasks, which will then be scheduled on different cores (step 4). Then, after the ack response is received from all cores and the tasks are set for execution (step 5, 6), data will be multicast in step 7, before the communication is closed (step 8).

**Deadlock Prevention:** During streaming inter-task communication, the ports involved must be acquired before data is sent, and held until the stream is complete. This ensures that data for multiple streams is not interleaved. Port acquisition has deadlock hazards, which we describe, along with solutions, as follows:

*Self-loop in the TaskStream Graph:* Consider the scenario when the parent and child tasks, setup for streaming communication, are scheduled for execution on the same core. The child task may never be able to lock the consumer port if the parent is already using it, and the parent cannot release the producer port until the streaming data is sent to the child, as it is waiting for the child to get lock of the consumer port. Our solution is to allocate a mutually exclusive set of resources/cores to the parent and child tasks. More specifically, in the case that streaming exists within tasks of the same type, tasks can form a dependence chain. We create virtual partitions of the cores to divide the cores into a number of sets equal to a maximum dependence-chain length. A child is always scheduled to a different virtual partition than any of its parents.

*Capturing multiple ports for multicast:* For multicast, the parent task needs to acquire multiple ports, one for each core. Here we break the possibility of cyclic deadlock by ensuring the ports are acquired in the order of core ID (a unique ID assigned to each core).

## 3.2 Programming

One of the key advantages of programming in TaskStream is that it manages task scheduling at high performance, with only modest programmer help. The process to port a C/C++ workload to TaskStream programming model involves three steps: 1. Defining task types and their functionality, 2. Determining the dependencies among task types to form a task graph, and 3. Managing the start and stop of a program phase. We will discuss each of these in detail below, using Cholesky as an example, as depicted in Figure 4.

**Defining Task Types:** When porting a program, a code region should be assigned to a new task type if it performs different computations, the same computations at a different rate, or has different

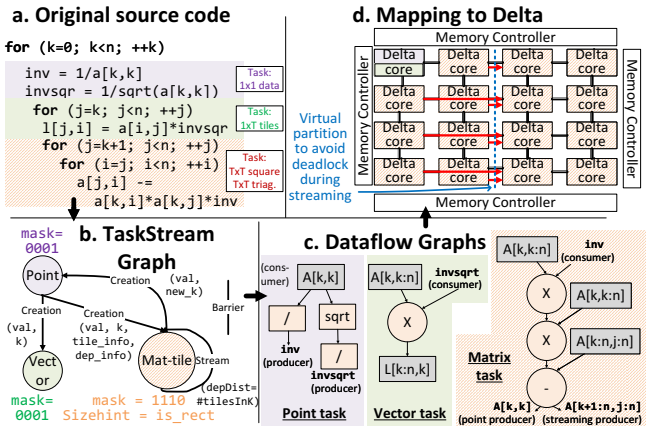


**Figure 4: Cholesky Implemented in TaskStream** (for brevity, only two outer loop iters. run in parallel in one program phase)

**Table 1: Node properties in Task Graph**

| | Description of Node Property |
|---|---|
| coreMask | Bitmask indicating cores a task may be assigned to. |
| sizehint | Task argument that indicates relative task length. |
| spatialhint | Task argument indicating a preferred core (for locality) |

locality behavior. For example, two computations may either be combined into a single task so that the task granularity is higher or may be split if the data associated with two computations are not expected to have similar locality behavior.

For each task type, the programmer first defines the instruction-level dataflow graphs. The programmer assigns certain cores to that task type using the `coreMask`, and may also define a task argument as a `sizehint` or `spatialhint`. The node characteristics are as defined in Table 1.

For the example in Figure 4a, Cholesky has three task types, one for each loop nesting degree: 1. *Point:* performs only one inverse and square root for every outer loop iteration. 2. *Vector:* performs O(n) multiplication operations. 3. *Matrix:* performs O(n×n) multiplication and subtraction operations. Since the work required for *Point* and *Vector* is much smaller than *Matrix*, the `coreMask` is set to assign one core to both *Point* and *Vector*, while all other cores are assigned to the matrix task. Figure 4d shows the mapping of Cholesky to our accelerator, called Delta, which will be explained in the next section.

Task granularity is also a significant choice. Smaller tasks may suffer task management overheads, while larger ones are more difficult to load balance. If the task-size distribution is too wide, even the `size-hint` optimization may not be sufficient. Cholesky is challenging to tile into tasks, because the iteration domain is triangular. Therefore, we split the task into square and triangle tiles (along the diagonal); these are still of relatively different sizes. The new task types are *Point*, *Vector-tile*, and *Matrix-tile*. The tile information is passed as arguments on the *Point*-to-*Matrix* edge.

**Defining the TaskStream Graph:** Next, the programmer uses algorithmic knowledge to identify edges among task type nodes.

**Table 2: Edge properties in Task Graph**

| | | Description of Edge Property |
|---|---|---|
| Program Exposed | Edge_type | Either Creation, Streaming, or Batching |
| | Producer & consumer ports | Interfaces for TaskStream I/O |
| Program Hints | depDistance | By comparing the distance (port), it identifies a task as parent/child. |
| | DataID | ID used to batch shared-read data/requests |
| | Bytes | Either used as size for streaming or meta-data for batching |
| Hardware Managed | TaskID | Stores location where a task is buffered |
| | Ack | Ack buffer maintains TaskIDs of tasks whose ready signal is waiting to be served |
| | Sched | Stores TaskIDs of child tasks, as identified using depDistance |
| | SchedParent | Stores TaskIDs of parent tasks, as identified using depDistance |

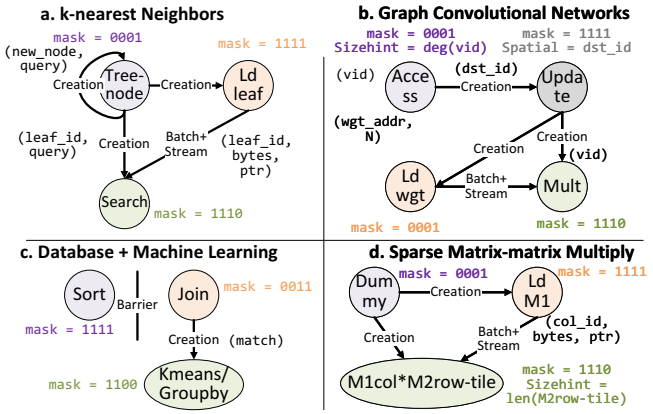


**Figure 5: TaskStream Graphs for Evaluated Workloads**

These include determining whether any task computation is triggering another computation (task creation), whether the tasks have pipelined reuse among them, and whether there is shared read-data among tasks. Another important component is to decide the task arguments for a type and then create an edge interface from producer ports at the source task to consumer ports at the destination task. The list of supported edge characteristics is defined in Table 2; we list hardware managed aspects as well, to make it clear what the programmer needs to reason about.

In the Cholesky example in Figure 4c, there is a creation edge from *Point* to *Vector* and *Matrix-tile*. There exists data dependencies among multiple matrix tasks, hence there is a streaming edge from the *Matrix-tile* task to itself. The dependence distance is the number of matrix-tiles in the $k^{th}$ iteration of the outermost loop.

Finally, we need to limit the number of recursive tasks created by the self-loop in *Matrix-tile* task – this is done by setting the maximum dependence-chain length, determined based on hardware resource limitations (the sensitivity to this length is studied in Section 6). The program phase will end, shown as "barrier" in the figure, after all dynamic tasks are complete.

**Managing a Program Phase:** A program phase starts when the programmer pushes an explicit task of any type. The phase is complete when all tasks have finished execution. Cholesky is initiated by creating a task for the outer-loop point task.

## 3.3 Workload Mapping

Here we discuss how we implemented each of the four additional evaluated workloads. Figure 5 shows examples (with only a `coreMask` for 4 cores, for simplicity).

**k-nearest neighbors:** Figure 5a shows the TaskStream graph for kNN search. For every query, a binary kd-tree is searched. When the leaf node is reached, data associated with the leaf is accessed to perform linear search (similar to Tigris [71]). We define two task types: 1. *Tree node:* A tree traversal. Since each traversal will incur long latencies to access pointers, we split it into small tasks that compares with the current node and outputs the next tree node. 2. *Leaf search:* Here the query is searched linearly in a long vector associated with the leaf. Many queries may search in the same leaf, generating coarse-grained reuse. Hence, we separate the leaf load task and add a Batch+streaming edge to the leaf search task.

**Graph Convolution Network (GCN):** Figure 5b shows the TaskStream graph for GCN. Every vertex accumulates its feature vectors into its outgoing neighbors, and when all the incoming feature vectors are received, the accumulated vector is multiplied with a weight matrix. To enable flexible load distribution, we define three task types: graph access, feature vector updates (together performing aggregation) and matrix-vector multiplication. As updates involve irregular accesses, we use *spatialhint* based scheduling for atomic updates to ensure that remote accesses are minimized. For accesses to the vertices with varying degrees, we use `sizehint` based scheduling. For matrix-vector multiplication, different graph vertices access the common weight matrix, creating an opportunity for batching reuse (shown by the weight load task and the Batch+Streaming edge). We store the weight matrix in the private scratchpad, and multicast from there.

**Database + Machine Learning:** Figure 5c shows the TaskStream for a Database/Machine Learning kernel from Gorgon [62], specifically query 2. We define three task types: 1. *Sort:* Sort requires sufficient work to utilize all resources. Moreover, it reuses its outputs as inputs to the next iteration of sort, and therefore, its output cannot form pipelined communication with other dependent task types. Hence, we treat the subsequent computation as another phase, executed after a task barrier. 2. *Join:* requires O(n) comparison operations. 3. *kmeans-groupby:* requires O(#matched-rows*d) operations to find the minimum distance. Since the work required among *Join* and *kmeans-groupby* may be highly different depending on the number of matched rows and the number of dimensions, we assign 2 cores to *Join* and the rest to the *kmeans-groupby* task type. These tasks are connected by the creation edge.

**Sparse matrix multiply:** Figure 5d shows the TaskStream for sparse matrix-sparse matrix multiply. Here we use a tiled outer product implementation (similar to SCNN [44]). We define a task type as the product of a column of matrix 1 and a tile of matrix 2's corresponding row. Since different tiles of matrix 2's row access a common column, there is an opportunity of batching reuse (shown by "Ld M1" and the Batch+Streaming edge). Moreover, we store copies of matrix 2's row and partial sums in the private scratchpad, and use *spatialhint* based scheduling to ensure that a task is scheduled where its data is stored.

### 3.4 Discussion of Limitations and Extensions

**Our Programming Experience:** After identifying tasks and dependences, writing TaskStream code is not overly complex. For programming TaskStream, we use a unified graph domain-specific language for both the TaskStream and dataflow graphs. For reference, tiled-Cholesky on a static-parallel accelerator (REVEL [67]) is 163 lines of code, and the TaskStream version is 210 lines.

**Adapting Task-parallel Programs:** While we focus on specialized implementations, it is somewhat straightforward to adapt programs from languages with fork-join parallelism like Cilk [10]. The essential idea is that whenever a child synchronizes with a parent task using backward dependence in Cilk, in TaskStream, the parent-task can create a successor task, and the child tasks will now have a forward dependence to the successor task. This is possible because TaskStream allows *pending* tasks, where tasks are waiting on arguments from dependent tasks. The child tasks can communicate with the successor task using `TaskID`. Because this is implemented in hardware, it puts a limit on the number of in-flight tasks. In addition, various dataflow-inspired programming models [23, 28, 52, 59, 70] also rely on static inter-task dependencies. These could be a natural fit for expressing TaskStream programs.

**A Hypothetical Compiler:** It is future work to automate the intuition above to construct a high-level-language compiler. In addition to the above, such a compiler would need to identify some program structure to apply the optimizations we consider: The `sizehint` could be determined by estimating loop trip counts and instruction counts. The `coreMask` can be determined by balancing load, based on the relative ratio of average per-task work, and this could be calculated similarly. For structure-recovery edges with simple nested-loop programs, loop dependence analysis could be sufficient (e.g. Cholesky). Workloads with dynamic dependencies, like sparse-matrix-multiply and kNN, may require programmer help (e.g. a loop annotation indicating dependencies).

## 4 DELTA: A TASKSTREAM ACCELERATOR

Delta is our proposed multicore accelerator, which implements the TaskStream execution model. Delta tiles are interconnected with a mesh-based NoC, and Figure 6 overviews a single tile.

The computation unit is a coarse-grained reconfigurable array, connected via hardware ports (vector ports). The stream controller generates memory requests from stream access patterns, and the responses are sent to the port interface for further communication.

The novel aspect of the hardware is for task management, particularly the task-creation unit (used for storing arguments for ready and pending tasks), and the task-batching unit (used for detecting and scheduling tasks that read the same data). We next describe the design and operation of these components.

**Memory Hierarchy:** Each core has a small private scratchpad, and all cores share access to a shared, distributed on-chip cache. To explain the rationale for this design, we consider the three predominant forms of reuse for memory access: 1. Small, read-only data shared among tasks: This data should be cached on-chip during the entire algorithm. Therefore, each core has a small private scratchpad. 2. Shared data across a subset of tasks: Here the use of scratchpad would require software coherence, and would be difficult to manage. Therefore, Delta has a shared cache, and the reuse across tasks is exploited using task batching. 3. Streaming data with no reuse: This data can bypass the cache hierarchy and will be directly streamed from memory.

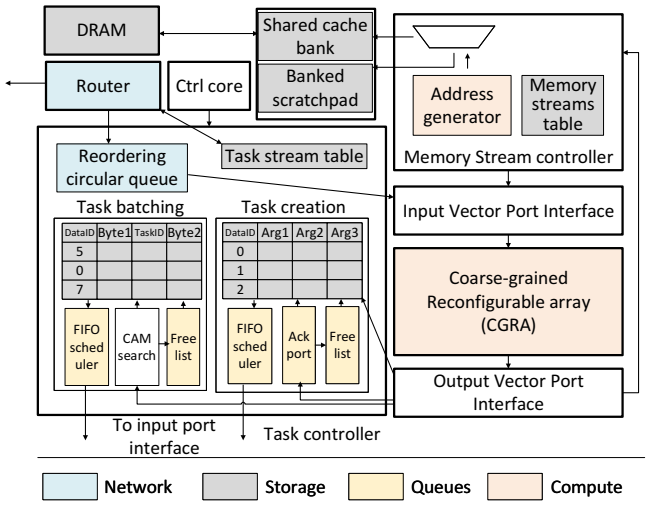


**Figure 6: Single Tile of Delta Accelerator**

**Memory Stream Controller:** The memory stream controller is designed to generate memory addresses using the access patterns and size determined by task type and arguments. Along with an address generator, the memory stream controller also has a stream table that holds their running state. For each stream in the task dataflow program, we reserve some number of entries in this stream table so that forward progress can always be guaranteed (otherwise streams for a single port could fully occupy the table, and streams for another port could not be scheduled).

**Task Creation Unit:** This unit holds tasks until all of their arguments have been received. This unit includes a free list, task argument buffer, dedicated acknowledgment buffer, and a FIFO scheduler (see Figure 6). The task argument buffer is a simple SRAM memory that stores task arguments sequentially, and the free list queue maintains free entries in this buffer.

When data at producer ports is available, if the free list has space, the arguments are pushed in the task creation buffer, and a unique `TaskID` is assigned (using current core and task buffer location information). When a task is ready (i.e. does not require an explicit acknowledgment or already has one), the address location of the current task (`TaskID`) is pushed to the FIFO scheduler. When all consumer ports have sufficient space, the FIFO scheduler will release the task arguments to the input ports in the given order. At that time, the resulting entry will be pushed into the free list.

**Task Batch Buffer:** This unit enables dynamic batching of tasks. Similarly to the creation unit, this unit also has a free list, batching buffer, a small CAM and a FIFO scheduler. The batching buffer is a banked SRAM memory; the difference here is that the free list maintains `TaskIDs` of a "batch" of task arguments instead of just one, as in the task creation unit. Also, here all the task arguments are annotated, which includes `bytes` and `TaskID` of the dependent tasks, as shown in Figure 6. When the data at the producer ports

**Table 3: Datasets Used in this Work**

| Workload | Dataset-size | Workload Parameters |
|---|---|---|
| **kNN** | Queries=512<br>Queries=1024<br>Queries=2048 | kd-tree-depth=8<br>leaf-size=2048 |
| **SpMM** | M1=M2=512x512<br>M1=M2=1k*1k<br>M1=M2=4k*4k | density=0.10 |
| **DB-ML** | T1=T2=10M<br>T1=T2=15M | join=0.10 |
| **Cholesky** | N=128<br>N=256 | tile-size=32 |
| **GCN** | cora (V=2708, E=10556)<br>citeseer (V=3243, E=4536) | feat-len=64 |

is ready, the CAM is searched for the current `DataID` entry. If the entry is found, the new task arguments are stored at this `DataID`'s next empty entry. Otherwise, a new entry is popped from the free list for this purpose. The CAM is filled whenever a new entry is assigned to the task batching buffer, and is cleared whenever a task batch is full or served by the data response. The usage of free list and FIFO scheduler is similar to the task creation unit.

**Task Stream Table:** This table maintains the streams associated with TaskStream edges. These include the streams that transfer data: 1. from producer ports to the corresponding task creation/batching buffer, 2. from task creation/batching buffer to corresponding consumer ports, 3. Any streams for streaming data to remote cores.

**Communication Protocol:** For managing streaming communication with low network overhead, we use a coarse-grained credit-based flow control. The consumer core sends credits to the producer core when some number of entries become free in the consumer port (we find 8 keeps traffic low).

Also, during streaming communication, handshaking messages are exchanged to schedule the parent and child tasks. Here, we need to reorder messages so that the correct parents are matched to the correct children, which we accomplish using the "reordering circular queue". Space in this queue is allocated when a message is sent. At responses, the router messages are passed via this queue, then finally pushed to the vector port interface in order. Finally, we use a separate virtual channel for inter-accelerator messages to avoid deadlock. Round-robin scheduling is used for fairness.

## 5 METHODOLOGY

**Delta Power/Area:** We implemented Delta's CGRA by extending the Chisel-based DSAGEN [40, 66] framework. Components were synthesized using a 28nm UMC library. We use Cacti 7.0 [37] for estimating the overhead of the SRAM buffers and CAM within the task creation and batching units.

**Baseline Architectures:** For reference, we compare against a 24-core SKL CPU running optimized libraries: MKL [2] for Cholesky and SpMSpM, and MADLib [1] for DB-ML. For kNN, we use the popular FLANN [36] library. For GCN, we use the PyG library [3].

We developed a simulator for Delta and integrated it with gem5 [9, 47, 58], using a RISCV ISA [8] for the control core. For accelerator comparison points, we evaluated three designs:

- **Static Parallel**: Work is partitioned statically to each core by the programmer, and data is tiled into each core's scratchpad.

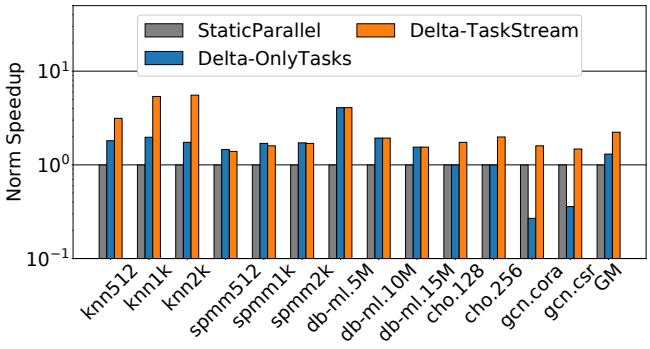


**Figure 7: Overall Performance Comparison**

- **Delta+OnlyTasks**: In this version, Delta tasks are scheduled dynamically in hardware using size-hint scheduling policy.
- **Delta-TaskStream**: This includes both load balance and locality/structure-recovery optimizations within TaskStream.

**Datasets:** We use synthetic datasets with varying sizes and natural skewness; GCN uses popular real-world graphs. Table 3 shows the dataset sizes and workload parameters.

**Parameters:** Table 4 shows the common hardware parameters of Delta and baselines.

**Table 4: Arch. Parameters**

| Characteristics | Value |
|---|---|
| Cores | 16 |
| FP Units | 1024 |
| Task buf. entries | 16 64-byte |
| Memory b/w | 256 GB/s |
| Shared Cache | 2 MB |
| Private Scratch | 256 kB |
| Network | 64-byte mesh |

## 6 EVALUATION

Broadly, the goal of our evaluation is to analyze whether our Delta proposal is able to recover locality structure while retaining the benefits of task parallelism. First we compare against a CPU and an equivalent static parallel design. Then we give insight into performance benefits with stream recovery (i.e. inter-task streaming/-batching) by examining network traffic and core utilization over time. Then, we explore sensitivity to the task scheduling strategy and pipelining depth. We conclude by discussing area overheads.

**Overall Performance:** First, we validate that Delta provides accelerator-like performance over existing multicore CPUs. Table 5 below shows the speedup over a 24-core baseline CPU. `Cholesky` has the least speedup, as Delta exhausts all the parallelism in this workload at the evaluated array size. The other workloads have more data-parallelism: `kNN` during linear search, `DB-ML` for performing kmeans search on many data points, `spmspm` and `GCN` for matrix multiply. Thus, they achieve higher speedup, as the data-parallel resources of the accelerator can be fully realized.

**Table 5: Speedup over 24-core SKL CPU**

| Wkld | Knn | Cholesky | DB-ML | Spmspm | GCN | GM |
|---|---|---|---|---|---|---|
| Speedup | 43.2 | 6.9 | 1729.2 | 65.3 | 105.4 | 81.3 |

**Benefit of Tasks:** Figure 7 shows that *Delta-OnlyTasks* achieves 1.3× speedup against the static reconfigurable accelerator that represents the state-of-the-art. *Delta-OnlyTasks* improve speedup for kNN and DB-ML by enabling better distribution of work irrespective of where the kmean's input (or leaf in kNN) is mapped statically,

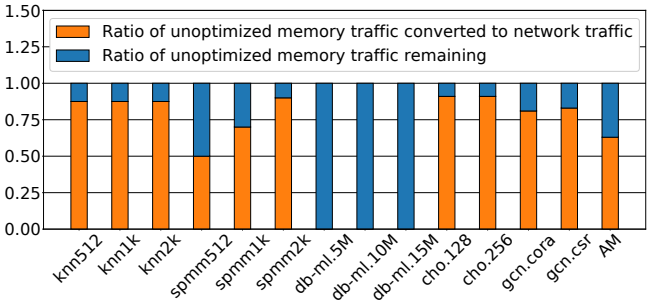


**Figure 8: Traffic-breakdown with Stream Recovery**

while it would matter in the static parallel implementation. For SpMSpM, the speedup comes from increased parallelism from overlapping execution of task types as data becomes available. Cholesky does not benefit due to low parallelism in both static-parallel and OnlyTasks without the stream-recovery optimization. GCN is an exception where there is a slowdown, as consecutive requests to access the weight matrix will create a hotspot in the network.

**Benefit of Structure Recovery:** Figure 7 also shows that with stream-recovery optimizations, the speedup increases to 2.2× over the static-parallel accelerator. SpMSpM does not benefit from stream-recovery, as it is not bottlenecked by communication; in the outer-product implementation, the batched column of the first matrix has high reuse – each element in the column is multiplied with each of the elements in the second matrix's corresponding row. However, kNN and GCN have less reuse, and batching tasks can reduce memory traffic by nearly an order of magnitude. Cholesky benefits from explicit communication among dependent tasks, which alleviates the overheads of shared memory synchronization. DB-ML does not have opportunity for stream recovery.

**Memory Traffic Reduction:** To explain the source of performance improvement using stream recovery, Figure 8 demonstrates what percentage of memory traffic is converted into network traffic in *Delta-TaskStream*. In most cases, more than 50% of the memory traffic is converted, with even fewer packets due to multicast.

**Fine-grained Core-wise Throughput Comparison:** TaskStream optimizations improve the core utilization[3] by alleviating communication bottlenecks, while providing load balance. We give insight into its capabilities by showing the per-workload utilization over time in Figures 9/10 and discuss below. Note that we only plot for a subset of cores for clarity; also, core 0 is used for lower-rate (or less compute intensive) tasks, therefore is generally under-utilized. The title shows the average utilization across *all* cores.

- **kNN:** Tasks enable overlap of the tree search tasks, improving utilization of Core 0, which is executing tree tasks. With stream batching optimization, the accesses to the leaf data are batched and multicast to co-scheduled tasks. This increases effective bandwidth and improves utilization (see Core 1, 2, 3).
- **DB-ML:** Also in Figure 9, the legend lists the task-to-core mapping in our implementation. Sort is performed in a separate phase for both implementations, hence we omit from this figure. Stream recovery opportunities do not exist in DB-ML, hence we compare static parallel and *Delta-OnlyTasks*. The *Delta* implementation dynamically distributes the "kmeans" tasks that are created by "join" and hence is able to balance load much better (see how static-parallel Core 1 and 2 are heavily under-utilized in some phases).
- **Cholesky:** For Cholesky, we show 12 outer-loop iterations (1 phase in TaskStream). In *Delta-OnlyTasks*, only one outer-loop iteration can be performed at a time due to inter-loop dependencies. This both reduces the available parallelism and introduces barrier overheads. *Delta* allows many more parallel tasks with the support for chained streaming. It further ensures balanced execution by distributing load based on `sizehint`.

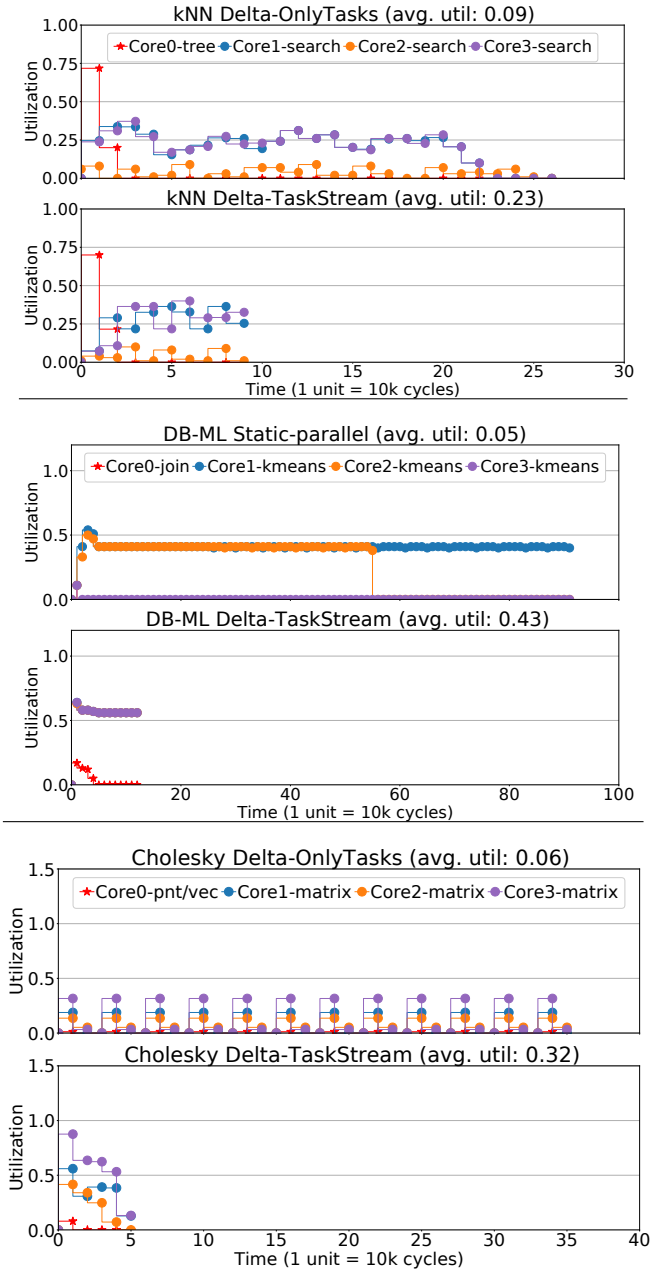


**Figure 9: Utilization Comparison with Stream Recovery.**

[3]Utilization is defined as the percentage of functional units (FUs) fired in each cycle.

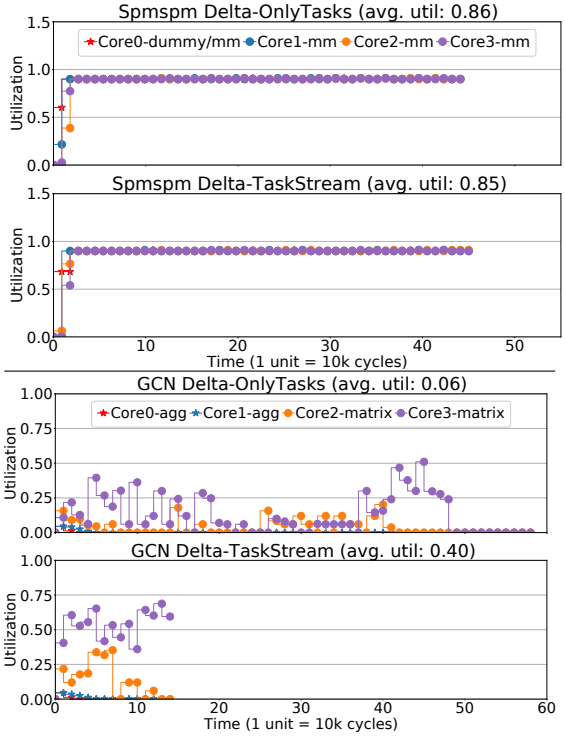


**Figure 10: Utilization Comparison with Stream Recovery**

- **Sparse matrix-multiply:** Now referring to Figure 10, due to the outer-product implementation of sparse matrix-multiply, there is high reuse available. Therefore, the utilization of task-parallel versions are nearly ideal, and there is not much potential left for task batching.
- **Graph Convolution Networks:** Stream-recovery significantly improves the throughput of the matrix-multiply cores, as batched reads of the shared weight matrix enables higher effective network bandwidth. Since 13/16 cores work on the matrix-multiply, this improves the average utilization by 6.6×.

**Comparing Task Scheduling Strategies:** Figure 11 compares four scheduling policies on *Delta*: round-robin, random, `spatialhint`, and `sizehint`. For *kNN*, `spatialhint` schedules tasks with the same leaf_id to the same core, so multicasting becomes irrelevant. For *Cholesky*, `sizehint` gains 1.63× performance over the baseline round-robin due to distributing square and triangular tiles better. For *DB-ML*, kmeans on all data items takes a similar amount of time, so load balance is less important, except that `spatialhint` restricts scheduling, thereby aggravating load imbalance. For *SpMSpM*, only `spatialhint` applies, as we are using an outer product, where partial sums are maintained in scratchpads, and we only allow tiles to be scheduled near their partial sums. This is because scheduling elsewhere would introduce excess remote fine-grained atomic update traffic, hurting performance. In *GCN*, the scheduling policy affects the vertex-access ordering. `sizehint` can balance load better by intelligently distributing vertices with varying degree. In conclusion, `sizehint` consistently outperforms the simpler random and round-robin policies. `Spatialhint` does not benefit much, as it restricts scheduling for load balancing without surpassing the locality benefits of stream recovery.

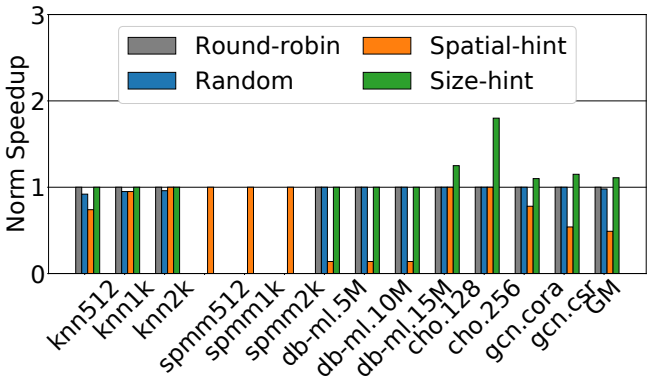


**Figure 11: Sensitivity to Load Balancing Strategies**

**Sensitivity to the Depth of Pipeline Parallelism:** TaskStream gives control to programmers on how to expand different facets of parallelism. We elucidate with an experiment on Cholesky, by varying the depth of streaming dependence chain (i.e. how many tasks are allowed to chain), and the results are below in Table 6. Higher depth improves available parallelism at the cost of extra streaming network traffic, and the optimal point for these array sizes occurs at a depth of 12. At higher depths, latency stalls to set up the pipeline streaming communication are more compared to the benefits of improved concurrency. One interesting finding is the pathological case at a depth of 10; performance drops because on our 16-core 4x4 mesh, around half of the cores are sending data to the next half, causing much of the communication to be on a few bisecting links. Detecting and mitigating this is future work.

**Table 6: Sensitivity to Dependence Chain Depth**

| depth | 1 | 2 | 6 | 8 | 10 | 12 | 14 |
|---|---|---|---|---|---|---|---|
| Cholesky-256 | 1 | 1.8 | 3.1 | 5.6 | 3.6 | 7.7 | 7.5 |
| Cholesky-128 | 1 | 2.0 | 2.4 | 2.6 | 3.1 | 3.6 | 3.4 |

**Area Overhead:** Table 7 shows the area breakdown of Delta, within and across cores. The only additional components required over the static architecture are the task management units, which consume 3.6% of the total on chip area.

**Table 7: Area and Power breakdown for Delta (28nm)**

| | Area ($mm^2$) | Power (mW) |
|---|---|---|
| Control Cores | 0.053 | 11.5 |
| Task Creation | 0.01 | 4.76 |
| Task Batching | 0.033 | 10 |
| scratchpad+ctrl | 0.08 | 11.2 |
| CGRA (4x5) | 0.21 | 80 |
| **1 Delta-Core** | 0.386 | 117.46 |
| 4x4 64 byte mesh (1) | 0.2 | 44.7 |
| Shared cache | 12.39 | 2280 |
| **Delta Total** | 18.77 | 4203.7 |

## 7 RELATED WORK

Table 8 compares Delta and prior works based on three critical factors: 1. *Sched-flex*: Flexibility to schedule work spatially across cores, which can help balance locality and load. 2. *Mem-sched*: Ability to exploit shared read-reuse among tasks. 3. *Inter-task-comm*: Write-read dependencies among tasks may be resolved via shared memory synchronization or explicit communication. No prior work simultaneously supports hardware task scheduling flexibility and memory scheduling, and none supports read-reuse with task-based parallelism. The remainder of this section discusses in further detail.

**Locality in Task-Runtimes:** An inspiration for our work is the study of locality-enhancing techniques for software-only threading and task-parallel systems [7, 13, 22, 32, 56, 75]. A prevailing mechanism is work-stealing [7, 22, 32, 49, 75], which lends itself to locality by keeping parent and child tasks together. This is effective in divide-and-conquer algorithms, but does not guarantee locality in general, and task stealing may incur non-negligible latency on the critical path (especially when ported to accelerated systems).

Of particular note are techniques which recover program structure. One approach is to annotate tasks with the dataset they may access, and use this to schedule tasks near-data. This has been explored in task-parallel languages [12], speculative parallel models [27, 74], and partitioned global address space (PGAS) systems [25]. Another approach is Splicing [31], which is a compiler optimization for recursive task-parallel programs that interleaves tasks with locality to optimize for locality.

**Accelerating Task Parallel Codes:** Task management is a significant overhead that has been mitigated by hardware based [29, 76], or hardware assisted schedulers [51]. Anton2 [55] uses a hardware-assisted task runtime for geometry processing.

Prior work has also explored accelerating task parallel workloads in reconfigurable hardware. TAPAS [33] is a high level synthesis (HLS) toolchain that leverages the Tapir IR [53] to create application-specific hardware for task-parallel workloads. $\mu$IR [54] is a hardware design IR, also useful in the context of HLS, that supports task-parallel constructs. ParallelXL [14] is a framework that enables building custom hardware accelerators using task-parallel execution and work stealing. Chronos [6] is a framework to build task-parallel accelerators for applications with speculative parallelism. Three recent works address flexible parallelism for reconfigurable accelerators: Aurochs [61] proposes a threading model for reconfigurable dataflow architectures. PolyGraph [18] similarly adds an integrated task/dataflow model for a reconfigurable accelerator. Fifer [38] temporally reorders fine-grained tasks for load balancing. TaskStream's structure-recovery optimizations can be applied to any of the above systems.

**Exploiting & Recovering Streaming Structure:** In the context of general-purpose processing, stream floating [65] has a "confluence" optimization that dynamically combines prefetching streams from different cores (confluence). Near-stream computing [64] can pipeline data between tasks offloaded to the last-level cache. Several prior reconfigurable accelerators [19, 67, 79] have primitives for multicast, but they are suitable for only for regular programs.

Prior domain-specific accelerators *can* exploit batching and pipelining in irregular programs, but the expected structure is baked into the hardware. One example is multicasting in sparse matrix/DNN accelerators [15, 24, 44, 46, 73, 77, 78]. Other specialized architectures perform load balance optimizations while exploiting various forms of reuse in hardware, like SparTen [20], BARISTA [21] and GraphDynS [72]. MTP [11] hides memory latency for database selections by using concurrent fine-grained accelerator threads. Centaur [42] exploits streaming locality in databases by combining multiple pipelined SQL operators.

In summary, our work is the first to demonstrate techniques for exploiting structured communication in general task-parallel accelerators.

**Table 8: Related work comparison**

| | | Sched-flex | Mem-sched | Inter-task-comm |
|---|---|---|---|---|
| S/w | Splicing* [31] | High | Read-reuse | Memory |
| | Gramps* [50] | High | | Memory |
| Accel-assisted CPU | Stream-floating* [65] | High | Read-reuse | Memory |
| | Carbon [29] | High | | Memory |
| | ADM [51] | High | | Memory |
| | Minnow [76] | High | | Memory |
| | Spatial-Hints [27] | Low | Near-data | Memory |
| Reconfigurable Accelerators | ParallelXL [14] | High | | Memory |
| | Centaur [42] | High | | Explicit |
| | Plasticine* [45] | Restricted | Read-reuse | Explicit |
| | PolyGraph [18] | Restricted | Near-data | Memory |
| | Aurochs [61] | High | | Memory |
| | Fifer [38] | Restricted | Near-data | Memory |
| | **Delta (ours)** | **Medium** | **Read-reuse** | **Explicit** |

* Uses traditional threads for parallelism; no hardware support for tasks.

## 8 CONCLUSION

For reconfigurable accelerators, task-parallelism is both a blessing and a curse: while it bestows dynamic parallelism and load balance capabilities, it also breaks and hides program structure that such architectures are designed to exploit. We discovered that this structure can be recovered dynamically, provided that the hardware's execution model is rich enough to express inter-task communication patterns. The potential gains are significant, and the costs are little: mean 2.2× speedup for this class of workloads, with only 3.6% area overhead. While the programmer/compiler does have to identify the potential for structure, they are also relieved from the difficult job of task scheduling, which is especially challenging for accelerators with heterogeneous memories and reconfiguration overheads.

Finally, this work contributes to a broader effort to enable reconfigurable accelerators to be efficient on increasingly irregular and challenging workloads; this is important if we want accelerators to not only crunch though data at high throughput, but also to do so using algorithmically-efficient irregular algorithms.

## ACKNOWLEDGMENTS

This work was supported by NSF awards CCF-1751400 and CCF-1937599, as well as gift funding from VMware.

# REFERENCES


[1] [n.d.]. Apache MADlib: Big Data Machine Learning in SQL. https://madlib.apache.org/.

[2] [n.d.]. Intel Math Kernel library. http://software.intel.com/en-us/intel-mkl.

[3] [n.d.]. PyTorch Geometric. https://www.pyg.org/.

[4] 2021. Accelerated Computing with a Reconfigurable Dataflow Architecture. (2021). https://sambanova.ai/wp-content/uploads/2021/06/SambaNova_RDA_Whitepaper_English.pdf.

[5] 2021. Cerebras Systems: Achieving Industry Best AI Performance Through A Systems Approach. (2021). https://cerebras.net/wp-content/uploads/2021/04/Cerebras-CS-2-Whitepaper.pdf.

[6] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1247–1262. https://doi.org/10.1145/3373376.3378454

[7] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The Data Locality of Work Stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '00)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/341800.341801

[8] Krste Asanović and David A Patterson. 2014. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014). https://riscv.org/publications/

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011). https://doi.org/10.1145/2024716.2024718

[10] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69. https://doi.org/10.1145/209936.209958

[11] Prerna Budhkar, Ildar Absalyamov, Vasileios Zois, Skyler Windh, Walid A Najjar, and Vassilis J Tsotras. 2019. Accelerating in-memory database selections using latency masking hardware threads. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 2 (2019), 1–28. https://doi.org/10.1145/3310229

[12] Rohit Chandra, Anoop Gupta, and John L. Hennessy. 1993. Data Locality and Load Balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)*. Association for Computing Machinery, New York, NY, USA, 249–259. https://doi.org/10.1145/155332.155358

[13] Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, et al. 2007. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. 105–115. https://doi.org/10.1145/1248377.1248396

[14] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. 2018. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. IEEE Press, Piscataway, NJ, USA, 55–67. https://doi.org/10.1109/MICRO.2018.00014

[15] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/jetcas.2019.2910232

[16] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 184–189. https://doi.org/10.1109/ASAP.2017.7995277

[17] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. 2014. A fully pipelined and dynamically composable architecture of CGRA. In *FCCM*. IEEE, 9–16. https://doi.org/10.1109/fccm.2014.12

[18] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[19] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms *(MICRO '52)*. ACM, New York, NY, USA, 924–939. https://doi.org/10.1145/3352460.3358276

[20] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[21] Ashish Gondimalla, Sree Charan Gundabolu, T. N. Vijaykumar, and Mithuna Thottethodi. 2021. Barrier-Free Large-Scale Sparse Tensor Accelerator (BARISTA) For Convolutional Neural Networks. *CoRR* abs/2104.08734 (2021). arXiv:2104.08734 https://arxiv.org/abs/2104.08734

[22] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470425

[23] Gagan Gupta and Gurindar S Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 59–70. https://doi.org/10.1145/2155620.2155628

[24] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *SIGARCH Comput. Archit. News* 44, 3 (June 2016), 243–254. https://doi.org/10.1145/3007787.3001163

[25] Brandon Holt, Preston Briggs, Luis Ceze, and Mark Oskin. 2014. Alembic: Automatic Locality Extraction via Migration. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 879–894. https://doi.org/10.1145/2660193.2660194

[26] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. 2013. Elastic CGRAs. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. Association for Computing Machinery, New York, NY, USA, 171–180. https://doi.org/10.1145/2435264.2435296

[27] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-Centric Execution of Speculative Parallel Programs. In *Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49)*. https://doi.org/10.1109/micro.2016.7783708

[28] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 68–80. https://doi.org/10.1145/3178487.3178493

[29] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 162–173. https://doi.org/10.1145/1273440.1250683

[30] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *ASPLOS (ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 461–475. https://doi.org/10.1145/3173162.3173176

[31] Jonathan Lifflander and Sriram Krishnamoorthy. 2017. Cache Locality Optimization for Recursive Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3062341.3062385

[32] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2014. Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 857–868. https://doi.org/10.1109/SC.2014.75

[33] S. Margerm, A. Sharifian, A. Guha, A. Shriraman, and G. Pokam. 2018. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 245–257. https://doi.org/10.1109/MICRO.2018.00028

[34] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*. Springer, 61–70. https://doi.org/10.1007/978-3-540-45234-8_7

[35] Ethan Mirsky, Andre DeHon, et al. 1996. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources.. In *FCCM*, Vol. 96. 17–19. https://doi.org/10.1109/fpga.1996.564808

[36] Marius Muja and David G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)* 2, 331-340 (2009), 2. https://doi.org/10.5220/0001787803310340

[37] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories* (2009), 22–31. https://www.hpl.hp.com/techreports/2009/HPL-2009-85.html

[38] Quan M Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1064–1077. https://doi.org/10.1145/3466752.3480048

[39] Chris Nicol. 2017. A Coarse Grain Reconfigurable Array (CGRA) for Statically Scheduled Data Flow Computing. *WaveComputing WhitePaper* (2017). http://www.silicon-russia.com/public_materials/2017_10_08_msu_rountable/background/CGRA+Whitepaper.pdf

[40] Tony Nowatzki, Newsha Ardalani, Karthikeyan Sankaralingam, and Jian Weng. 2018. Hybrid Optimization/Heuristic Instruction Scheduling for Programmable Accelerator Codesign. In *27th PACT*. https://doi.org/10.1145/3243176.3243212

[41] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration *(ISCA '17)*. ACM, New York, NY, USA, 416–429. https://doi.org/10.1145/3079856.3080255

[42] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A

framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 211–218. https://doi.org/10.1109/fccm.2017.37

[43] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. 2013. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 142–153. https://doi.org/10.1145/2485922.2485935

[44] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/3079856.3080254

[45] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns *(ISCA '17)*. ACM, New York, NY, USA, 389–402. https://doi.org/10.1145/3079856.3080256

[46] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[47] Alec Roelke and Mircea R. Stan. 2017. RISC5: Implementing the RISC-V ISA in gem5. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*. https://carrv.github.io/2017/papers/roelke-risc5-carrv2017.pdf

[48] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. arXiv:cs.AR/2104.12760

[49] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 22–32. https://doi.org/10.1109/PACT.2011.9

[50] Daniel Sanchez, David Lo, Richard M Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic fine-grain scheduling of pipeline parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 22–32. https://doi.org/10.1109/pact.2011.9

[51] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible Architectural Support for Fine-grain Scheduling. *SIGPLAN Not.* 45, 3 (March 2010), 311–322. https://doi.org/10.1145/1735971.1736055

[52] Alina Sbîrlea, Yi Zou, Zoran Budimlic, Jason Cong, and Vivek Sarkar. 2012. Mapping a data-flow programming model onto heterogeneous platforms. *ACM SIGPLAN Notices* 47, 5 (2012), 61–70. https://doi.org/10.1145/2345141.2248428

[53] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3018743.3018758

[54] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. 2019. $\mu$IR -An Intermediate Representation for Transforming and Optimizing the Microarchitecture of Application Accelerators. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 940–953. https://doi.org/10.1145/3352460.3358292

[55] David E Shaw, JP Grossman, Joseph A Bank, Brannon Batson, J Adam Butts, Jack C Chao, Martin M Deneroff, Ron O Dror, Amos Even, Christopher H Fenton, et al. 2014. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 41–53. https://doi.org/10.1109/SC.2014.9

[56] Harsha Vardhan Simhadri, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Aapo Kyrola. 2014. Experimental Analysis of Space-Bounded Schedulers. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*. Association for Computing Machinery, New York, NY, USA, 30–41. https://doi.org/10.1145/2612669.2612678

[57] Hartej Singh, Ming-Hau Lee, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. 2000. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Trans. Comput.* 49, 5 (May 2000), 465–481. https://doi.org/10.1109/12.859540

[58] Tuan Ta, Lin Cheng, and Christopher Batten. 2018. Simulating Multi-Core RISC-V Systems in gem5. (2018). https://carrv.github.io/2018/papers/CARRV_2018_paper_3.pdf

[59] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. 2017. General purpose task-dependence management hardware for task-based dataflow programming models. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 244–253. https://doi.org/10.1109/ipdps.2017.48

[60] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. 2021. Ultra-Elastic CGRAs for Irregular Loop Specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 412–425. https://doi.org/10.1109/HPCA51647.2021.00042

[61] Matthew Vilim, Alexander Rucker, and Kunle Olukotun. 2021. Aurochs: An Architecture for Dataflow Threads. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 402–415. https://doi.org/10.1109/ISCA52012.2021.00039

[62] Matthew Vilim, Alexander Rucker, Yaqi Zhang, Sophia Liu, and Kunle Olukotun. 2020. Gorgon: Accelerating Machine Learning from Relational Data. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 309–321. https://doi.org/10.1109/ISCA45697.2020.00035

[63] Dani Voitsechov and Yoav Etsion. 2014. Single-graph Multiple Flows: Energy Efficient Design Alternative for GPGPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 205–216. https://doi.org/10.1109/ISCA.2014.6853234

[64] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. 2022. Near-Stream Computing: General and Transparent Near-Cache Acceleration. In *HPCA*. https://seanzw.github.io/pub/hpca2022-near-stream-computing.pdf

[65] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. 2021. Stream Floating: Enabling Proactive and Decentralized Cache Optimizations. In *HPCA*. https://doi.org/10.1109/hpca51647.2021.00060

[66] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *ISCA*. IEEE, 268–281. https://doi.org/10.1109/ISCA45697.2020.00032

[67] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 703–716. https://doi.org/10.1109/HPCA47549.2020.00063

[68] Bob Wheeler. 2020. Growing AI Diversity and Complexity Demands Flexible Data-Center Accelerators. (2020). https://www.linleygroup.com/uploads/simple-machines-wp.pdf.

[69] WikiChip. 2019. Configurable Spatial Accelerator. https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator.

[70] Justin M Wozniak, Timothy G Armstrong, Michael Wilde, Daniel S Katz, Ewing Lusk, and Ian T Foster. 2013. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 95–102. https://doi.org/10.1109/ccgrid.2013.99

[71] Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. 2019. Tigris: Architecture and algorithms for 3d perception in point clouds. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 629–642. https://doi.org/10.1145/3352460.3358259

[72] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating Irregularity in Graph Analytics Acceleration: A Hardware/Software Co-Design Approach *(MICRO '52)*. ACM, New York, NY, USA, 615–628. https://doi.org/10.1145/3352460.3358318

[73] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. 2020. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 711–724. https://doi.org/10.1109/micro50266.2020.00064

[74] Victor A Ying, Mark C Jeffrey, and Daniel Sanchez. 2020. T4: Compiling sequential code for effective speculative parallelization in hardware. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 159–172. https://doi.org/10.1109/isca45697.2020.00024

[75] Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. 2013. Locality-Aware Task Management for Unstructured Parallelism: A Quantitative Limit Study. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '13)*. Association for Computing Machinery, New York, NY, USA, 315–325. https://doi.org/10.1145/2486159.2486175

[76] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching *(ASPLOS '18)*. ACM, New York, NY, USA, 593–607. https://doi.org/10.1145/3173162.3173197

[77] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3445814.3446702

[78] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W. Keckler, and Zhengya Zhang. 2021. SNAP: An Efficient Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference. *IEEE J. Solid State Circuits* 56, 2 (2021), 636–647. https://doi.org/10.1109/JSSC.2020.3043870

[79] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable interconnects for reconfigurable spatial architectures. In *46th ISCA*. https://doi.org/10.1145/3307650.3322249