The 5th Annual Parallel Applications Workshop,
Alternatives To MPI+X (PAWATM) November 14 2022

Task Fusion in Distributed Runtimes

Shiv Sundram
Department of Computer Science
Stanford University
Stanford, CA
shiv1@stanford.edu

Wonchan Lee
NVIDIA
Santa Clara, USA
wonchanl@nvidia.com

Alex Aiken
Department of Computer Science
Stanford University
Stanford, CA
aaiken@cs.stanford.edu

***Abstract*—We present *distributed task fusion*, a runtime optimization for task-based runtimes operating on parallel and heterogeneous systems. Distributed task fusion dynamically performs an efficient buffering, analysis, and fusion of asynchronously-evaluated distributed operations, reducing the overheads inherent to scheduling distributed tasks in implicitly parallel frameworks and runtimes. We identify the constraints under which distributed task fusion is permissible and describe an implementation in Legate, a domain-agnostic library for constructing portable and scalable task-based libraries. We present performance results using cuNumeric, a Legate library that enables scalable execution of NumPy pipelines on parallel and heterogeneous systems. We realize speedups up to 1.5x with task fusion enabled on up to 32 P100 GPUs, thus demonstrating efficient execution of pipelines involving many successive fine-grained tasks. Finally, we discuss potential future work, including complementary optimizations that could result in additional performance improvements.**

***Index Terms*—Legion, Legate, cuNumeric, MPI+X, Task-based runtimes, NumPy**

I. Introduction

Task-based programming models are an increasingly prominent and useful way to program parallel and heterogeneous machines for domains including scientific computing [8], [9], [16], machine learning [3], [23], and data-analytics [31]. These models allow users to express self-contained tasks and their dependencies, which can be executed on a distributed, heterogeneous machine by a tasking runtime. In MPI+X programming models, users must manually partition data across processors, map computations to processors, and manage inter-processor communication. Task-based programming models help automate some or all of these steps, making task-based programming more productive.

Task-based systems have associated overheads with launching tasks—tasks must be analyzed, resources allocated, and scheduled. If the execution time of tasks is comparable to or smaller than these task overheads, then performance will be poor as the overheads are not sufficiently amortized by application work [27]. The classic solution to having tasks that are too short is to *fuse* tasks: to combine two or more otherwise unrelated tasks into a single task to improve overall task granularity [1], [12].

We consider a new twist on task fusion, *distributed task fusion*, that arises in task-based models that support higher-level launches of *task groups* [28], collections of tasks that are launched in a single operation across a machine. Fusing task groups is more efficient than fusing individual tasks, but because task groups are fundamentally distributed collections of tasks there are new constraints on when fusion is legal.

We illustrate and evaluate distributed task fusion within cuNumeric [5], [6], a drop-in replacement for NumPy [14] that enables execution on parallel and heterogeneous architectures. The cuNumeric library automatically partitions NumPy operations into task groups, which are then submitted to a task-based runtime. Because cuNumeric performs this task partitioning online and fully automatically, it can create very small tasks that benefit significantly from distributed task fusion.

We implement distributed task fusion within Legate [6], cuNumeric's backend task-based parallel programming model. Legate itself is built on Legion, which is the underlying tasking runtime [8]. We demonstrate distributed task fusion's performance improvement on several NumPy/cuNumeric benchmarks, achieving 1.1x-1.5x speedups entirely through the fully automatic amortization or complete elimination of certain task overheads.

II. Execution Model

We assume an execution model where a machine is a collection of processors $\{p_0, \ldots, p_{n-1}\}$, each of which has a *task queue*.

In our initial examples we assume a simple setting where a program is a single top-level task that executes a sequence of *index launches*. An index launch is a group of homogeneous subtasks and an associated *projection function* f. Each subtask t_i in an index launch has a unique *index* i and is entered in the task queue for processor p_i. Separately, we assume that each array A used by a program is *partitioned* into pieces $\{A_0, \ldots, A_{n-1}\}$ with A_i placed in memory belonging to processor p_i. When the task t_i runs on p_i, the projection function is used to determine which subsets of t_i's array arguments t_i will access.

For example, if $f(A,i) = A_i$ then task t_i uses the partition of A local to p_i, but if $f(A,i) \cap (A - A_i) \neq \emptyset$ then communication is required to copy the data in $f(A,i) - A_i$ to p_i before task t_i can execute.

In the general case, an index launch takes an additional parameter, a *launch space* I, where I is a set of possibly multidimensional indices, and the index launch executes one instance of the task t for each index in I. In our simplified model, the launch space is implicitly $\{0, \ldots, n-1\}$. We will present examples with interesting launch spaces in Section III-C. In real programs, the assignment of tasks to processors also need not be 1-1, and arrays do not need to be partitioned evenly across the entire set of processors. However, none of these features are relevant to distributed task fusion, so we will focus on the simple execution model for clarity while developing our fusion algorithm.

A cuNumeric program consists of a sequence of calls to NumPy library operations, each of which is a bulk operator over some NumPy arrays. The cuNumeric library partitions the arrays across the processors and converts each of the library calls into an index launch. It also automatically generates a projection function for each index launch that ensures each subtask accesses the correct portion of the partitioned data. We use the following simple NumPy program (adapted from a line in Listing 1, a Black-Scholes implementation) for illustration:

```
T3 = np.expr(-R*T)
```

In this simple example, there are two input arrays R and T which we will assume have size 16. We will use this as a running example to illustrate the effects of fusion. There are three array operations: a negation, a multiplication, and an exponentiation.

Figure 1 shows the execution of this program on a single processor. Input arrays `R` and `T` are placed on the processor p_0, the three index launches have only a single subtask each (so the launch space is $\{0\}$), and the projection function simply returns all the data $f(X,0) = X$. The figure shows the state of execution when all three tasks have been enqueued on p_0 and before any of them execute. The only other task (shown at the left) is the top-level task that launches the three index tasks [7]. The output arrays `T1`, `T2` and `T3` are also stored entirely on p_0.

Now consider the same program executed in parallel on four processors, shown in Figure 2. The arrays `R` and `T` are evenly partitioned across the processors with 4 elements each, and each index launch now launches four subtasks (the launch space is $\{0,1,2,3\}$). Since each indexed task will access the local portion of the arrays, the projection function is $f(X,i) = X[i*4 : (i+1)*4]$. Each queue will receive the the same sequence of three subtasks (as in Figure 1) operating on the local partition of each

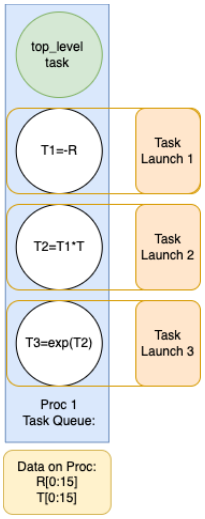

Figure 1: Execution of program `np.expr(-R*T)` on one processor. Each task results from an index launch (orange) from a top level control task (green). Temporaries `T1`, `T2`, and `T3` will also reside completely on the sole processor.

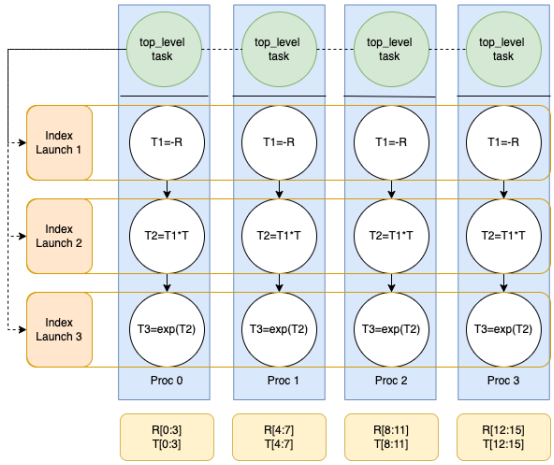

Figure 2: Execution of the program `np.expr(-R*T)` on four processors. The arrays, including the temporaries, are partitioned equally across the processors.

array. As before, cuNumeric will choose to partition the output arrays `T1`, `T2`, and `T3` across processors identically to `R` and `T` (and so the same projection function f is used for these arrays as well).

As mentioned above, communication occurs when a task requires data other than the local partition of an array on a processor. Consider the following 1D, three point stencil:

```
1 input = np.array(N=18)
2 central = input[1 : -1]
3 west = input[0 : -2]
4 east = input[2 : N]
5 for i in range(num_iters):
6     output = east + west
7     central[:] = output
```

In Python a *slice* `A[x:y]` of an array names the elements `A[x]` through `A[y-1]`. Negative indices refer to positions before the end (the maximum index) of the array, so `input[1:-1]` names everything from the second element `input[1]` (Python arrays are 0-based) to the next-to-last element `input[16]`. Note that slices are *views* (NumPy terminology), meaning they are aliases of the underlying physical array. Thus the arrays `central`, `west` and `east` are aliases for different subpieces of the array `input` in this example.

Assume `input` is partitioned such that processor 0 contains `input[0:4]` as shown in Figure 3. Now consider the data requirements for the subtask of the addition `output = east + west` on processor 0. This task will require `west[0:4]` and `east[0:4]` (i.e., the projection function f(west,0) = west[0:4] and f(east,0) = east[0:4] for the addition task) which are aliases for the subarrays `input[0:4]` and `input[2:6]`, respectively. This task will thus require `input[0:6]`, a range that is missing `input[4:6]` on processor 0. During the index launch the runtime will detect that this data is needed on processor 0 and prompt input[4:6] to be communicated from processor 1 before the task `output = east + west` proceeds. Analogous communication will happen at other processors, shown in Figure 3.

If this example's execution is distributed, any purely local task fusion (i.e., that does not consider the distributed nature of the execution) may perform incorrect fusions that result in race conditions where none existed before. To illustrate, each iteration of the stencil's inner loop results in two tasks: an addition task (line 6) that reads from the underlying buffer, and a subsequent copy task (line 7) that writes to it. Recall that communication can occur as a result of the addition task (shown in Figure 3). If we run two iterations of this loop and fuse all four tasks, we could, in the third task, potentially be reading from the underlying buffer before the updates from other processors have been fully communicated and written to the buffer. Additional constraints are needed prevent this unsafe fusion in the distributed case, which we present in Section III-B.

As in all task-based models, tasks do not communicate with each other except through inputs and outputs—no communication with other tasks takes place internally during the execution of a task.

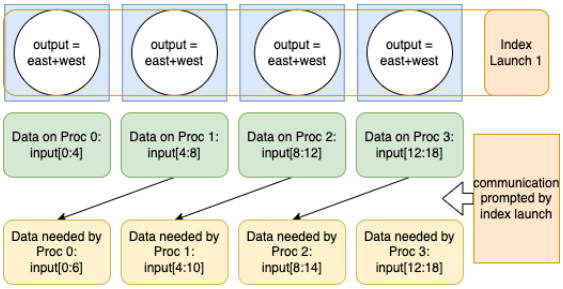

Figure 3: Executing a 1D two point stencil on four processors. Inter-processor communication (solid arrows) is needed because each task requires more data than is stored in a processor's local partition of the `input` array. The control tasks and `central[:] = output` (which is a trivial unary/copy task) are not shown.

III. Task Fusion

Recall our first example:

```
1 T3 = np.expr(-R*T)
```

The cuNumeric library, through Legate, issues a separate index task launch for each of the three respective function calls (the negation, multiplication, and exponentiation). While this design enables a great deal of parallelism, it has negative performance implications if the subtasks are small. Each subtask is associated with additional meta-tasks that manage scheduling, synchronization, and data movement. An abundance of index task launches of small tasks will spawn many such meta-tasks that can bottleneck the application's performance when the tasks' execution times do not sufficiently amortize these overheads.

The goal of distributed task fusion, therefore, is to mitigate these overheads by combining n index tasks into a single aggregate index task, thus reducing the number of required meta-tasks (and their associated overhead) by a factor of n. In this case, the negation, addition and exponentiation subtasks all have input and output tensors of the same shape on every processor. Furthermore, the dependence graph of the subtasks on each processor is such that each subtask is dependent solely on outputs of other subtasks on the same processor. These dependencies are represented by solid arrows between subtasks (white circles) in Figure 2. As these arrows never cross processor boundaries, all inter-task dependencies are local to a processor. It thus makes sense for these operations to be fused and share meta-tasks to amortize the associated overhead. The effect of fusion is illustrated in Figure 4, in which the three involved operations (negation, additions, and exponentiation) now take place in the same task.

An efficient implementation of distributed task fusion requires some changes to the execution model,

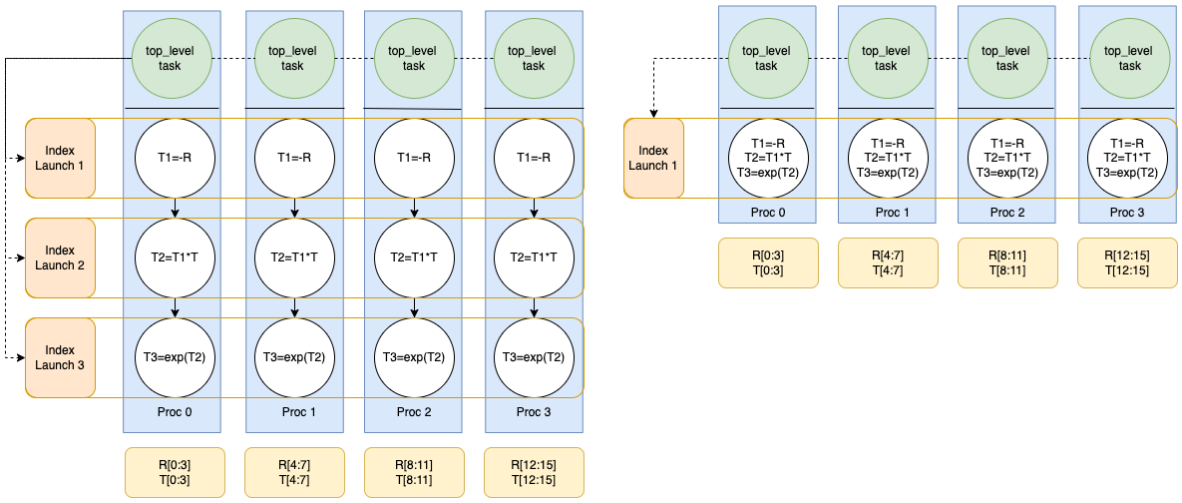

Figure 4: Task graph before (left) and after (right) fusion of the sub-program `T3 = np.exp(-R * T)`, from line 26 of the Black-Scholes code in Listing 1. Each operation (eg `T1 = -R`) is executed in parallel by a sequence of subtasks (white) distributed across separate processors (blue). Each subtask executes the given operation on distinct partitions (yellow) of the arrays. After task fusion, only one index launch is necessary.

```python
import cuNumeric as np

def cnd(d):
    A1 = 0.31938153
    A2 = -0.356563782
    A3 = 1.781477937
    A4 = -1.821255978
    A5 = 1.330274429
    RSQRT2PI = 0.39894228040143267793994605993438
    K = 1.0 / (1.0 + 0.2316419 * np.absolute(d))
    cnd = (
        RSQRT2PI
        * np.exp(-0.5 * d * d)
        * (K * (A1 + K * (A2 + K * (A3 + K * (A4 + K * A5)))))
    )
    return np.where(d > 0, 1.0 - cnd, cnd)

def black_scholes(S, X, T, R, V):
    sqrt_t = np.sqrt(T)
    d1 = np.log(S / X) + (R + 0.5 * V * V) * T / (V * sqrt_t)
    d2 = d1 - V * sqrt_t
    cnd_d1 = cnd(d1)
    cnd_d2 = cnd(d2)
    T3 = np.exp(-R * T)
    call_result = S * cnd_d1 - X * T3 * cnd_d2
    put_result = X * T3 * (1.0 - cnd_d2) - S * (1.0 - cnd_d1)
    return call_result, put_result
```

Listing 1: Black-Scholes code in Python + cuNumeric. As seen on line 1, replacing NumPy's backend with cuNumeric's is a matter of changing this import line. Every operation is fusable—the entire program could be run as a single index task launch.

including a software mechanism for buffering operations for possible fusion before submitting them to the task queue. Fusing multiple index launches into a single index launch is also semantically different from fusing single tasks. In fact, fusing index launches is not always correct and we will motivate and present the conditions under which fusion is possible in Section III-B. While task fusion is not new [1], [12], our contribution addresses this more general problem of fusing collections of distributed tasks. Our particular fusion implementation resides in the Legate runtime's back-end as a new part of Legate's task generation pipeline. Our task fusion algorithm is completely automatic and transparent to Legate programmers. Fusing tasks requires no changes to user-facing cuNumeric code.

A. Fusion Algorithm

The first step in distributed task fusion is to analyze a window of index launches which have not yet been submitted to the runtime. In the context of our initial example `T3 = np.exp(-R * T)`, the three tasks would not be executed immediately, but instead placed in a buffer (a queue which holds the window of tasks) for deferred execution. This window is flushed, and its tasks executed, once it reaches a certain size or when the runtime encounters an I/O operation that requires access to a data value computed by any of these tasks. Similar buffering is used in many dynamic runtimes where there is a need to collect and analyze a sequence of operations before execution [10], [13], [19], [22], [23].

The length of the window of deferred tasks is an upper bound on the maximum number of tasks that can be fused. While the maximum length of this window is adjustable, in our benchmarks it is always beneficial to fuse as many tasks as possible. We set the maximum window size to 50, which is around the highest degree of fusion that could be achieved in any of our benchmarks.

The fusion process is illustrated with Python pseudocode in Listing 2. Once we have a window of tasks we apply four fusion constraints (described in Section III-B), which are rules that dictate which tasks can be fused. Barriers are placed between tasks wherever fusing across that barrier would violate program correctness. Once barriers have been placed for all four constraints, we are left with an ordered list of subsequences of barrier-free tasks—each of these subsequences is a fusable sequence of tasks. The *fuser* then builds a new task F for each fusable subsequence, removes the original unfused buffered tasks from the window, and replaces them with F. If a subsequence contains only one task it is not replaced, as creating a fused task for a single subtask is not an improvement.

```
window = [ordered sequence of Legate tasks]
fusable_subsequences = [window]
#apply fusion constraints
for constraint in constraints:
  fusable_subsequences = constraint.
    place_fusion_barriers(
    fusable_subsequences)

fused_tasks = []
for subsequence in fusable_subsequences:
  fused_tasks.append(createFusedTask(
    subsequence))

#sends fused tasks to legion for execution
#for distributed execution
for fused_task in fused_tasks:
  launch(fused_task)
```

Listing 2: Pseudocode for generating and launching a sequence of fused tasks from an initial window of unfused tasks. This code applies each of the four constraints to the window of unfused, buffered tasks. Each constraint will place barriers within the window that denote where we cannot fuse across tasks. Contiguous subsequences of tasks without a barrier are thus safely fusible subsequences of task. Each such fusible subsequence will then be converted to a fused task.

During the fused task's execution, the fused task will call the respective functions for launching each subtask's operation by simply invoking a function pointer—all of the task overhead at this point has been handled by the interface to the fused task and the internal invocation of the subtasks is simply a function call. A fused task's outputs are the union of all its subtasks' outputs, and its inputs are the union of all its subtasks' inputs, minus those which are also outputs of other subtasks. This increase in the number of task inputs and outputs reflects the increased granularity of the fused task—some task level parallelism amongst the tasks is traded for reductions in index task launch overheads.

The order of tasks in the fusion window always respects any task dependences, and is thus a valid scheduling order. Our implementation only fuses contiguous tasks within the window, which simplifies the implementation of the fuser. Higher levels of fusion could possibly be achieved by fusing non-contiguous tasks; we leave this as potential future work.

Our task fusion implementation has the ability to operate on tasks regardless of whether they execute on CPUs or GPUs, but currently we assume that all tasks must run either completely on GPUs or completely on CPUs. An extension to support hybrid parallel workloads, with some tasks running on CPUs and others on GPUs, is straightforward: we would add a constraint to enforce that fused tasks must run on the same processor type. This extension was not

needed in any of our benchmark applications, which map all NumPy operations uniformly to either CPUs or GPUs.

B. Constraints on Fusion

It is not the case that the subtasks of multiple index launches can always be fused. In this section we give four constraints on when tasks in the fusion window can be combined. These constraints represent sufficient, but not strictly necessary, conditions under which task fusion is permissible. While in theory some of these conditions could be relaxed to enable further optimization, in each case it would add significant complexity to the design point we have chosen. The constraints are:

Communication Absence: Recall that tasks cannot communicate with other tasks during their execution. Thus, fusion may only be applied to a sequence of operations that does not involve communication within the tasks' collective execution period. Fusing index tasks where the fused subtasks only communicate internally (such as our motivating example `T3 = np.expr(-R*T)` is legal. Reduction operations, however, cannot be fused, as the implementation of parallel reductions requires partial reductions to be communicated between processors during the reduction's execution. We enforce this constraint by ensuring that only certain operations are considered for fusion, implemented by simply inspecting each task's opcode.

Launch Space Equivalence: Each index launch is associated with a launch space. Each launch space is a tuple specifying dimensions of a grid of processors used to execute the respective task. Given a window of contiguous tasks, all encapsulated tasks must have equivalent launch spaces for the fused task to have a single launch space for a single overall index launch. In general, the launch space can be multidimensional (though we have only considered one dimensional launch spaces for simplicity) and will often correspond to a domain decomposition of the input data (e.g., a 2D input matrix naturally maps to a 2D launch space). Consider, for example, one index launch that is associated with a 2x2 grid of processors, and a subsequent index launch that is associated with a 3x3 grid of processors. If the 2x2 and 3x3 launch spaces use a common array, the index launches are not fusable; communication would be required to move data from a 2x2 domain decomposition to a 3x3 decomposition, violating the requirement that fused tasks cannot have inter-task communication except at task boundaries.

Projection Equivalence: Each view of an array is associated with a projection function that maps each subtask's index to the subview of the array that subtask needs. Tasks can only be fused if, for any array they have in common, the result of the projection function for that array is the same. In other words, fused subtasks must all have the same view of any array they have in common.

Producer-Consumer Restriction: This constraint prevents fusion of tasks in which there exist producer-consumer relationships between different views of the same array. If an index launch writes to view `A` of an array, and a subsequent index launch reads from a different view `B` of the same array, then the index launches are not fusable. This constraint is motivated by the fact that in producer-consumer relationships involving different views, we cannot guarantee within a single fused task that writes through one view of an array (originating in one subtask) are guaranteed to be seen by reads from a different view of the same array (originating in a different subtask). The issue is the same as in standard languages such as C, where aliasing of arguments to a function call is not well-defined, as the implementation may make copies of the aliased arguments and lose the aliasing relationship. Conversely, if two index launches use the same view of an array, or only read from arrays, then there is no restriction on fusion.

C. Examples

To illustrate the effect of the fusion constraints in practice, we consider two additional examples, a Jacobi solver and a 1D three point stencil.

1) Communication Absence and Launch Space Equivalence: In a Jacobi solver, given matrix `A` and vector `b`, we iteratively solve for `x`:

```
d = np.diag(A)
R = A - np.diag(d)
for i in range(iters):
    x = (b - np.dot(R, x)) / d
```

Each iteration of the for loop involves the following tasks and launch spaces (we omit one auxiliary task for initializing array x).

```
temp1 = DOT(R, x, launch_space=(2,2))
temp2 = SUB(b, temp1,launch_space=(4,1))
x     = DIV(temp2, d, launch_space=(4,1))
```

If, for example, there are four processors available, the product of the dimensions in a launch space must be less than or equal to four. The dot product task above uses a 2x2 launch space that requires a 2D domain decomposition of the matrix-vector operation, whereas the following two vector operations operate on 1D arrays with a 1-dimensional launch space. As each index task (including fused tasks) in cuNumeric must have exactly one launch space, the Launch Space Equivalence constraint allows the SUB and DIV tasks to be fused, but not the DOT product task. Of course, since 2*2=4*1, the total size of the launch spaces are the same, so with some rewriting of the tasks the launch spaces could be made compatible. However, fusion would still fail; a

dot product involves inter-processor communication to reduce partial results and thus violates the Communication Absence constraint.

2) Producer-Consumer Restriction and Projection Equivalence: We modify our earlier three point stencil to involve weights.

```
east    = input[0:-2]
central = input[1:-1]
west    = input[2:N]
for i in range(num_iters):
    output     = east+west
    central[:] = .5*output
```

Executing two iterations of the `for` loop issues the following index task launches:

```
#iteration 1
output = ADD(east, west, launch_space=(4,1))
central = MUL(.5, output, launch_space=(4,1))

#iteration 2
output = ADD(east, west, launch_space=(4,1))
central = MUL(.5, output, launch_space=(4,1))
```

In this case, all four index launches have the same launch space, and none of the subtasks require inter-processor communication. However, if all four subtasks are fused, we will write to `central` in the second subtask (MUL) and then read from views `east` and `west` in the third subtask (ADD). In other words, in the fused subtask, we would write to one view of the input array, and then subsequently read from a different view of the same array, violating the Producer-Consumer Restriction. In this case, only separate fusions of index launches 1-2 and 3-4 are permitted.

It is more difficult to give a small and natural example where Projection Equivalence comes into play, as the projection functions depend on how arrays are partitioned. Partitioning decisions are made by cuNumeric's auto-partioning heuristics, which may or may not generate multiple partitions of the same array such that the partitions have identical projection functions. In practice, the heuristics will generally, but not always, partition the same array the same way in different tasks. In our three point stencil program, for example, if the partitioner decided to partition the input array differently for the first and second index launches, then they could not be fused.

IV. Results

The results of distributed task fusion on five benchmarks are shown in Table I. Experiments were conducted on the Piz Daint supercomputer, which contains one P100 GPU per node. All experiments were conducted on 32 node allocations against an unfused baseline to demonstrate the speedups resulting from elimination of task overheads. For four of the five benchmarks, speedups between 1.1x and 1.55x are

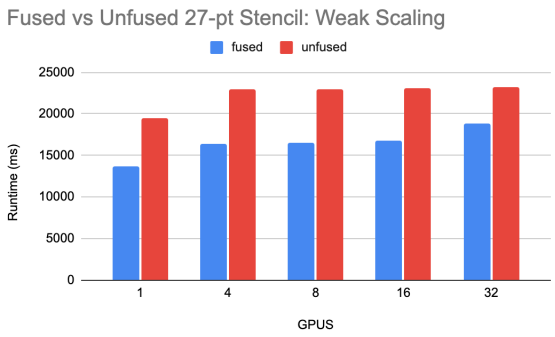

Figure 5: Weak scaling of the stencil benchmark, with fusion turned on and turned off.

observed. We also include one benchmark (Jacobi Iteration) that contains little fusable work, in order to characterize the performance and overhead of our task fusion when there is little opportunity to fuse. The Mandelbrot benchmark is adapted from the NPBench [32] benchmark suite, while remaining benchmarks are adapted from the cuNumeric repository.

As the workload per GPU decreases, runtime overheads will represent a higher percentage of execution time, meaning that task fusion speedups should increase. We illustrate this trend with the Black-Scholes benchmark in Table IV, running variable sized workloads on a 4 GPU allocation on Piz Daint.

We show in Table III that each constraint is used in at least one benchmark. As shown in Listing 2, constraints are applied to the task window one at a time, which saves work as subsequent constraints are not tested at positions where there is already a barrier. The Communication Absence constraint is the cheapest to test and generally the most restrictive, so we test it first and then test the more expensive constraints at positions where Communication Absence did not apply. Thus, the numbers in Table III do not represent all the places the constraints other than Communication Absence could have been used, just the places where they were used because no previously tested constraint applied.

While fusion always improves the performance of the stencil benchmarks, the speedup can decrease when weak scaling across more processors, as seen in Table I and Figure 5. This is not a limitation of our implementation, but a behavior inherent to task fusion due to the decrease in task-level parallelism that occurs when tasks are aggregated and coarsened. Specifically, with coarser tasks there is less opportunity to overlap the communication required by one index launch with the computation performed by another.

In the 27-pt stencil benchmark, for example, each fusable subtask operates on shifted views of the same input tensor (with each view corresponding to a

Speedup from Task Fusion on 1-32 GPUs					
Benchmark	1 GPU	4 GPUs	8 GPUs	16 GPUs	32 GPUs
27-pt (3D) stencil	1.43x	1.40x	1.39x	1.38x	1.23x
Black-Scholes	1.55x	1.43x	1.45x	1.42x	1.44x
Mandelbrot	1.17x	1.13x	1.12x	1.12x	1.11x
Logistic Regression	1.18x	1.13x	1.11x	1.16x	1.15x
Jacobi Iteration	1.04x	1.00x	.98x	1.02x	.96x

Table I: Speedups resulting from task fusion, computed relative to the same program on the same number of nodes without task fusion.

Percentage of All Tasks Fused and Metrics on Length of Fused Tasks				
Benchmark	min length	avg length	max length	% tasks fused
27-pt (3D) stencil	2	18	47	98%
Black-Scholes	3	31.4	49	98%
Mandelbrot	2	49.8	50	100%
Logistic Regression	3	3	3	75%
Jacobi Iteration	2	2.75	3	63%

Table II: Summary statistics of the fused tasks' sizes (number of encapsulated subtasks) in each benchmark, and percentage of all tasks that are fused. Unfused tasks are not considered in the calculation of min, max, and avg fused task sizes.

Barriers Generated by Each Constraint				
Benchmark	CommunicationAbsence	LaunchSpace	ProducerConsumer	Projection
27-pt (3D) stencil	0%	.1%	49.5%	50.4%
Black-Scholes	100%	0%	0%	0%
Mandelbrot	0%	0%	0%	0%
Logistic Regression	75%	25%	0%	0%
Jacobi Iteration	97.8%	2.2%	0%	0%

Table III: Percentage of total fusion barriers generated by each fusion constraint. Constraints are applied in order shown, e.g. a ProjectionEquivalance barrier is not be placed if a CommunicationAbsence barrier was already placed in the same place. Small percentages generally reflect barriers placed around initialization tasks.

Black-Scholes speedups w/varying workloads.					
N	51200	25600	6400	3200	1600
Speedup	1.39x	1.42x	1.46x	1.45x	1.44x

Table IV: N = number of options to price, in thousands.

different point in the stencil). Ensuring that each processor has its respective shifted partition necessitates some data movement induced by the index launch. Such communication is allowed by fusion, as this movement can precede all of the fusable subtasks' execution and does not need to occur between or during subtasks's execution. As fusion reduces opportunities to overlap subtasks' execution with the data movement, fusion speedups in the stencil can decrease with the higher latencies that come with larger numbers of nodes.

The drop from 1.38x to 1.23x speedup going from 16 to 32 GPUs in the 27-pt stencil is likely due to these increased communication times. The 32 GPU allocation on Piz Daint is not a contiguous set of nodes (but rather a set of subsets, which each contain contiguous nodes). In this case, the allocation contains a contiguous 16 node allocation, so scaling from 16 to 32 nodes introduces higher communication costs that are harder to mask with coarser, fused tasks. The Black-Scholes and Mandelbrot benchmarks involve no memory movement during their execution, meaning that task fusion speedup does not decay when weak scaling up and remains relatively constant.

The Jacobi Iteration benchmark is shown to demonstrate what happens when there is little fusable work. The associated speedups and slowdowns are mostly inconsequential. In practice, as fusion has a small overhead, our implementation could simply disable fusion if it detects little fusable work. This relative lack of potential fusion is illustrated in Table II, which shows minimum, average and maximum fused task sizes for each of our benchmarks, as well as the percentage of total tasks that do get fused. All four of these metrics are the lowest for the Jacobi Iteration benchmark compared to those of the other four benchmarks.

Finally, we note that in the Black-Scholes and stencil benchmarks, about half of the speedup comes from task fusion itself, and the other half from a complementary optimization that primarily enables higher levels of fusion, but also yields performance benefits in itself. In cuNumeric, if a task t applies a binary operation to a 64 bit floating point constant and a 32 bit floating point array or vice versa, the constant

must be converted to the array's type. Normally cuNumeric dispatches an entire task to handle this conversion, which generates a future as its result. Since we must have the value of the future to execute the task t, the entire window of buffered operations is flushed, effectively making the tasks unfusable. By doing all such type conversions inline, outside of any task, we eliminate the conversion task entirely and increase the amount of potential fusion.

V. Related Work

Fusing individual tasks is commonly done in practice, and has been utilized since the advent of multiprocessors [26]. Distributed task fusion is different in that we are fusing multiple index launches, which may communicate with data distributions in ways that make classic fusion semantically incorrect.

A. Kernel Fusion

It is useful to make a distinction between *kernel fusion* and our presented form of task fusion. Kernel fusion refers to transformations for generating a more efficient binary, requiring recompilation of code when such transformations (e.g., loop optimizations) are performed at run-time. These transformations involve classic loop optimizations and are often complemented by algebraic simplifications of mathematical operations that reduce FLOP counts. On a GPU, where some loops are implicit as a grid of threads, analogous optimizations can be made by fusing separate passes over memory into a single GPU compute-kernel, thus eliminating the need to write and then read temporary results from DRAM. Task fusion, however, refers to the aggregation of separate tasks into a single task launch. The program binaries and loops remains the same, as the primary objective is not to apply program transformations via JIT compilation, but to amortize the overheads associated with launching and managing a series of tasks in a parallel, distributed system. With task fusion, finer granularity tasks can be efficiently supported, as the management overhead is paid only once per task launch.

While we distinguish between task fusion and kernel fusion, they share similarities that warrant a brief discussion of the existing kernel fusion literature. Multiple frameworks, including JAX [13], Bohrium [17], and Numba [18] are capable of JIT compiling Python functions to C while applying loop transformations and parallelization operations, offering significant speedup over a baseline such as CPython [2]. There is also considerable recent interest in kernel fusion in machine learning frameworks, such as PyTorch's [23] TorchScript [11], Tensorflow's [3] XLA compiler [25] (which forms the backbone of JAX), and Nvidia TensorRT [29], all of which provide kernel fusion capabilities. Such optimizations are warranted by the existence of common layer patterns in neural networks (e.g. convolution-ReLU pairs), which are generally amenable to kernel fusion on CPUs, GPUs and other accelerators.

Weld [22] is a recent framework that allows for kernel fusion of operations from different libraries, under the condition that each library can be compiled to a common, Weld-specific IR. Similar to task fusion, Weld operations are deferred before being JIT compiled into an optimized binary.

B. Parallel Runtimes

We have implemented distributed task fusion in Legate, the backend for cuNumeric [5]. Task fusion comprises a new stage in Legate's task launching pipeline. Legate is a tasking layer for Python. Legate itself is built on Legion, a more general-purpose and lower level distributed runtime [8]. Legate partitions arrays, determines the associated projection functions, and then generates tasks for each cuNumeric call before submitting the tasks and partitioning information to Legion.

StarPU [4] and PaRSEC [15] are two runtimes with some similarities to Legion, in particular targeting high performance computing applications and supporting relatively fine-grained distributed task execution. While neither has a programming model with a notion of group task launches directly comparable to Legion's index launches, the issues around fusing a sequence of related subtasks across processors are nevertheless the same, though expressed differently at the program level.

On a single GPU, the Pagoda [30] and Pastel-Palettes [20] runtimes manage execution of operations within a GPU single kernel. The runtime and control logic of these systems is implemented directly in a CUDA kernel, which complicates the engineering of implementing task fusion.

Ray [21] is a recent framework for distributed Python applications that allows for the asynchronous launch of parallel Python tasks. While Ray's core library is a general purpose framework for distributed applications, Ray primarily targets machine learning and reinforcement learning workloads. Ray tasks are not fundamentally distributed like those in index launches, but Ray tasks are meant to be coarse, so it could potentially benefit from a form of task fusion.

Dask, which has a similar programming model to Ray, has a notion of task fusion in which the user can aggregate tasks to be collectively scheduled as one unit [1], [24]. In Dask, however, the goal of task fusion is primarily to guarantee that certain tasks run on a single processor, rather than to reduce runtime overheads.

VI. Future work

One area of future work involves Legate's auto-partitioning mechanism, which determines a map be-

tween processors and partitions of arrays. Currently, the partitioner makes decisions based on individual tasks alone, and does not have any knowledge of task fusion. There are cases when multiple valid partitions of the data are possible, specifically in large fused tasks that may involve many different views of the same input tensor (such as the 27-pt stencil). In such cases, different partitioning strategies can result in different amounts of pre-task execution data movement. Hence, the auto-partitioner could be improved with heuristics that determine a reasonable partitioning strategy that results in data locality reasonably amenable to each of the subtasks' data needs.

Another direction is related to the previously mentioned possibility of fusing tasks that are not contiguous in a task window. One possible strategy could be to fuse only dependent tasks, forming fused tasks that internally have mostly sequential dependencies. Such an approach could conceivably retain more task parallelism while still reducing run-time overheads substantially.

Our algorithm in Listing 2 is not the only way to generate subsequences of fusable tasks. If for example, given a sequence of four tasks, a constraint determines that tasks 1 and 4 are not fusable, then our algorithm will place a barrier immediately before the first unfusable task, i.e. between tasks 3 and 4. It is equally valid to place the barrier anywhere between tasks 1 and 4. However, we have not observed any instances in our benchmarks where such flexibility actually occurs.

Finally, once tasks have been fused, it is then theoretically possible to further optimize the fused task with kernel and loop fusion, especially in the interest of reducing accesses to memory. While this would require JIT compilation and a backend IR for rewrites, it would likely generate further speedup.

VII. Conclusion

We have presented distributed task fusion, a runtime optimization for reducing the overheads associated with fine-grain group task launches in task-based systems. We describe and implement this optimization in the context of cuNumeric workloads, as cuNumeric's runtime performs automatic partitioning of data and index launches that often produce very small tasks. We have described four conditions under which task fusion is permissible and demonstrated the benefits of task fusion on several benchmarks, observing 1.1x - 1.5x speedups on several workloads scaled up to 32 GPU nodes without any changes to the underlying program.

Acknowledgements

We would like to thank our anonymous reviewers for their feedback related to improving this manuscript. We would like to thank Manolis Papadakis, Elliot Slaughter, and Mike Bauer for their support during development.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. This work was also supported by a grant from the Swiss National Supercomputing Centre(CSCS) under project ID d81.

References

[1] Dask optimization. https://docs.dask.org/en/stable/optimize.html. Accessed: 2022-08-02.

[2] python/cpython. https://github.com/python/cpython, 2022. Accessed: 2022-08-02.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, 2009.

[5] Michael Bauer and Michael Garland. Legate numpy: Accelerated and distributed array computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[6] Michael Bauer, Wonchan Lee, Manolis Papadakis, Marcin Zalewski, and Michael Garland. Supercomputing in python with legate. *Computing in Science & Engineering*, 2021.

[7] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[9] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices*, 1995.

[10] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003.

[11] Zachary DeVito. Torchscript: Optimized execution of pytorch programs. *NeurIPS, December*, 14, 2019.

[12] Robert Dyer. Task fusion: Improving utilization of multi-user clusters. In *Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity*, 2013.

[13] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[14] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 2020.

[15] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2017.

[16] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993.

[17] Mads RB Kristensen, Simon AF Lund, Troels Blum, Kenneth Skovhede, and Brian Vinter. Bohrium: unmodified numpy code on cpu, gpu, and cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[18] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[19] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. 2014.

[20] Qianxiang Ma and Rio Yokota. Runtime system for gpu-based hierarchical lu factorization. 2019.

[21] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[22] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. 2018.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[24] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, 2015.

[25] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[26] Vivek Sarkar. *Partitioning and scheduling parallel programs for execution on multiprocessors*. Stanford University, 1987.

[27] Elliott Slaughter, Wei Wu, Yuankun Fu, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, et al. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[28] Rupanshu Soi, Michael Bauer, Sean Treichler, Manolis Papadakis, Wonchan Lee, Patrick McCormick, Alex Aiken, and Elliott Slaughter. Index launches: scalable, flexible representation of parallel task groups. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[29] Han Vanholder. Efficient inference with tensorrt. In *GPU Technology Conference*, 2016.

[30] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G Rogers. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. *ACM SIGPLAN Notices*, 2017.

[31] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 2016.

[32] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. Npbench: A benchmarking suite for high-performance numpy. In *Proceedings of the ACM International Conference on Supercomputing*, 2021.

APPENDIX

A. Appendix A: Artifact Evaluation/Description Appendix

1) Summary of Experiments: All experiments were run on the Piz Daint system in the Swiss National Supercomputing Centre. We used allocations of up to 32 nodes. All code and scripts used to generate results are available in public github branches of legate.core and cuNumeric

2) Artifact Availability: DOIs for repositories of Legate.core and cuNumeric containing the task fusion implementation are provided:
Legate.core:
https://zenodo.org/badge/latestdoi/411056638
cuNumeric:
https://zenodo.org/badge/latestdoi/409288575

Links to the github repos are provided in Subsection A5.

3) Piz Daint Hardware: :
Processor: Xeon E5-2690v3 12C 2.6GHz
GPU: Nvidia Tesla P100
Interconnect: Aries interconnect
compute node CPU info (lscpu):

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
Address sizes:         46 bits physical, 48
    bits virtual
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5
    -2690 v3 @ 2.60GHz
Stepping:              2
CPU MHz:               1452.011
CPU max MHz:           2601.0000
CPU min MHz:           1200.0000
BogoMIPS:              5199.78
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-23
```

4) Software Environment: :
OS: SUSE Linux Enterprise Server 15 SP2
OS Version: 15.2
CUDA Compiler Version: 11.1
CUDA arch: sm_60
NVIDIA Driver Version: 470.57.02
Python Version: 3.8.13(Anaconda)
C++ compiler version: g++ (SUSE Linux) 7.5.0

5) Software Installation: Both legate.core and cuNumeric (previously known as legate.numpy) must be cloned into the same parent directory, and then built.

cuNumeric repo:
git@github.com:shivsundram/legate.numpy.git
legate.core repo:
git@github.com:shivsundram/legate.core.git

For both repositories, the partitionExp2 branch should be used. Directions for installing both are on the repositories' github pages and repeated here:
https://github.com/shivsundram/legate.core
https://github.com/shivsundram/legate.numpy

OpenBLAS should also be downloaded and built within the same directory as legate.core and cuNumeric/legate.numpy, from the following repo:
https://github.com/xianyi/OpenBLAS

The cuNumeric conda environment should be downloaded using:

```
conda env create -n legate -f conda/
    cunumeric_dev.yml
```

legate.core can then be installed via:

```
cd legate.core
./install.py --gasnet --conduit aries --cuda
     --arch pascal --with-cuda /usr/local/
    cuda-11.1/
```

cuNumeric can then be installed via:

```
cd ../legate.numpy
mkdir install
python setup.py --prefix ./install --with-
    core ../legate.core/install/ --with-
    openblas ../OpenBLAS/install/
```

6) Experiment Scripts: On Piz Daint first request a 32 node allocation with

```
salloc -N 32 -C gpu -A ACCOUNT_NAME
```

From the legate.numpy directory, results from Table I can be reproduced with

```
conda activate legate

#run 27pt stencil
./sweep.sh -w

#Jacobi
./sweep.sh -j

#Black-Scholes
./sweep.sh -b

#Mandelbrot
./sweep.sh -m

#Logistic Regression
./sweep.sh -l
```

From the legate.numpy directory, results from Table IV can be reproduced with

```
conda activate legate
#black scholes on 4 nodes with various
    workloads
./sweep.sh -v
```

The above can be done with a 4 node allocation, though it can be run in the same 32 node allocation requested before.

7) System Environment: The system must be accessed first via the gateway ela.cscs.ch, from which one can ssh into daint.cscs.ch.

We use the standard environment provided by Piz Daint, with two additional submodules imported via the following commands to enable GPU support:

```
module load daint-gpu
module load cudatoolkit/11.1.0_3.39-4.1
    __g484e319
```

8) Statistical Information of Benchmarking Output: For reproducibility purposes, the tables below provide statistics on each benchmark mentioned above (using the commands mentioned in this appendix). This data was collected on Piz Daint and used to calculate the speedups in Table I and Table IV. Each benchmark was run 12 times. The minimum and maximum times were discarded when calculating the average runtimes and their standard deviations. All times are in milliseconds.

27-pt stencil: Average Runtimes (ms), Standard Deviation (ms), nSamples, Input Size						
nGPUs	Avg Fused Runtime	Stdev Fused	Avg Unfused Runtime	Stdev Unfused	nSamples	Input size (tensor volume)
1	13644.17	12.23	19520	30.63	10	6751269
4	16394.02979	23.13587	22981.549	23.42940097	10	27005076
8	16522.64399	19.38054138	22935.1987	18.12132	10	54010152
16	16789.2873	23.23641161	23141.67	15.86677094	10	108020304
32	18896.932	419.494833	23228.158	89.2769162	10	216040608

Table V: Statistical information regarding data in Table I for stencil example. Each sample performs 500 iterations of the stencil loop.

Black Scholes: Average Runtimes (ms), Standard Deviation (ms), nSamples, Input Size						
nGPUs	Avg Fused Runtime (ms)	Stdev Fused	Avg Unfused Runtime (ms)	Stdev Unfused	nSamples	Input size (thousands of options)
1	7761.2302	8.211	12061.6361	20.796	10	3200
4	9850.6536	30.6158	14075.568	19.309	10	12800
8	9948.499	5.50006	14472.0861	15.70065	10	25600
16	10011.3695	13.9958	14189.81399	37.845709	10	51200
32	10025.0068	14.889	14400.74549	20.0482	10	102400

Table VI: Statistical information regarding data in Table I for Black Scholes example.

Mandelbrot: Average Runtimes (ms), Standard Deviation (ms), nSamples, Input Size						
nGPUs	Avg Fused Runtime (ms)	Stdev Fused	Avg Unfused Runtime (ms)	Stdev Unfused	nSamples	Input size (nPixels)
1	6956.4325	11.846573	8154.1689	27.54872	10	1000000
4	8705.2407	23.530865	9808.867	19.11836667	10	4000000
8	8817.6824	19.68045065	9792.0721	37.620265	10	8000000
16	8934.8546	71.07874	9891.1459	92.09724704	10	16000000
32	8963.2311	177.1096279	9909.8162	91.42899066	10	32000000

Table VII: Statistical information regarding data in Table I for Mandelbrot example. Each sample performs 500 iterations on the image.

Logistic Regression: Average Runtimes (ms), Standard Deviation (ms), nSamples, Input Size						
nGPUs	Avg Fused Runtime	Stdev Fused	Avg Unfused Runtime	Stdev Unfused	nSamples	Input size (nPoints)
1	4775.507399	13.6548	5658.874	23.83421	10	1600000
4	6058.1377	14.86975	6871.0497	293.36	10	6400000
8	6154.466	12.9798	6836.79	328.867	10	12800000
16	6457.2917	19.0241	7513.4415	300.3213	10	25600000
32	11258.193	164.219881	12919.134	301.016362	10	51200000

Table VIII: Statistical information regarding data in Table I for Logistic Regression example. Each data point has 32 features. Each sample performs 500 iterations.

Jacobi Iteration: Average Runtimes (ms), Standard Deviation (ms), nSamples, Input Size						
nGPUs	Avg Fused Runtime (ms)	Stdev Fused	Avg Unfused Runtime (ms)	Stdev Unfused	nSamples	Input size (matrix size)
1	9225.5936	11.5285	9612.5388	23.9947	10	40955775
4	11833.4186	20.599334	11879.41449	31.008652	10	163823100
8	14772.8015	1823.002094	14542.982	846.70585	10	327646201
16	14092.8893	245.711837	14422.4889	186.65693	10	655292402
32	32301.1099	2394.2737	31075.4364	806.3412	10	1310584804

Table IX: statistical information regarding data in Table I for the Jacobi example. Each sample performs 5000 iterations of Jacobi iteration.

Black Scholes w/varying workloads (4 GPUs): Average Runtimes (ms), Standard Deviation (ms), nSamples, Input Size						
nGPUS	Avg Fused Runtime	Stdev Fused	Avg Unfused Runtime	Stdev Unfused	nSamples	Input size (thousands of optinos)
4	9908.701	11.23012	13763.407	26.8776	10	51200
4	9845.357	14.6384085	14065.97	22.307366	10	25600
4	9818.412	11.325159	14375.0252	21.7134586	10	6400
4	9823.119	10.57022	14214.1891	18.082326	10	3200
4	9832.4827	14.36487186	14179.728	15.992181	10	1600

Table X: Statistical information regarding data in Table IV for Black Scholes example w/ varying workloads.