Two optimizations to GPU code generation in the Futhark compiler

Christian Påbøl and Anders Holst

Supervisor: Cosmin E. Oancea

February 14, 2024

Contents

1 Introduction

This report documents our work on two modifications made to GPU code generation in the Futhark compiler. The first focuses on implementing thread block virtualization in single-pass segscan kernels and is not a performance optimization *per se*, although it may enable, while the second is an optimization to sequentialization in certain segreduce kernels, specifically reductions with noncommutative, primitive and non-vectorized operators (more on this later).

Futhark[1] is a functional array programming language with an optimizing compiler[2] targetting primarily, but not exclusively, GPU execution. The compiler is in ongoing development but is showing promising results in various domains. While the compiler generates efficient and optimized code for SOACs such as scan and reduction, the window of opportunity for optimization is always open.

In this report, we first give a small bit of relevant theory on parallel scan and reduction algorithms in section 2, as well as some practical theory behind the optimizations we want to make. In section 3, we detail important steps taken in implementation of both changes. Finally, in sections 4 and 5, we present a validation and benchmarking plan, including experimentation with parameterization of the modified reduce kernel; execution of said plan; the results of testing; and finally, an analysis and conclusion upon the results.

2 Background

2.1 Terminology and abbreviations

We use CUDA terminology in all discussion of GPU hardware and GPU code. We use **tblock** as an abbreviation for thread block; `TBLOCK_SIZE` and `NUM_TBLOCKS` to denote the size of a tblock and the number of tblocks for a given kernel, and `NUM_THREADS` = `NUM_TBLOCKS` $*$ `TBLOCK_SIZE`.

We use "tblock-reduction" to mean a tblock-level parallel reduction over `TBLOCK_SIZE` elements in shared memory.

2.2 The reduce and scan operations

Let $(S,\ \oplus,\ e)$ be a monoid, where S is a set closed under binary associative operator $\oplus$ with neutral element $e \in S$, and let $\mathbf{x} \subseteq S$ be a sequence of elements from S with $\mathbf{x} = [x_1, x_2, \ldots, x_n]$. Then reduce $\oplus$ e $\mathbf{x}$ is the reduction of $\mathbf{x}$ to a single element $x \in S$ through pairwise application of $\oplus$ to neighboring elements from $\mathbf{x}$:

$$\text{reduce} \oplus e\ \mathbf{x} = e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n. \tag{1}$$

Note that if $\oplus$ is additionally *commutative*, then reduce $\oplus$ e $\mathbf{x}'$ produces the same result for any permutation $\mathbf{x}'$ of the sequence $\mathbf{x}$.

scan $\oplus$ e $\mathbf{x}$ is essentially the same computation, but the result is a sequence (or submultiset) $\mathbf{y}$ of each partial results s.t. $\mathbf{y} \subseteq S$ and $|\mathbf{y}| = |\mathbf{x}|$:

$$\text{scan} \oplus e\ \mathbf{x}\ = [e \oplus x_1,\ e \oplus x_1 \oplus x_2,\ \ldots,\ \text{reduce} \oplus e\ \mathbf{x}]. \tag{2}$$

[1] `futhark-lang.org`

[2] `https://github.com/diku-dk/futhark`

```
1 reduce(op, ne, xs) =
2
3   acc = ne
4   for (i = 0; i < len(xs); i++)
5     acc = op(acc, xs[i])
6   return acc
```

(a) Reduce.

```
1 scan(op, ne, xs) =
2   ys = array(len(xs), elemtype(xs))
3   acc = ne
4   for (i = 0; i < len(xs); i++)
5     ys[i] = acc = op(acc, xs[i])
6   return ys
```

(b) Scan.

Figure 1: Pseudocode for naive and sequential reduce and scan.

As an example, using the set of reals $\mathbb{R}$, addition, and zero as neutral element, then reduce $(+)$ 0 $\mathbf{y}$, where $\mathbf{y}$ is a sequence of $\mathbb{R}$, is the sum of elements in $\mathbf{y}$, whereas scan $(+)$ 0 $\mathbf{y}$ is the sequence of all $|\mathbf{y}|$ prefix sums of $\mathbf{y}$ (similar to e.g. `np.cumsum`).

2.2.1 Sequential reduce and scan in code

Figure 1 shows C-like but high-level pseudocode for sequential reduce and scan. In line 5 of each snippet, the current accumulator variable `acc` is updated by application of the binary operator between `acc` and the next input element. As such there is a clear dependency on `acc` between the i'th and $(i-1)$'st iteration, hence neither algorithm is inherently parallel. However, efficient parallel algorithms exist for each.

2.3 High-level overview of existing Futhark implementations

In this section, we present and discuss existing Futhark code generation for parallel scan and reduction.

2.4 Single-pass parallel scan

The Futhark compiler can generate code for both single- and two-pass scan. The latter is always used on the OpenCL backend, and on the CUDA and HIP backends when one or more scan operators is non-primitive. In all other cases, single-pass scan is used. The authors of the algorithm give a detailed description of the single-pass scan algorithm for GPUs (as well as some of the optimizations employed in Futhark) in [1], while details of the implementation into the Futhark compiler can be found in [2] and [3].

In the typical two-pass scan, each tblock computes a partial *reduction* of its assigned partition of the input, requiring n global memory reads; next, per-block prefixes are computed using a single intra-block scan; finally, each tblock scans its own partition, incorporating the prefix from previous partitions, requiring n global reads and n global writes. In total, $2n$ and n global memory reads and writes, respectively.

Meanwhile, the algorithm in question is called *single-pass* because it requires only a single pass over the input, performing n reads and n writes from/to global memory at the cost of some extra bookkeeping and synchronization overhead. In order to avoid the initial pass over global memory, each thread block incorporates the prefix from previous partitions *as* the scan is computed. This implies that the prefixes for partitions $0, \ldots, k-1$ must be computed before tblock k scans the k'th partition.

Each tblock is free to compute and publish its own *aggregate* prefix in parallel, but must stall until it obtains the prefix, since its scan result depends on it. In addition, in order to avoid the situation where the GPU is fully occupied by executing tblocks while a tblock responsible for a previous prefix has not spawned, we must guarantee that all tblocks $i = 0, \ldots, k-1$ have at least spawned, but not necessarily finished execution, before tblock k spawns and begins execution.

This must be specifically handled because CUDA does not guarantee that tblocks are spawned in order of their tblock index (and neither does e.g. OpenCL), and the solution is to let tblocks obtain dynamic tblock indices via an atomically accessed scalar counter variable in global memory. We are somewhat abusing the CUDA/GPU programming model (correct execution should not depend on the order of spawning of tblocks), but the behavior is well-defined and deterministic when dynamic indexing is used peroperly, so we accept the abuse.

2.4.1 Tblock-virtualization and single-pass scan in Futhark

Since Futhark compiled binaries allow the user to set kernel parameters such as `NUM_TBLOCKS` on a per-program basis (rather than per kernel, as is usual in hand-written CUDA code), code generated for the different kernels must support *tblock-virtualization* when the number of physical tblocks spawned for a kernel invocation is smaller than the number of tblocks required for the given problem size. With tblock-virtualization, we can use a fixed number of tblocks to emulate the work and execution of arbitrarily many tblocks. Tblock-virtualization is simple and typically only involves wrapping parts or all of a kernel in a loop over some number of virtualization loop iterations (which varies per physical tblock if the number of physical tblocks does not divide the number of required tblocks; as an example, with 2 physical tblocks and 3 required tblocks, one physical tblock will loop twice while the other will loop only once).

One benefit of using a fixed number of physical tblocks has to do with *memory expansion*, which we shall not go into specific detail with in this report, except to say that it is a Futhark compiler pass which attempts to eliminate per-thread global memory array allocations by hoisting them outwards to global scope. Each thread can then access their own chunk of the hoisted array, possibly with some stride to ensure coalesced access. In practice this technique requires that the number of physical tblocks is bounded, since allocations for all eventual physical threads must be made ahead of time.

In regards to reduce and scan, memory expansion is e.g. used for operators with array operands and for primitive but vectorized operators. Currently, two-pass scan is chosen on the CUDA backend whenever the scan operator has array operands, while single-pass scan fires for all other scan operators, including vectorized primitive operators. This implies that memory expansion is required in select cases of single-pass scan.

However, the current Futhark implementation of single-pass scan ignores the `NUM_TBLOCKS` user-configurable parameter that is otherwise passed around in the IR, and instead spawns exactly as many physical tblocks as the number of required tblocks by the problem size. Meanwhile, the memory expansion pass takes its information from the user-chosen `NUM_TBLOCKS`. This means threads from tblocks outside the number of required tblocks access memory from outside the bounds of the expanded allocation(s). This is a bug which should be solved.

```
noncomm_reduce_GPU_kernel_OLD(op, ne, xs, NUM_TBLOCKS, TBLOCK_SIZE) =
  /* STAGE ONE */
  Q = ceil(len(xs) / (NUM_TBLOCKS * TBLOCK_SIZE))
  for (k = 0; k < Q; k++) {
    /* 1) collective copy of TBLOCK_SIZE sized slice of xs into shared mem.
     * 2) tblock-reduction of shared mem and accumulation.
     */
  }
  /* STAGE TWO
   * ...
   */
```

Figure 2: High-level pseudocode for the current Futhark generated GPU code for non-commutative reductions.

2.4.2 Efficient parallel reduction

Many algorithms for parallel reductions suitable for GPU execution exist. However, we shall only concern ourselves with the method currently implemented in the Futhark GPU backends, since we will only be making a modification to this.

The two main obstacles to an efficient implementation are 1) coalesced access to global memory and 2) efficient sequentialization.

For commutative reductions[3], coalesced access is easily obtained since, as mentioned earlier, any order of application of the binary operator yields the same result – hence threads are free to apply the reduction with a stride in global memory, and thus coalesced access is trivially obtained. Threads each reduce $Q = \lceil N/\texttt{NUM\_THREADS} \rceil$ elements *sequentially* in a virtualization loop of Q iterations, and subsequently write their result to shared memory; each tblock then performs a tblock-reduction over `TBLOCK_SIZE` elements; finally, a single tblock reduces the `NUM_TBLOCKS` number of results to a single value.

For non-commutative reductions we must take a little extra care because the operator(s) must be applied to elements in order. Hence the tblock-reduction that succeeded the virtualization loop is now moved inside the loop, and thus we now have Q parallel reductions rather than 1, where $Q = N/\texttt{NUM\_THREADS}$ is again the sequentialization factor.

A tblock-reduction has depth $5 + \log_2(\texttt{TBLOCK\_SIZE}/32) \leq 10$ (given $\texttt{TBLOCK\_SIZE} \leq 1024$) per fused operator, and hence is not particularly expensive in itself, however it adds significant overhead to the virtualization loop as opposed to the commutative kernel, and hence a significant degradation in performance is to be expected when the operator is non-commutative.

Figure 2 shows pseudocode for this method, and this is what is followed by the code generated by the current GPU backends of the Futhark compiler for non-commutative reductions. Note that the kernel is hyperparameterized over `NUM_TBLOCKS` and `TBLOCK_SIZE` only.

[3] A reduction is, in this context, said to be commutative if the reduction operator is a fusion of one or more commutative binary operators. As examples, fusing a summation and a product yields a commutative reduction since both addition and multiplication is commutative, whilst fusing a summation and an MSS yields a non-commutative reduction because the MSS binary operator is non-commutative.

3 Implementation

3.1 Tblock-virtualization for the single-pass scan kernel

To make the single-pass scan kernel respect the NUM_TBLOCKS embedded in the IR, we want to extend the kernel with tblock-virtualization. For the single-pass kernel, this is as simple as wrapping the entire kernel body in a virtualization loop of

$$\texttt{NUM\_VIRT\_ITERS} = \left\lceil \frac{\texttt{NUM\_LOGICAL\_BLOCKS} - \texttt{tBlockIndex}_{\text{physical}}}{\texttt{NUM\_TBLOCKS}_{\text{physical}}} \right\rceil \tag{3}$$

iterations, where $\texttt{tBlockIndex}_{\text{physical}}$ is the *physical* index of the given *physical* tblock, and NUM_LOGICAL_BLOCKS is the number of logical blocks in the entire input. A logical block is defined as exactly the amount of input processed by a single tblock, so NUM_LOGICAL_BLOCKS is also the number of physical tblocks we would otherwise use without tblock-virtualization. This formula is lifted from an existing, different use of tblock-virtualization in the compiler.

Note that NUM_VIRT_ITERS varies between tblocks. Equation (3) ensures that the difference in the number of virtualization loop iterations between any two physical tblocks is at most 1. To avoid spawning tblocks with no work to do, the compiler ensures that the number of physical tblocks spawned is the minimum between the requested and the required number of tblocks – this assertion is already in place.

Recall that the single-pass scan kernel uses dynamic indexing of tblocks. This means physical tblocks must fetch a *new* dynamic tblock index at the head of *each* virtualization loop, however because the process of a physical tblock entering a new virtualization loop iteration and fetching its next dynamic index corresponds directly to a new physical tblock taking its place and fetching the same dynamic index, this should not raise any concerns w.r.t. synchronization.

Optimal NUM_TBLOCKS

Different values of NUM_TBLOCKS is likely to have different effects on performance. Immediately, one might argue that the optimal value is one which maximizes occupancy without going overboard on the number of physical tblocks that fits on the given device with the given TBLOCK_SIZE.

3.2 Improving sequentialization in reduce

Our improvement to sequentialization is a decrease in the number of tblock-reductions in the stage one main loop by a factor CHUNK. The loop is essentially stripmined with a factor CHUNK by scaling Q by an additional factor $1/\texttt{CHUNK}$. To account for the stripmining, each thread must read CHUNK elements in each iteration, up from 1, for a total collective copy of size $\texttt{TBLOCK\_SIZE} \cdot \texttt{CHUNK}$ per tblock.

Elements are read into CHUNK-sized per-thread register memory arrays, which are sequentially reduced to a single value going into the tblock-reduction.

With k fused reduce operators, we need to read and reduce elements from k different inputs. k collective copies from global into k CHUNK-size register arrays via shared memory are performed in sequence, followed by k sequential reductions of private chunks. As such the decrease in tblock-reductions and loop iterations comes at a trade-off in extra overhead

from collective copies and per-thread reductions within individual loop iteration, but the total number of global memory reads across loop iterations remains the same, while the total number of sequential steps saved in tblock-reductions outweighs the extra steps needed to read and reduce thread-private chunks for most values of `CHUNK`.

Figure 3 shows pseudocode for the new reduce stage one loop.

```
noncomm_reduce_GPU_kernel_NEW(op, ne, xs, NUM_TBLOCKS, TBLOCK_SIZE, CHUNK) =
  /* STAGE ONE */
  Q = ceil(len(xs) / (NUM_TBLOCKS * TBLOCK_SIZE * CHUNK))
  for (k = 0; k < Q; k++) {
    /* 1a) collective copy of TBLOCK_SIZE number of CHUNK sized slices of xs
     *     into per-thread register mem.
     * 1b) per-thread sequential reductions of private chunks in register mem;
     *     TBLOCK_SIZE number of per-thread results written to shared_mem.
     * 2) tblock-reduction of shared mem and accumulation.
     */
  }
  /* STAGE TWO
   * ...
   */
```

Figure 3: High-level pseudocode for the desired changes to the reduction stage one loop GPU code generation.

3.2.1 Firing conditions

Our improvement to sequentialization will fire only for reduce kernels for which the following conditions hold:

1. the reduction is non-commutative, i.e. at least one fused operator is non-commutative. This is because our optimization seeks to decrease the number of tblock-reductions in the stage one virtualization loop, and the commutative reduce kernel already has none.

2. the reduction is a non-segmented or a large-segments segmented reduction, since the small-segments segmented reduction kernel does not follow the two-stage approach.

3. all fused operators are primitive. An operator is said to be primitive if its parameters are primitive typed and the operator is *not* vectorized. This restriction comes from the fact that static analysis of whether a given operator might fit is not possible in general for non-primitive-typed parameters, and not feasible in general for vectorized operators – and very likely most would not fit anyway. Since non-primitive operators will require the old version of the code generation and because we cannot interleave the two methods on a per-operator basis, we must require that *all* operators are simultaneously primitive.

3.2.2 `CHUNK` selection

The derivations presented and discussed in this section are in part based on similar derivations made in [3] for the segmented single-pass scan kernel.

As we have seen, the values of `TBLOCK_SIZE`and `CHUNK` determine the amount of shared memory used by the kernel, however they also affect the number of thread registers used. Since hardware specifications are fixed and `TBLOCK_SIZE`is a user-configurable kernel parameter, we must choose `CHUNK` last and compute its value depending on the other factors.

Intuitively, the optimal value of `CHUNK` might be the largest value available for the given reduction and hardware configuration – in practice this may not necessarily be the case, whicih we shall examine in section 5. In this subsection, we analyze the resources required by the kernel, and, conversely, how to bound `CHUNK` such that the kernel does not request or use too many resources.

What are "too many resources"? When a kernel requests too much shared memory, it simply crashes with a CUDA error, which we must avoid. However, when it requests too many thread registers, the compiler will neither crash nor give a warning, but rather silently spill registers to thread-private global memory [4], which obviously will be detrimental to performance [4].

Since the upper bound on shared memory is a *hard* bound, and because shared memory is directly controlled by the programmer (in our case, the code generator) it will be very easy to quantify and account for. On the other hand, the number of thread registers is not only a soft bound, but also an *elusive* bound, in the sense that because the register allocator is autonomous and the programmer is agnostic of placement, it is hard to control and profile.

Shared memory kernel requirements

As is, the reduction kernel uses shared memory for only the tblock-reductions of shared memory-held values, in which each tblock simultaneously reduces `TBLOCK_SIZE`elements from each reduction array. Since each reduction parameter p must be held simultaneously, the maximum number of bytes of shared memory required at any point by the kernel is:

$$\texttt{TBLOCK\_SIZE} \cdot \sum_{p \in P} \text{size}(p) \tag{4}$$

where size(p) is the byte element size of the parameter p.

As we introduce the additional chunking factor, the kernel will additionally require shared memory to perform collective copies from global memory to per-thread register chunks. Each collective copy will be performed separately and hence the same shared memory can be reused, meaning the amount of shared memory required by this step is determined by the maximum parameter element size:

$$\texttt{TBLOCK\_SIZE} \cdot \texttt{CHUNK} \cdot \max_{p \in P} \text{size}(p). \tag{5}$$

Hence the maximum amount of shared memory required at any point by the new reduction kernel is the maximum between eq. (4) and eq. (5):

$$\max\left(\texttt{TBLOCK\_SIZE} \cdot \sum_{p \in P} \text{size}(p), \quad \texttt{TBLOCK\_SIZE} \cdot \texttt{CHUNK} \cdot \max_{p \in P} \text{size}(p)\right). \tag{6}$$

[4] Called "local memory" in CUDA terminology.

For most cases, the maximum is likely to be decided by eq. (5), i.e. the second argument to the maximum, unless the number of fused operators is particularly high *or* we happen to have a `NUM_TBLOCKS` $>$ `TBLOCK_SIZE` $\cdot$ `CHUNK`, which is unlikely (but possible, since the user has access to these kernel parameters).

As per the CUDA programming guide [4], all modern CUDA devices[5] have *at least* 48 KiB of shared memory per tblock, while, for example, the devices we will use in testing have 163 KiB (NVIDIA A100 with CC 8.0) and 99 KiB (NVIDIA RTX 4090 with CC 8.9), respectively.

Computing the bound on `CHUNK` imposed by the amount of shared memory is straightforward. As an example, let's look at the maximum segment sum problem (MSSP) with 32-bit floats. The parallel MSS reduction operator parameter type is a quadruple of floats, which would be represented as four separate float parameters in the IR. We wish to launch our kernel with `TBLOCK_SIZE` $= 1024$ on a device with 65536 KiB of shared memory. Ignoring number of registers required, the largest available `CHUNK` is:

$$\begin{aligned} &\max\left(1024 \cdot \sum\Big\{4,4,4,4\Big\},\quad 1024 \cdot \texttt{CHUNK} \cdot \max\Big\{4,4,4,4\Big\}\right) \\ &= \max\left(16384,\ \texttt{CHUNK} \cdot 4096\right) \leq 65536. \end{aligned} \tag{7}$$

We see that the amount of shared memory required for the tblock-reduction, i.e. the 16384 bytes in eq. (7), is well within the bounds.

Hence the bound imposed on `CHUNK` by shared memory is decided by:

$$\begin{aligned} &\texttt{CHUNK} \cdot 4096 \leq 65536 \\ \iff \quad &\texttt{CHUNK} \leq 16. \end{aligned} \tag{8}$$

Thread register requirements

A number of factors and unknowns (rather, unpredictables) come into play when determining the bound on `CHUNK` imposed by the number of available registers.

For one, whereas shared memory is used *only* to hold reduction parameters, the kernel will require register memory for not only per-thread chunk arrays, but also to hold computed indices, loop bookkeeping, intermediate results, and more – for this reason, if `CHUNK` was to be pushed to the maximum value permitted under the assumption that only reduction parameters would need register allocation, then some spilling would be very likely to happen.

Secondly, it can be difficult to predict the actual number of available registers per thread or tblock for a given set of kernel parameters. As per the CUDA programming guide [4], there are multiple simultaneous restrictions on the number of registers registers per thread: One specification states a hard bound of 255 registers per thread (for all CC's ≥ 5.0), while another states a maximum of either 32K or 64K registers per tblock depending on CC, implying a maximum of 32 or 64 registers per thread for `TBLOCK_SIZE` $= 1024$, which is a conservative bound, albeit more reliable given the user-configurable `TBLOCK_SIZE`.

According to [4], the CUDA word size is 4 bytes, and 64-bit values and operations are emulated using multiple 32-bit registers and operations. Conversely, we must assume that

[5] Where "modern" means compute capability ≥ 5.0; i.e. Maxwell architecture (circa 2014) and newer.

reduction parameters with an element size smaller than 4 bytes are placed in separate 32-bit registers, even if the compiler could potentially transform some reduction operators to work on packed registers. Hence the number of registers needed for a parameter p is:

$$\text{regs}(p) = \left\lceil \max(\text{size}(p),\ 4)/4 \right\rceil. \tag{9}$$

One `CHUNK` sized register memory array is required for each reduction operator, and hence the total amount of registers needed to hold one of each parameter p of some reduction is:

$$\sum_{p \in P} \text{regs}(p) = \sum_{p \in P} \left\lceil \max(\text{size}(p),\ 4)/4 \right\rceil. \tag{10}$$

Again, please note that this ignores the additional number of registers required to hold other variables that may need to be in scope at the same time as the per-thread chunk arrays.

Returning to the previous MSS example with 32-bit float values and `TBLOCK_SIZE` $= 1024$, the number of 4-byte registers needed for one MSS quadruple is 4, meaning `CHUNK` ≤ 16 for a CC with 64 registers per thread.

Current Futhark implementation of `CHUNK` selection

Currently, the same computation is used to choose `CHUNK` for the reduce and single-pass scan kernels, and it is based on the work in [3]. It simply chooses the largest possible `CHUNK` that satisfies the bound on shared memory as well as an estimate of the number of available registers per thread. This estimate is based on the number of registers required to hold one parameter from each fused operator at once but also adds a small overhead of three registers on top; two to hold index computations and one to hold a loop variable.

Design and implementation of a new method for the choice of `CHUNK` is outside the scope of this project, but in section 6.1 we discuss how and why this might be beneficial.

4 Validation testing

In this section, we present our validation testing plan and results for both of the modifications as described in section 3.

First and foremost, we will use the Futhark CI test suite for validation testing, since we assume this to be sufficiently exhaustive in terms of testing different operators.

Secondarily, we want to assert that our implementations do not break or produce incorrect results under varying parameterizations, including different values of `TBLOCK_SIZE` for each kernel, as well as different values of `NUM_TBLOCKS` for the scan kernel, and different values of `CHUNK` for the new reduce kernel.

4.1 Validation testing plan

This section details the different sets of parameters used in the additional testing outside of the Futhark CI test suite. We wish to limit the number of test cases as much as we can, while still providing reliable and meaningful results.

Compiler backend and hardware used

We exclusively test code generated by the CUDA backend of the compiler, and validation tests are run on the A100 only.

Validation of the single-pass scan kernel are performed on commit `f7a36ee` [5] of the compiler, i.e. the PR merge commit for our changes to the single-pass scan kernel, while validation of the changes to the reduce kernel are made on commit `5234eb8` [6], i.e. the PR merge commit for our changes to the reduce kernel.

Test programs

For the purposes of the additional validation testing, we choose the two programs `lssp.fut` and `mm-5x5.fut` from among our benchmarking test programs for both the scan and reduce kernels. The first program is interesting because it maps a single global memory input array to an array of six-tuples in shared memory, whereas the second is interesting because its reduction operator has 25 global memory input arrays (the largest among our benchmarking programs). Both of these programs will be further introduced in section 5, but for validation purposes, we additionally write a segmented version of the program to also validate test the segmented code generation.

To view the source code for the test programs, see the benchmarking programs in appendix A and the additional validation testing entry points and validation test cases in appendix B.

Test input sizes

Each validation test program is tested using the same input sets as will be used in the A100 benchmarks, i.e. a small and a large input set of 3 and 30 GBs of randomly generated data.

`TBLOCK_SIZE` values

For both implementations, we want to test various values of `TBLOCK_SIZE`, including values that are small, large, non-powers of 2, and 1024 (the maximum value for CUDA devices). The scan kernel requires that `TBLOCK_SIZE` is a multiple of 3n while the reduce kernel does *not*, so we shall additionally test the reduce kernel with some odd `TBLOCK_SIZE`s.

Hence the values used for both kernels are of $\texttt{TBLOCK\_SIZE} \in \{32, 448, 1024\}$, while the reduce kernel is additionally tested with $\texttt{TBLOCK\_SIZE} \in \{31, 761\}$.

`NUM_TBLOCKS` values (single-pass scan only)

We wish to choose a set values of `NUM_TBLOCKS` s.t. we will have test cases where: one tblock performs all the work; many tblocks perform a large number of virtualization loop iterations each, and the number of iterations vary between tblocks, for both a small and large number of `NUM_TBLOCKS`; the number of requested tblocks is equal to the number of required tblocks (i.e. each tblock performs exactly 1 virtualization loop iteration).

Recall from section 3.1 that the host code will automatically choose the minimum between the required and the requested number of tblocks – hence we can force the latter test case by requesting a sufficiently high `NUM_TBLOCKS`. Note that this also implies that we cannot test cases where one or more tblocks perform no work.

Hence the values used are $\texttt{NUM\_TBLOCKS} \in \{1, 31, 1024, 2^{31} - 1\}$.

`CHUNK` values (reduce only)

At present, `CHUNK` is not a configurable or tunable kernel parameter; rather, the compiler simply chooses the largest possible value. Despite this, we see it fit to assert that the kernel does not fail for smaller `CHUNK` values, in case we want to make the parameter tunable in the future *or* in the case that benchmarking shows the optimal `CHUNK` value to not be the largest in sufficiently many cases.

We already know for certain that a given reduce kernel will break if `CHUNK` is set too high, due to requesting too much shared memory. Hence it is expected that some test cases will fail with a corresponding error message if we attempt to use the same set of `CHUNK` values for all test programs and values of `TBLOCK_SIZE`. What is interesting here is whether those cases which are fit to launch produce correct results – it is then up to the compiler to choose only between values of `CHUNK` that fit.

When we eventually get to benchmark testing, we will try all `CHUNK` values up to and including 40. However, for the purposes of validation testing, we limit ourselves to the: set of values $\texttt{CHUNK} \in \{1, 9, 24, 40\}$, which we argue is representative.

4.2 Validation testing results

All validation tests in the Futhark CI test suite run successfully for both the reduce and scan pull requests. All of the additional validation test cases run successfully as well.

4.3 Reproduction of validation results

To reproduce benchmarking results, please first find the source code for our benchmarking test programs in appendix A – the validation entry points and test cases can then be found in appendix B.

All validation tests of the reduce kernel changes were made on commit `5234eb8` [6] of the Futhark compiler, while the single pass-scan kernel changes were tested on commit `f7a36ee` [5], so these Futhark compiler versions should be installed.

To reproduce the different validation test cases, please refer to the validation plan in section 4.1.

5 Benchmarking

In this section, we present and discuss our benchmarking plan for the new reduce kernel, and later, we present, analyze, and conclude upon the results of benchmarking.

Using a small benchmarking suite of six reduction operators, we wish to examine the performance of our implementation, as well as to explore optimal `CHUNK` parameterization.

We first and foremost wish to examine if there are significant speedups to be had over the old implementation, and for those cases where we see speeddown, if any, we wish to explore why. Ideally we do not only obtain speedups over the reference implementation, but also come close to peak bandwidth for the different devices we test.

Secondarily, we want to explore different configurations of `CHUNK` and `TBLOCK_SIZE`, to determine optimal parameters but also to explore sensitivity in parameterization of the new reduce kernel.

Our benchmarking is divided into two rounds: In the first we explore kernel parameterization, and in the second we compare with the reference implementation.

5.1 Benchmarking plan

Again, we wish to limit the number of test cases as much as possible while still providing reliable and meaningful measurements for analysis. In this section we present some of the different factors in the benchmarking suite.

Compiler versions and hardware used

To benchmark our reduce optimization, we use commit `5234eb8` [6] of the compiler, i.e. the merge commit for the PR containing our changes. For the reference measurements, we use the immediately preceding commit, `06732a5` [7].

We exclusively test code generated by the CUDA backends of the compiler, and benchmarking tests are run on both an RTX4090 (CC 8.9; 24 GB device mem) and an A100 (CC 8.0; 40 GB device mem).

Note that measurements are made using `futhark-bench` and hence exclude certain things such as device/host memory copies.

Test programs

We examine six different reduction operators. The first three operators are matrix multiplication operators over 2×2, 3×3, and 5×5 matrices, respectively, i.e. reductions with 4, 9, and 25 global memory input arrays and progressively larger operators. These are interesting to benchmark because they can be used to examine the effect on performance of higher requirements for shared memory and register usage.

The next two are maximum segment sum (MSS) and longest satisfying subsequence (LSS) operators. These each map a single global memory input array to arrays of four- and six-tuples, respectively, in shared memory, and these are interesting to test because shared memory may be a potential bottleneck.

The last program is linear function composition, which takes as input two length n arrays representing n linear functions and computes the composition of these. This program is interesting because it is the most lightweight binary operator we can think of that is associative and non-commutative, but also non-trivial.

Test input sizes

For all combinations of test program and device, we use a large and a small test input, where the "large" test input is roughly 75% of the given device's memory and the "small" test is roughly a tenth of the large set. Hence for the RTX4090, the large and small sets are $\sim$18 GB and $\sim$1.8 GB, respectively, while for the A100, the large and small sets are $\sim$30 GB and $\sim$3 GB, respectively.

`TBLOCK_SIZE` values

All test cases are run with varying values of `TBLOCK_SIZE` to explore its effect on performance. For the reduce kernel changes, this is especially interesting to explore in relation to the `CHUNK` factor, since the two hyperparameters together determine the amount of thread registers (and shared memory) required, and because some combinations might cause register spilling.

We wish to test commonly used values, hence we choose all powers of 2 between 32 and 1024, i.e. `TBLOCK_SIZE` $\in \{32,\ 64,\ 128,\ 256,\ 512,\ 1024\}$.

`CHUNK` values (reduce only)

We want to find the optimal `CHUNK` parameterizations for various operators and `TBLOCK_SIZE`s. In particular, we are interested in examining whether there is a predictable pattern in the optimal `CHUNK` values for different `TBLOCK_SIZE`s. Fortunately, the range of possible `CHUNK` values is quite limited due to the constraint on shared memory (and, secondarily, thread registers), and hence we choose simply to test all values in the range `CHUNK` $\in \{1, \ldots, 40\}$.

Test rounds and performance metrics

Our benchmarking plan is split in two rounds: In the first, we explore the performance obtained for various parameters, to find optimal parameterization and to expose pitfalls and patterns; finally, we compare the best measurements for each `TBLOCK_SIZE` with the reference implementation.

In each round, we measure performance in terms of throughput in GB/sec, i.e. the number of bytes of global memory input/output processed by the kernel divided by the time taken to do so. This is an interesting metric as we are not only interested in comparing our implementation with existing code generation, but also to compare with the peak bandwidth of the given devices. Speedups over the reference implementation will be considered relative to the percentage of peak bandwidth obtained by both implementations – if, say, we obtain a 3x speedup for a certain test case, but reach only 60% peak bandwidth, then this should be taken as indication that there is still room for improvement.

5.2 Benchmarking round 1 results analysis

For all test cases in the first round of testing, the measurements made over the small input sets exhibit largely the same patterns as those of the large input sets. To limit the amount of figures in the report, we omit plots for the small input set test cases, but these can be found in appendix C.

In each results plot, the best `CHUNK` for each `TBLOCK_SIZE` is highlighted with a green box in the heatmap.

We first present all results from the A100 benchmarks and then the RTX4090 benchmarks.

5.2.1 A100 round 1 results

A100 round 1 results: `mm-2x2.fut`

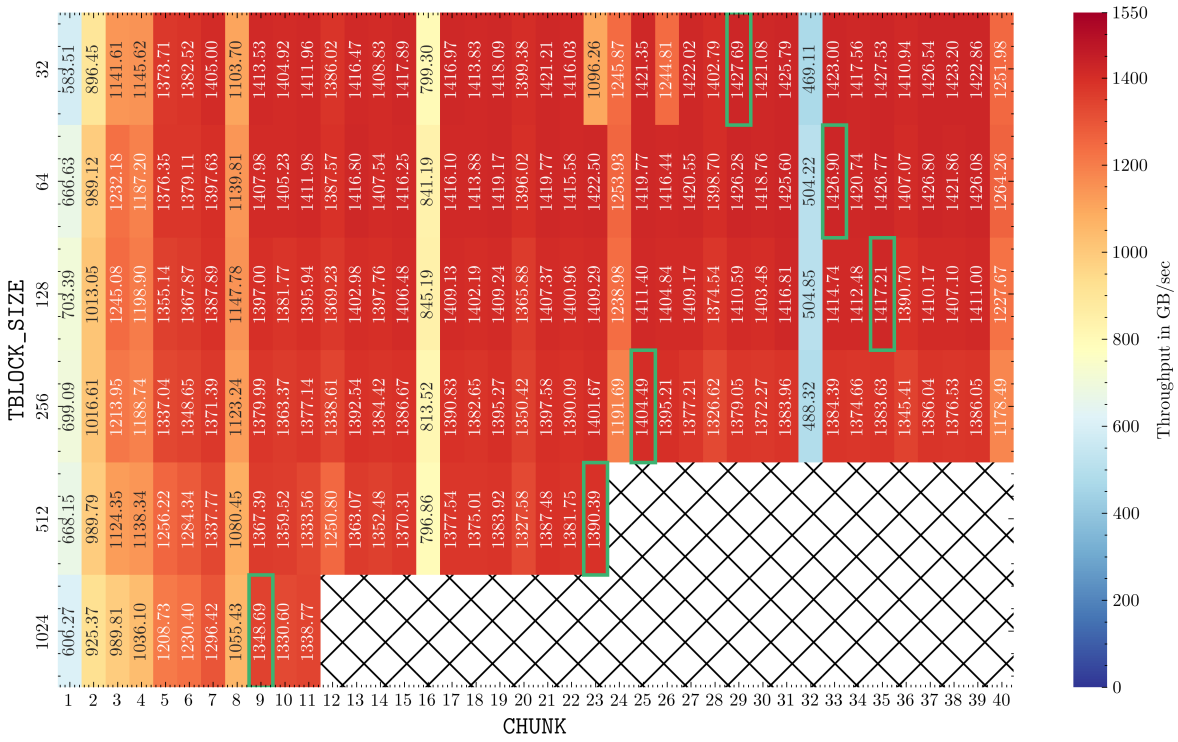

Figure 4: A100, `mm-2x2.fut`, large dataset (∼30 GB)

The A100 obtains high throughput for all six `TBLOCK_SIZE`s, reaching between 87% and 92.1% of peak bandwidth for `TBLOCK_SIZE` = 1024 and 32, respectively.

The best `CHUNK` values appear to be in the higher ends of the the available `CHUNK` ranges for each `TBLOCK_SIZE`.

There appear to be significant dips in performance for all `CHUNK`s divisible by 8 – most notably 16 and 32, but also to some degree 8, 24, and 40. While not nearly as significant, there also appear to be very slight dips in performance for all other multiples of 4 greater than or equal to 12. For all other `CHUNK`s greater than 8, the performance seems to be stable and reliable.

Note that the kernel failed to launch due to shared memory constraints for `TBLOCK_SIZE` = 512 $\wedge$ `CHUNK` $\geq$ 24, and for `TBLOCK_SIZE` = 1024 $\wedge$ `CHUNK` $\geq$ 12, however judging from the measurements alone, it seems that no significant register spilling occurred for any of the `TBLOCK_SIZE`s.

A100 round 1 results: `mm-3x3.fut`

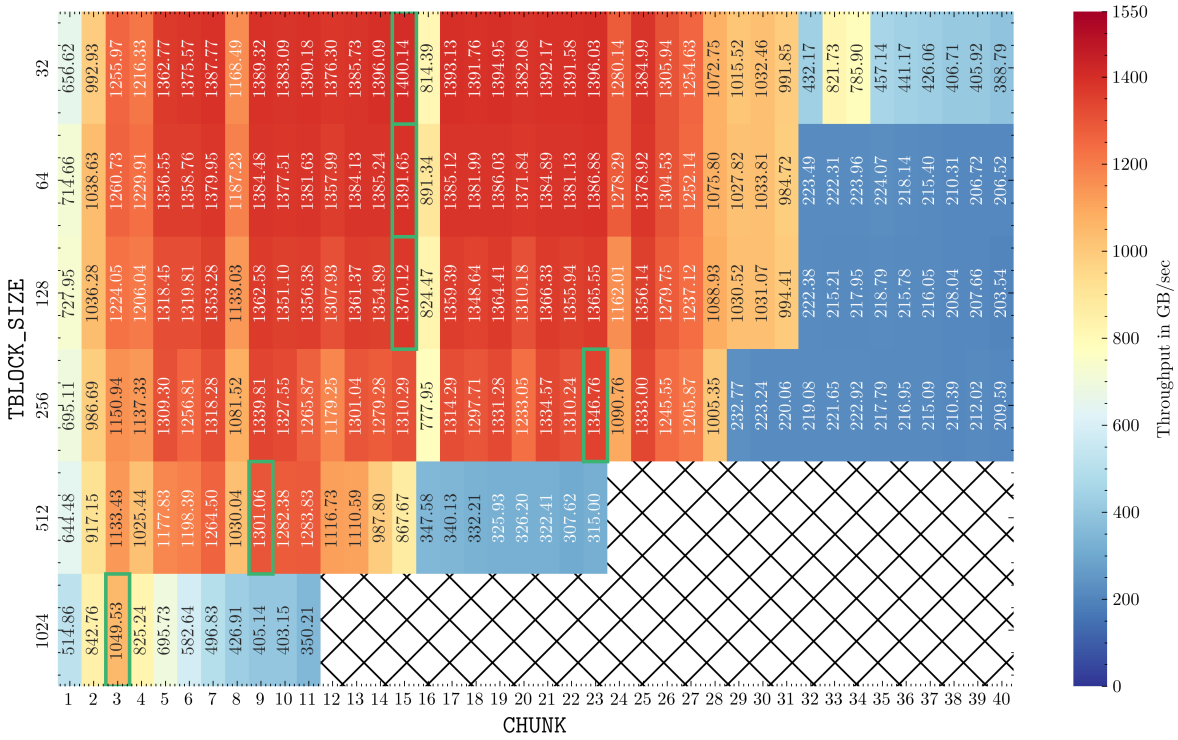

Figure 5: A100, `mm-3x3.fut`, large dataset (∼30 GB)

Again, A100 measurements are respectable, reaching between 83.9% to 90.3% of peak bandwidth for `TBLOCK_SIZE` ≤ 512, however only 67.7% for `TBLOCK_SIZE` $= 1024$.

However, again we see large variance in measurements across `CHUNK`s for each `TBLOCK_SIZE`s. To some degree, the pattern of performance dips for multiple-of-8 `CHUNK`s also appears here, in particular for `CHUNK` $= 16$ with the smallest four `TBLOCK_SIZE`s.

Interestingly, the heatmap now reveals a good and clear sign of register spilling for all `TBLOCK_SIZE`s, indicated by the sudden drop-off in performance (i.e. the heatmap suddenly turning blue going from one `CHUNK` to the next), for e.g. `TBLOCK_SIZE` $\leq 128 \wedge$ `CHUNK` ≥ 32.

A100 round 1 results: `mm-5x5.fut`

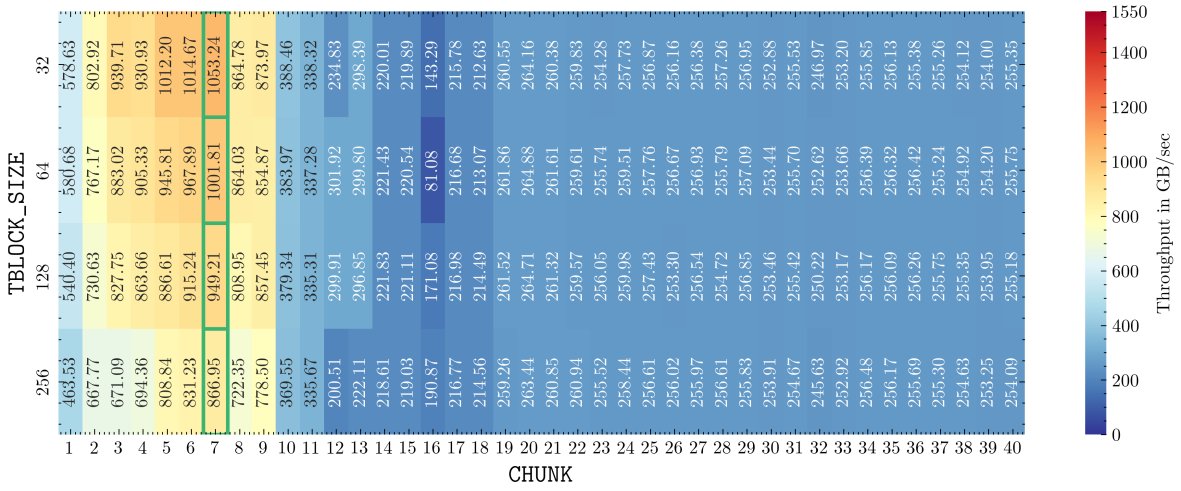

Figure 6: A100, `mm-5x5.fut`, large dataset (∼30 GB)

Notice that the kernel failed to launch entirely due to shared memory constraints for `TBLOCK_SIZE` ≥ 512 and all values of `CHUNK`.

The plot shows poor performance for all parameterizations relative to the previous two programs, with peak throughputs of 55.8% to 67.9% of peak bandwidth for `TBLOCK_SIZE` $= 256$ and 32, respectively.

On the other hand, measurements are more consistent here across all `TBLOCK_SIZE`s, until the relatively early dropoff at `CHUNK` ≥ 10, which is very likely due to register spilling.

A100 round 1 results: `linear_function_composition.fut`

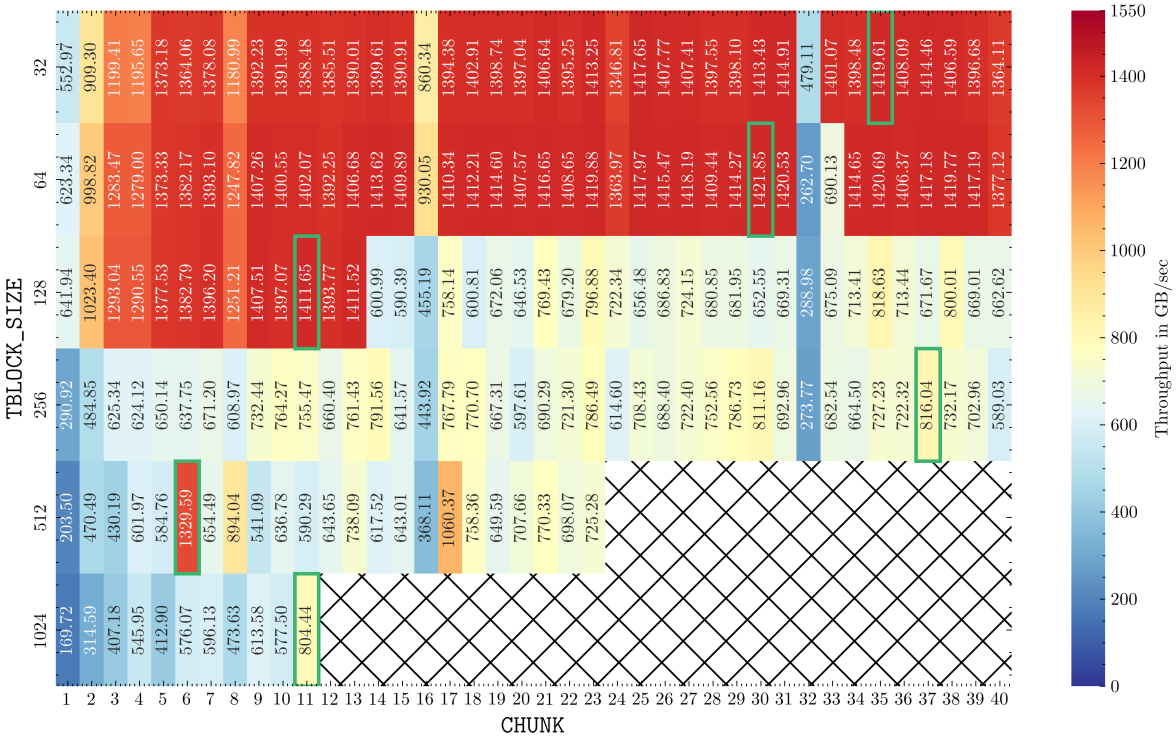

Figure 7: A100, `linear_function_composition.fut`, large dataset (∼30 GB)

Measurements show good performance for `TBLOCK_SIZE` ≤ 64 and most `CHUNK` values, except for the first 4 and, again, multiples of 8.

For `TBLOCK_SIZE` $= 128$, performance quickly drops at `CHUNK` ≥ 14, which is interesting. For all larger `TBLOCK_SIZE`s, performance is lackluster for almost all `CHUNK`s, except for a curiously high measurement for (`TBLOCK_SIZE`, `CHUNK`) $= (512, 6)$ as compared to other measurements for this `TBLOCK_SIZE`. However, the small input set results show something similar (see appendix C).

Usually we would relate the dropoffs to register spilling, however the linear function composition operator is the smallest in our test suite, and we did not see the same pattern for the 2×2 MM operator (the second smallest), so unfortunately we cannot be sure.

A100 round 1 results: `mssp.fut`

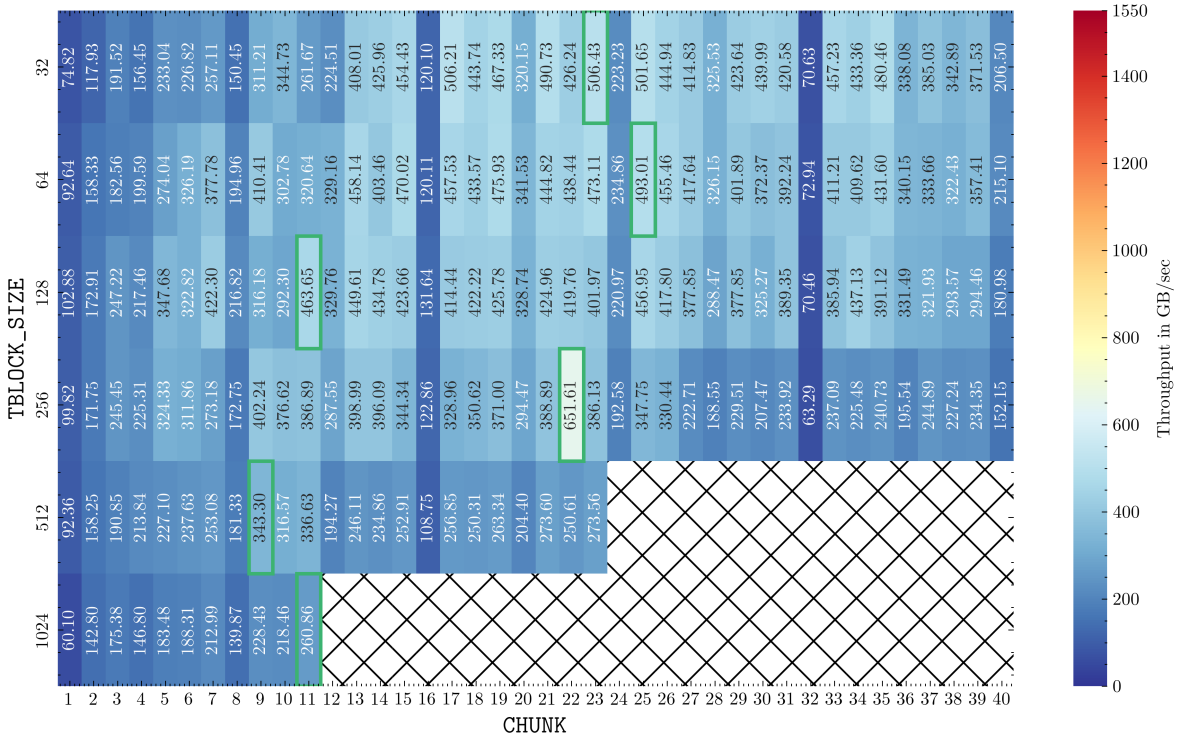

Figure 8: A100, `mssp.fut`, large dataset (∼30 GB)

As we go into the `mssp.fut` and `lssp.fut` results analyses, recall that the `mss` and `lss` operators work on 4- and 6-tuples, respectively. This means that as elements are loaded from the single global memory input array, they are mapped to tuples at the head of the stage one loop. This essentially means that `CHUNK` and the sequentialization factor each take a significant hit even though the input is small (in that there is only one input array of 32-bit values), and because throughput is measured in terms of the input size in bytes, this puts a strain on the kernel to compensate.

Still, the measurements for the A100 are disappointingly low, reaching only as high as 42% for $(\texttt{TBLOCK\_SIZE}, \texttt{CHUNK}) = (256, 22)$, and lower still for the other `TBLOCK_SIZE`s. As has been the case for virtually all A100 measurements, there are dips in performance for multiple-of-8 `CHUNK`s.

Overall, the heatmap indicates that performance is best for smaller `TBLOCK_SIZE`s and `CHUNK`s roughly in the range $13 \ldots 35$ (save for multiples of 8).

We do *not* think that the low performance can be attributed to register spilling, since the operator is still rather small (as small as the `mm-2x2` operator), so our best guess is that the performance hit is from the low sequentialization factor relative to the number of input elements.

A100 round 1 results: `lssp.fut`

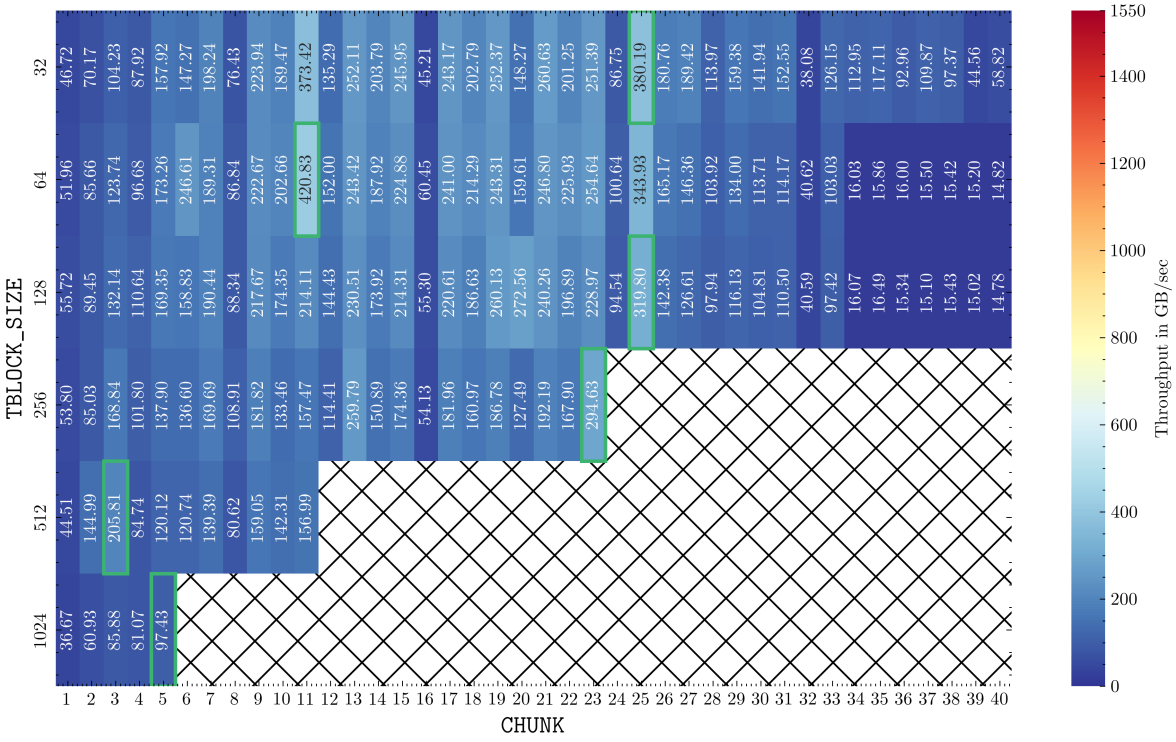

Figure 9: A100, `lssp.fut`, large dataset (∼30 GB)

The A100 `lssp.fut` plot largely resembles that of `mssp.fut` (**??**), except measurements are even lower here, reaching as low as 6.3% of peak bandwidth for `TBLOCK_SIZE` = 1024 and peaking at 27.1% for `TBLOCK_SIZE` = 64. In addition, for all but `TBLOCK_SIZE` = 32s, there seems to be a significant gap between the best and second best `CHUNK`s.

Even though speed is low across the board, multiple-of-8 `CHUNK`s show even poorer measurements, with speeds below 100 GB/sec for all but one measurement ((`TBLOCK_SIZE`, `CHUNK`) = (256, 8)), and for the smallest three `TBLOCK_SIZE`s, we see a total drop-off in performance for the largest `CHUNK`s, with speeds dropping as low as ∼15 GB/sec.

5.2.2 RTX4090 round 1 results

RTX4090 round 1 results: `mm-2x2.fut`

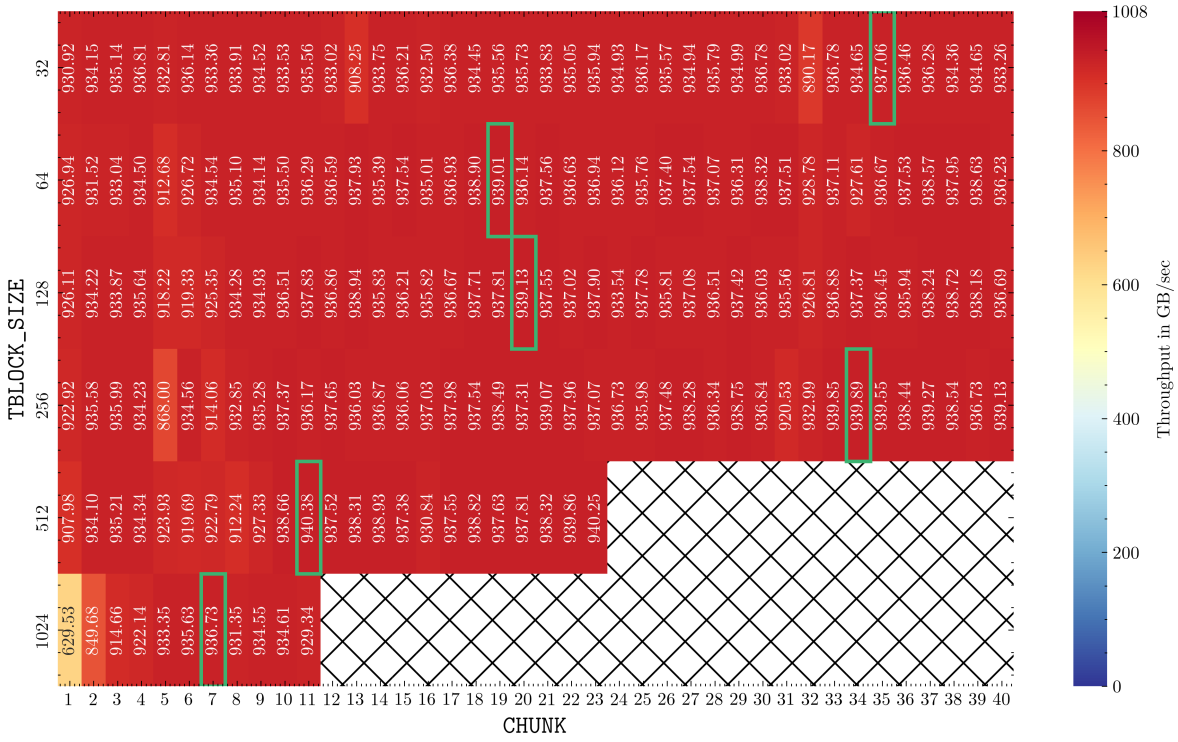

Figure 10: RTX4090, `mm-2x2.fut`, large dataset (∼18 GB)

Immediately there is significantly less spread in the RTX4090 measurements than in those of the A100, with much less variance in the heatmap, indicating better consistency across `CHUNK`s and a more reliable kernel. Second, the highest measured throughput per `TBLOCK_SIZE` all fall between 92.9% to 93.3% for `TBLOCK_SIZE` = 1024 and 512, respectively. Finally, the RTX4090 does not seem to exhibit the dips in performance for multiple-of-8 `CHUNK`s.

Again the kernel fails due to shared memory constraints for `TBLOCK_SIZE` = 512 ∧ `CHUNK` ≥ 24, and for `TBLOCK_SIZE` = 1024 ∧ `CHUNK` ≥ 12, but there seems to have been no noticable register spilling for any of the `TBLOCK_SIZE`s.

RTX4090 round 1 results: `mm-3x3.fut`

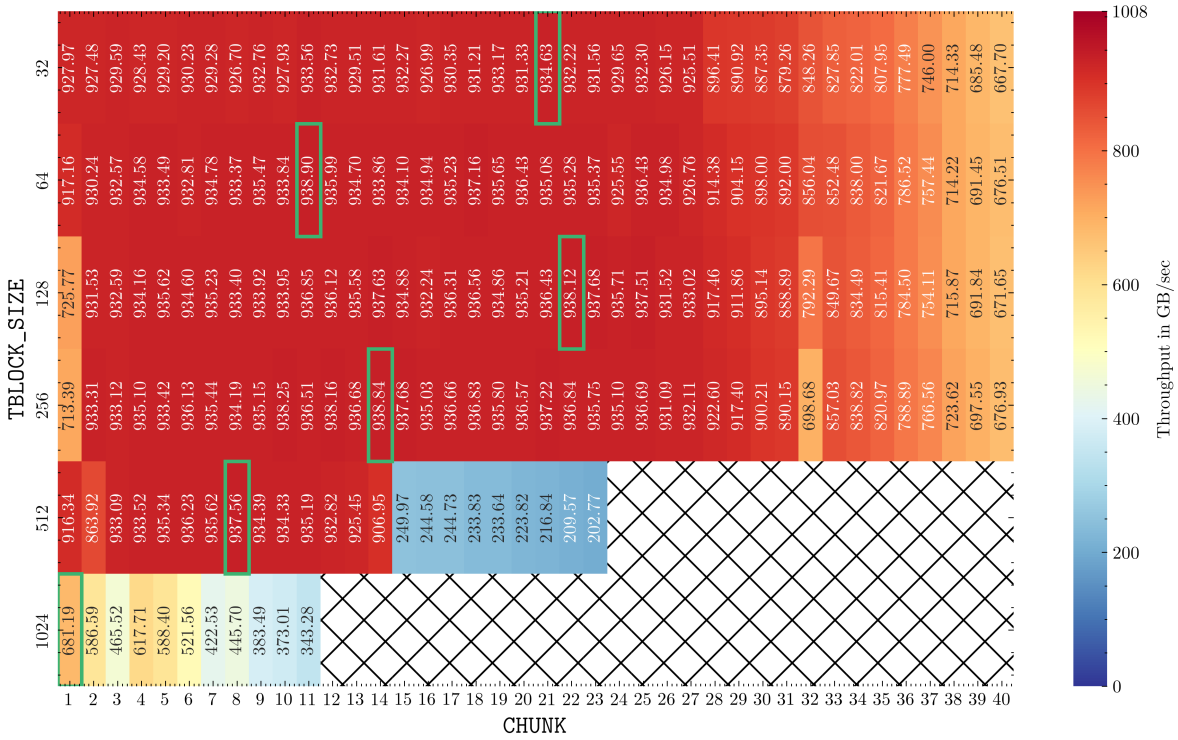

Figure 11: RTX4090, `mm-3x3.fut`, large dataset (∼18 GB)

The RTX4090 measurements show good speeds between 92.7% to 93.1% of peak bandwidth for `TBLOCK_SIZE` ≤ 512, but only 67.5% for `TBLOCK_SIZE` $= 1024$. This is interesting because it largely matches the A100 results.

The heatmap indicates that the RTX4090 measurements are again significantly more consistent and reliable across `CHUNK`s than the A100 measurements, except for `TBLOCK_SIZE` $=$ 1024, which for some reason performs poorly for all `CHUNK`s.

This time, however, we begin to see register spilling for `TBLOCK_SIZE` ≤ 256 as `CHUNK` grows large, indicated by the heatmap slowly turning orange and yellow. For the largest two `TBLOCK_SIZE`s, this is more sudden and noticable.

RTX4090 round 1 results: `mm-5x5.fut`

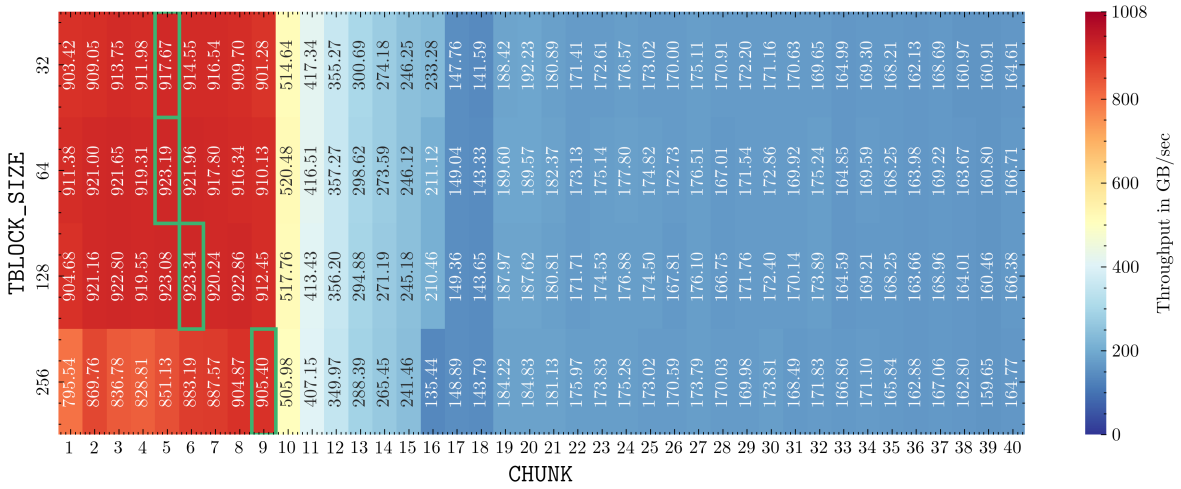

Figure 12: RTX4090, `mm-5x5.fut`, large dataset (∼18 GB)

Again, the kernel failed due to shared memory constraints for `TBLOCK_SIZE` ≥ 512 and all values of `CHUNK`.

Performance is significantly better on the RTX4090, with speeds between 89.8% and 91.6% for `TBLOCK_SIZE` $= 256$ and 128, respectively. This time, the raw throughput measurements even come close to those of the A100, which is respectable.

Interestingly, the drop-off in performance (likely due to register spilling) comes at the exact same point, i.e. `CHUNK` $= 10$, for each `TBLOCK_SIZE` – however, the performance seems quite consistent across `CHUNK`s until this cutoff point.

RTX4090 round 1 results: `linear_function_composition.fut`

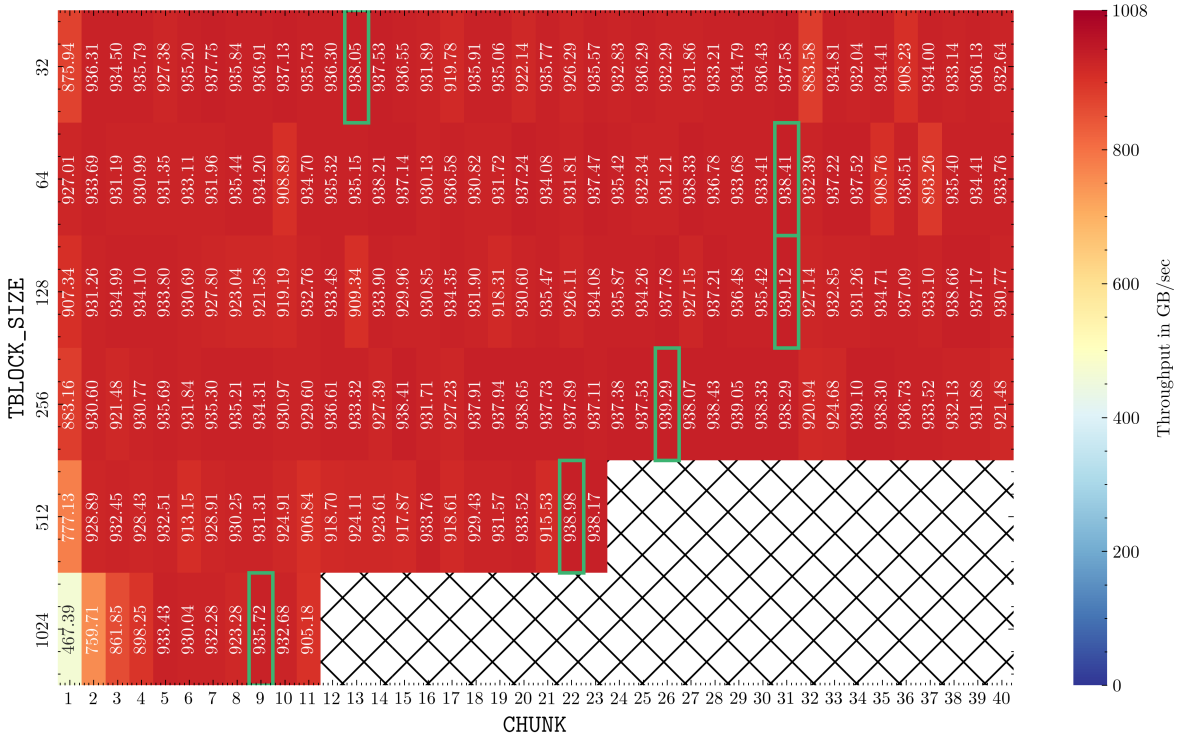

Figure 13: RTX4090, `linear_function_composition.fut`, large dataset (∼18 GB)

The RTX4090 is very consistent on the linear function composition reduction with best `CHUNK`s across `TBLOCK_SIZE`s reaching speeds between 92.8% and 93.1% of peak bandwidth.

In fact, this plot resembles almost exactly the RTX4090 results for `mm-2x2.fut` (see fig. 10) – this was expected, since the operators are similar in size, but this makes the A100 results (fig. 7) all the more curious.

RTX4090 round 1 results: `mssp.fut`

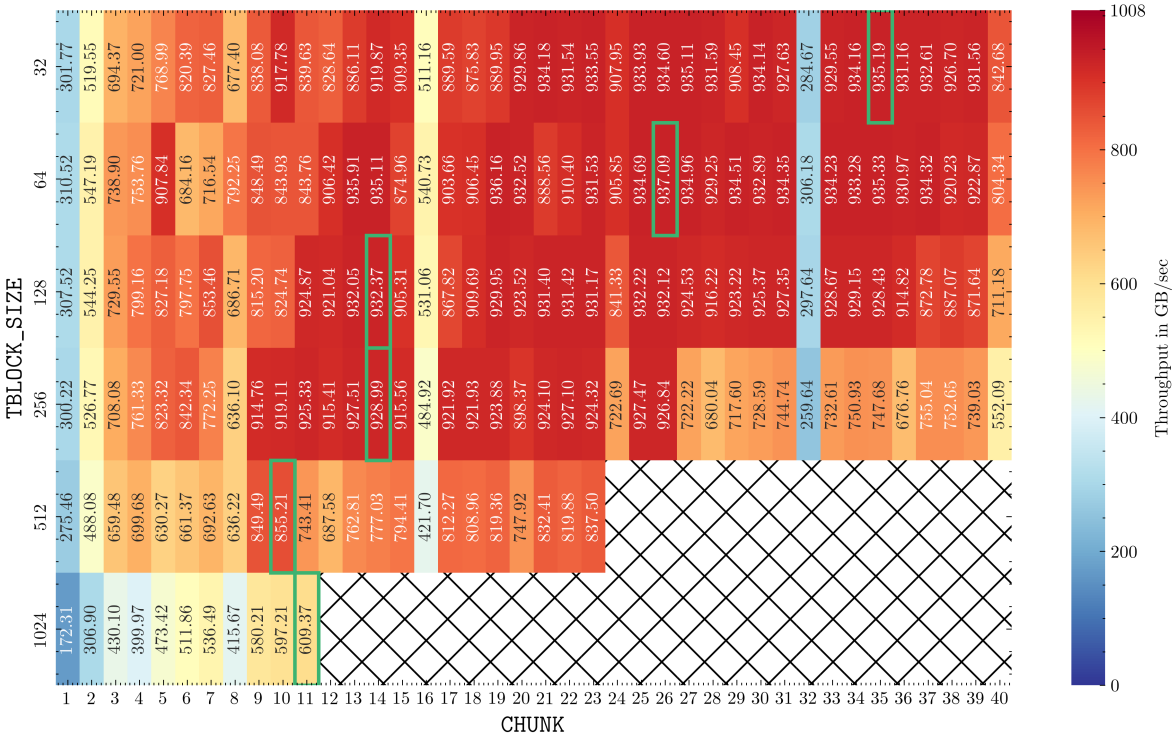

Figure 14: RTX4090, `mssp.fut`, large dataset (∼18 GB)

Once again, the RTX4090 performs exceptionally well relative to the A100. While the least consistent of the RTX4090 plots thus far, it obtains remarkable results for `TBLOCK_SIZE` ≤ 256, with speeds up to 92.9% of peak bandwidth. These are also the `TBLOCK_SIZE`s for which the kernel is most consistent in parameterization, except for the drop-off in performance for `TBLOCK_SIZE` $= 256 \wedge$ `CHUNK` ≥ 27, perhaps due to register spilling.

Interestingly, this is the first RTX4090 result for which we see the multiple-of-8 `CHUNK` pattern as has been evident in virtually all A100 measurements thus far.

RTX4090 round 1 results: `lssp.fut`

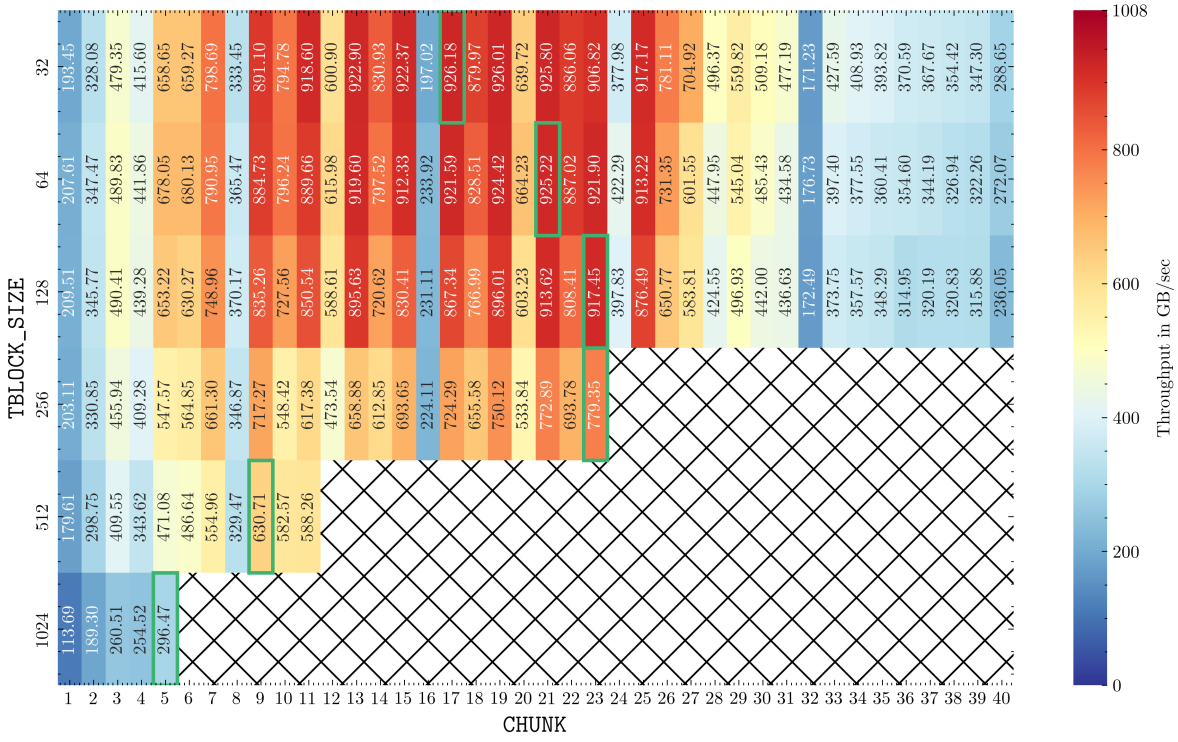

Figure 15: RTX4090, `lssp.fut`, large dataset (∼18 GB)

The RTX4090 `lssp.fut` plot shows the most unreliable results across the RTX4090 results. Despite reaching excellent speeds of up to ∼ 91.5% for the smallest three `TBLOCK_SIZE`s, the heatmap reveals that the kernel is highly unstable in `CHUNK` parameterization for all `TBLOCK_SIZE`s – with the good `CHUNK` values being few and far between – and that the kernel reaches at best acceptable speeds for the three largest `TBLOCK_SIZE`s.

As with the RTX4090 `mssp.fut` measurements (fig. 14), we see performance degradation for `CHUNK` multiples of 8 and, to a lesser degree, multiples of 4.

5.3 Benchmarking round 2 results analysis

In the second round of benchmarking, we compare the best results from round 1 for each `TBLOCK_SIZE` against measurements made using the reference implementation. Here we show measurements for both the small and large input sets, since it is more convenient this time given the smaller number of measurements to discuss.

For each measurement for the new implementation, the speedup and the associated `CHUNK` value is listed in parentheses.

We first present all results from the A100 benchmarks and then the RTX4090 benchmarks.

A100 round 2 results: `mm-2x2.fut`

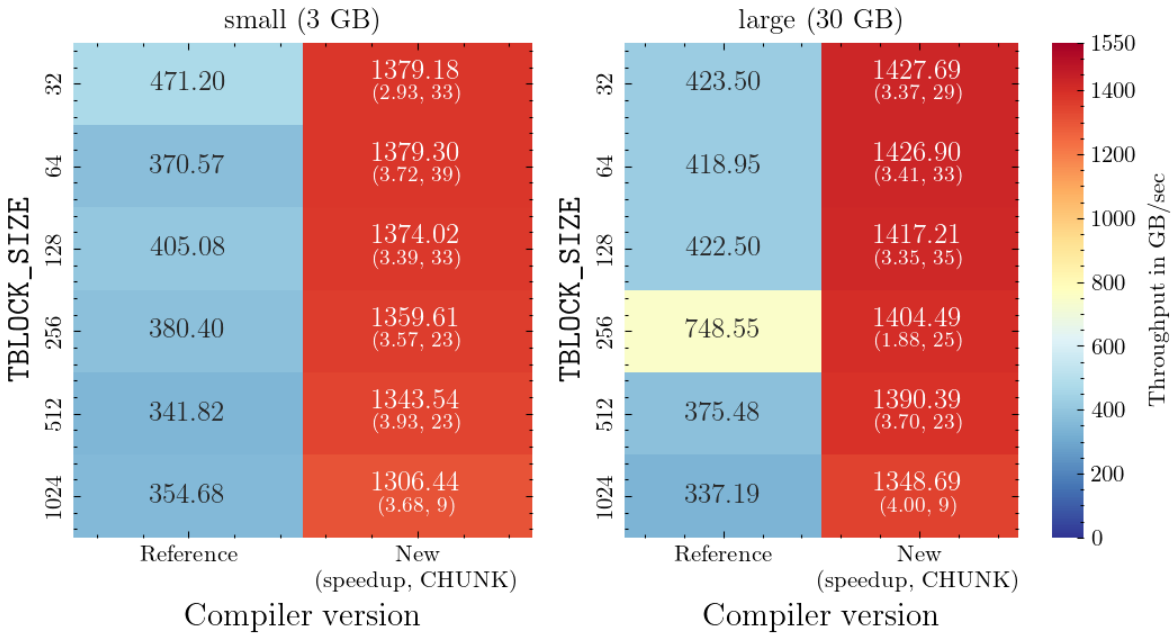

Figure 16: A100, `mm-2x2.fut`, comparison with reference.

We see excellent speedups over the reference implementation for all `TBLOCK_SIZE`s, however since throughput was relatively low for the reference implementation, the speedup factors are in themselves not necessarily impressive.

While the new implementation obtains better speeds for lower `TBLOCK_SIZE`s, the speedups appear to grow with `TBLOCK_SIZE`.

A100 round 2 results: `mm-3x3.fut`

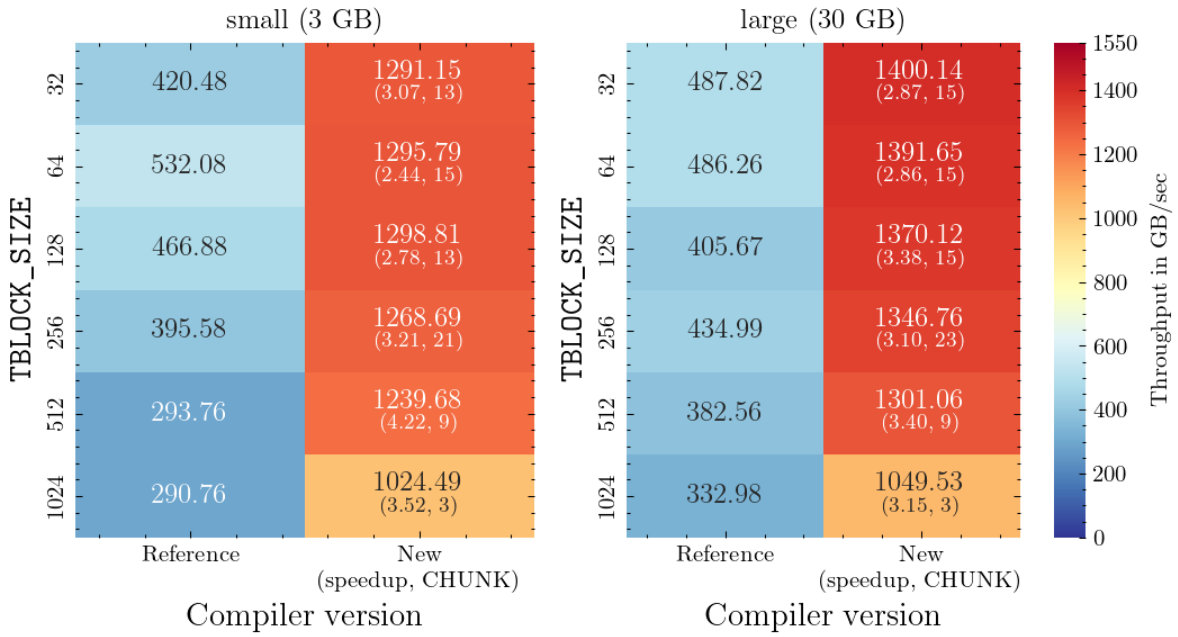

Figure 17: A100, `mm-3x3.fut`, comparison with reference.

This plot shows largely the same patterns in speedup as for the A100 `mm-2x2.fut` comparison plot (fig. 16), but with slightly lower throughputs measured on the new implementation.

A100 round 2 results: `mm-5x5.fut`

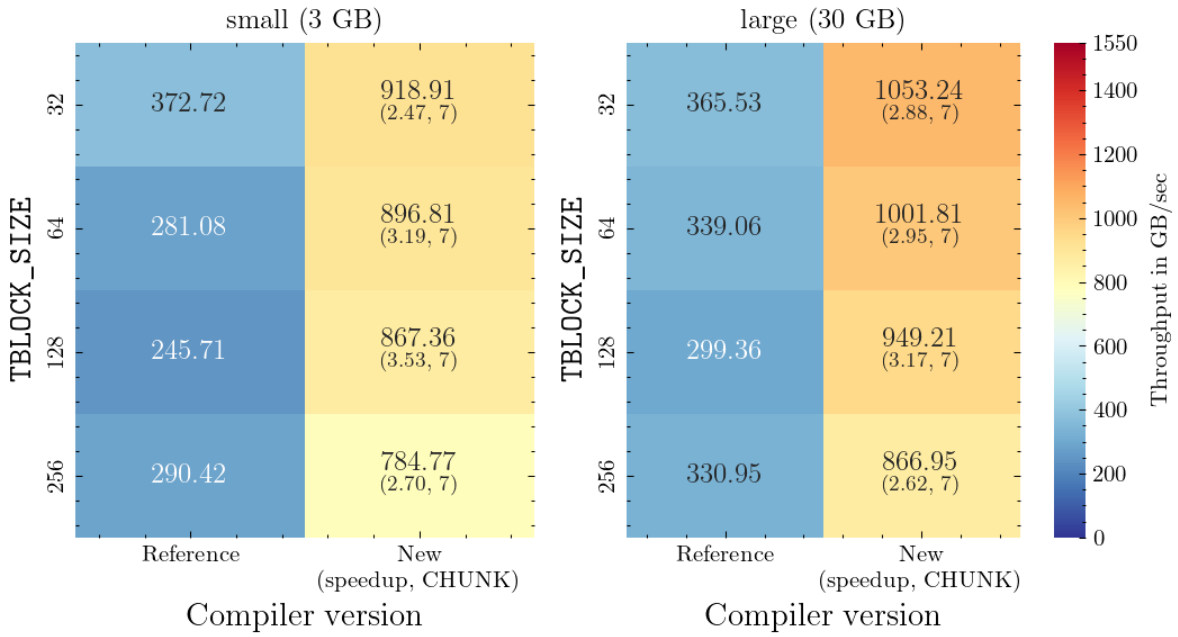

Figure 18: A100, `mm-5x5.fut`, comparison with reference.

We see significant speedups over the reference implementation, however the reference implementation did not perform particularly well w.r.t. peak bandwidth, hence there was much room for improvement. Still, the new implementation did not come close to peak bandwidth, so on the other hand, there is room for even bigger speedups.

Note that both implementations failed to launch the kernel for `TBLOCK_SIZE` ≥ 512.

A100 round 2 results: `linear_function_composition.fut`

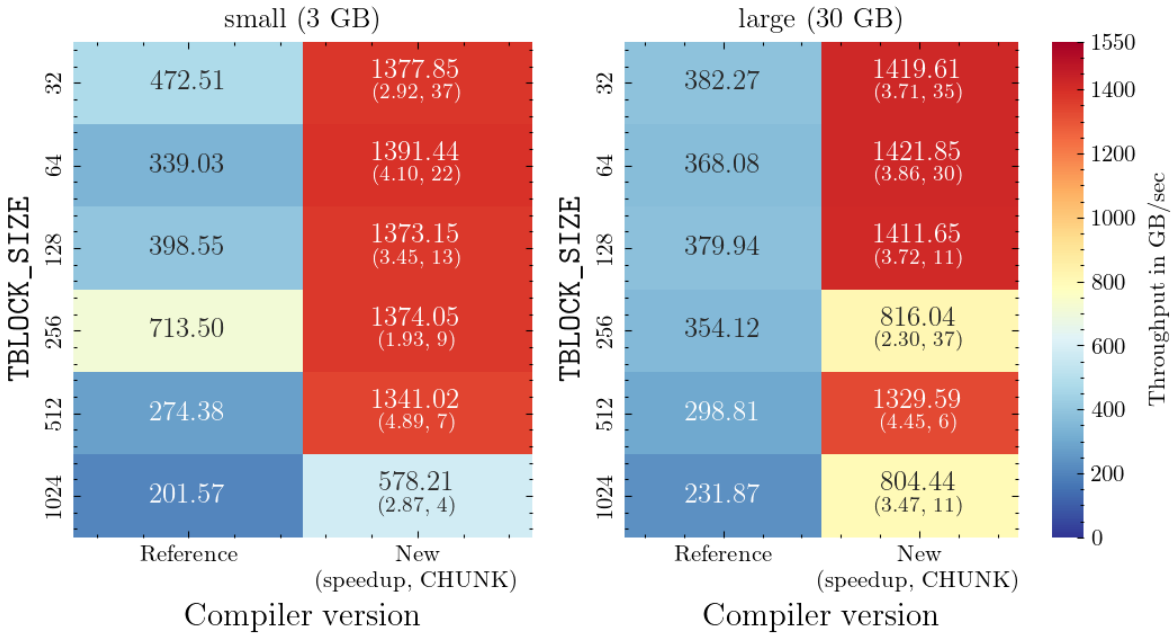

Figure 19: A100, `linear_function_composition.fut`, comparison with reference.

These results resemble the A100 `mm-2x2.fut` measurements. The reference implementation did not perform particularly well, but speedups are still interesting since we obtain throughput in the upper ranges for most `TBLOCK_SIZE`s (with a curious dip for `TBLOCK_SIZE` = 256 on the large set).

A100 round 2 results: `mssp.fut`

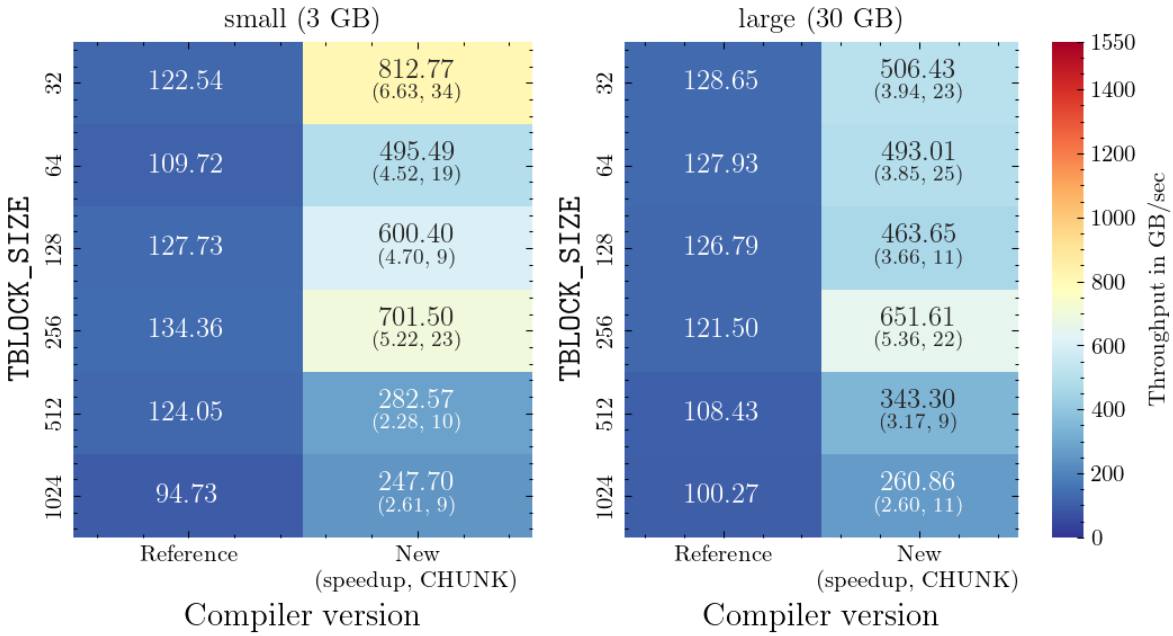

Figure 20: A100, `mssp.fut`, comparison with reference.

Interestingly, the `mssp.fut` tests are where we see some of the largest speedups on the A100, despite the relatively poor throughput obtained for all `TBLOCK_SIZE`s. This, of course, should be attributed to the low performance of the reference more so than the new implementation.

A100 round 2 results: `lssp.fut`

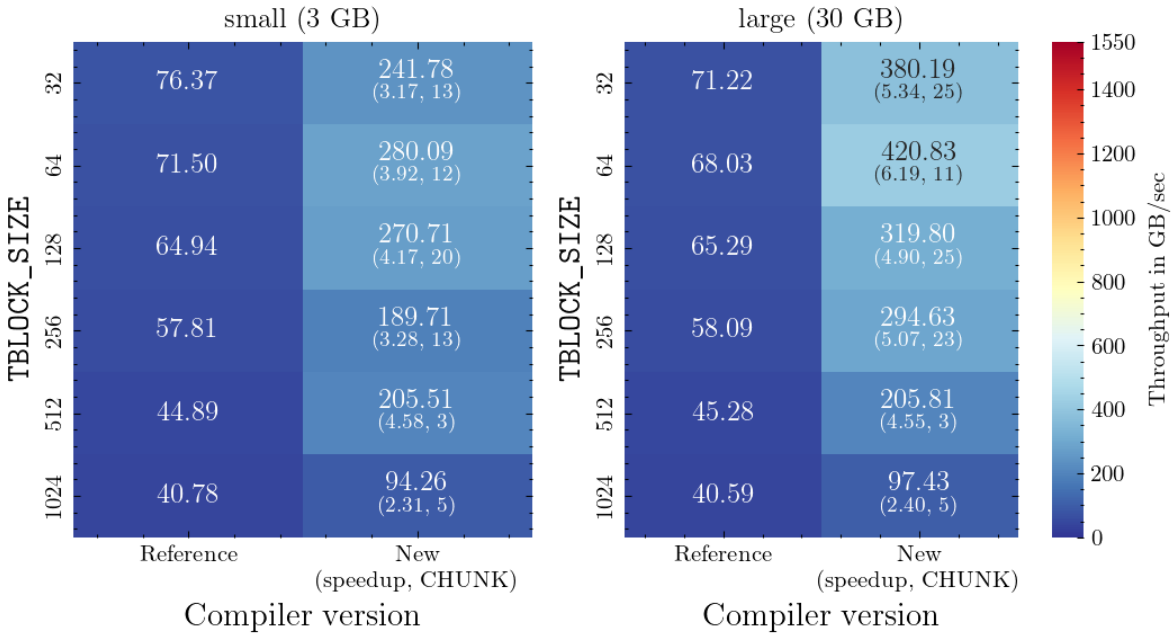

Figure 21: A100, `lssp.fut`, comparison with reference.

We see the same pattern as for the A100 `mssp.fut` comparison plot (fig. 20): Very big speedups due to remarkably low throughput from the reference implementation.

RTX4090 round 2 results: `mm-2x2.fut`

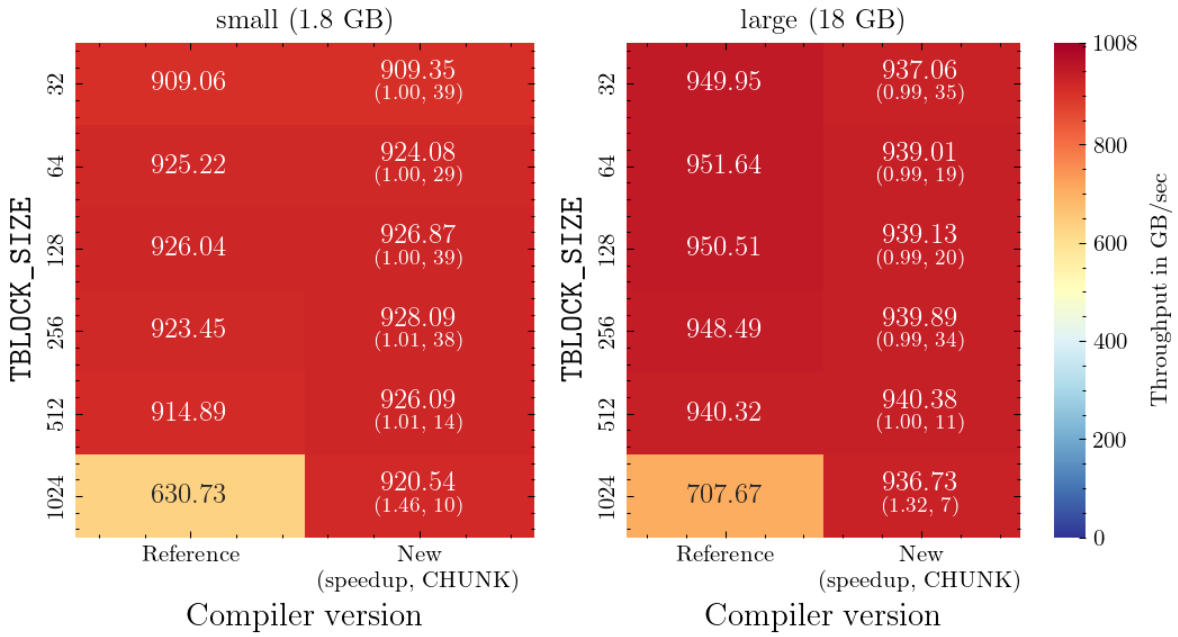

Figure 22: RTX4090, `mm-2x2.fut`, comparison with reference.

Evidently the reference implementation already performs remarkably well on the RTX4090, coming close to peak for most `TBLOCK_SIZE`s on the large input set.

Hence there was not much room for improvement to begin with, but the reference implementation even outperforms the new implementation by a slight margin for some `TBLOCK_SIZE`s on the large input set.

However, for some reason, the reference implementation did not do well for `TBLOCK_SIZE` = 1024, hence we see non-trivial speedup here.

RTX4090 round 2 results: `mm-3x3.fut`

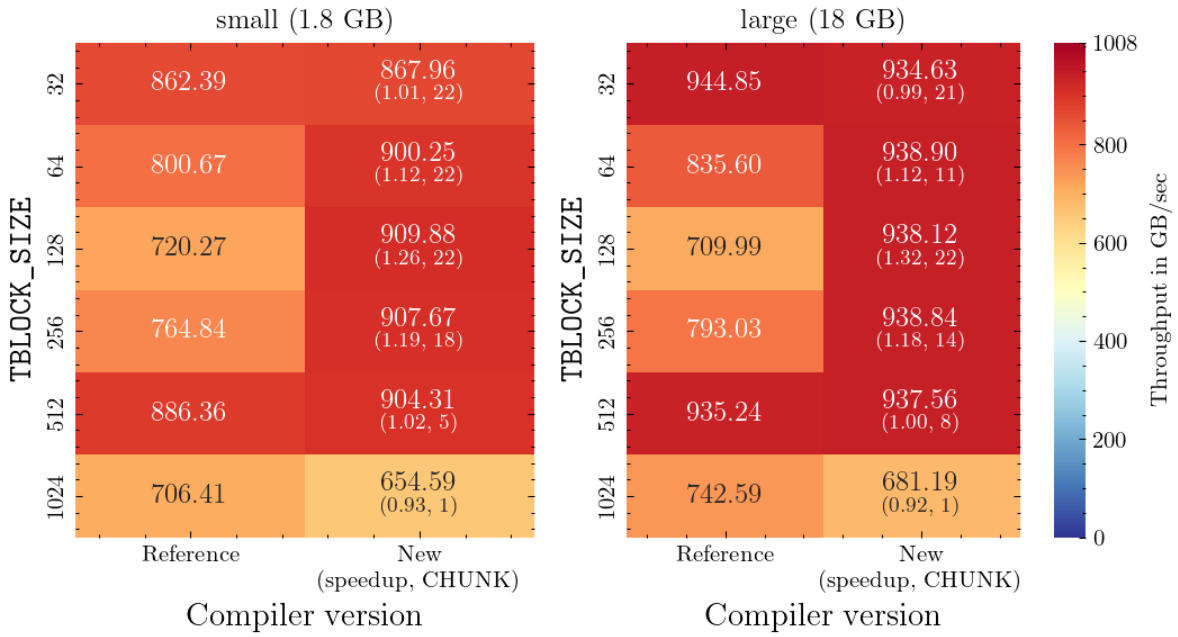

Figure 23: RTX4090, `mm-3x3.fut`, comparison with reference.

We see largely the same plot as with fig. 22: the reference implementation does relatively well for most `TBLOCK_SIZE`s, even coming close to peak and beating the new implementation for some values. Overall, however, the new implementation does significantly better with more consistent results across `TBLOCK_SIZE`s, save for `TBLOCK_SIZE` $= 1024$, where, interestingly, the optimal `CHUNK` was 1 (for which the generated kernel is essentially equivalent to that of the reference implementation, except with extra overhead in loop bookkeeping; hence the $\sim 8\%$ speeddown).

RTX4090 round 2 results: `mm-5x5.fut`

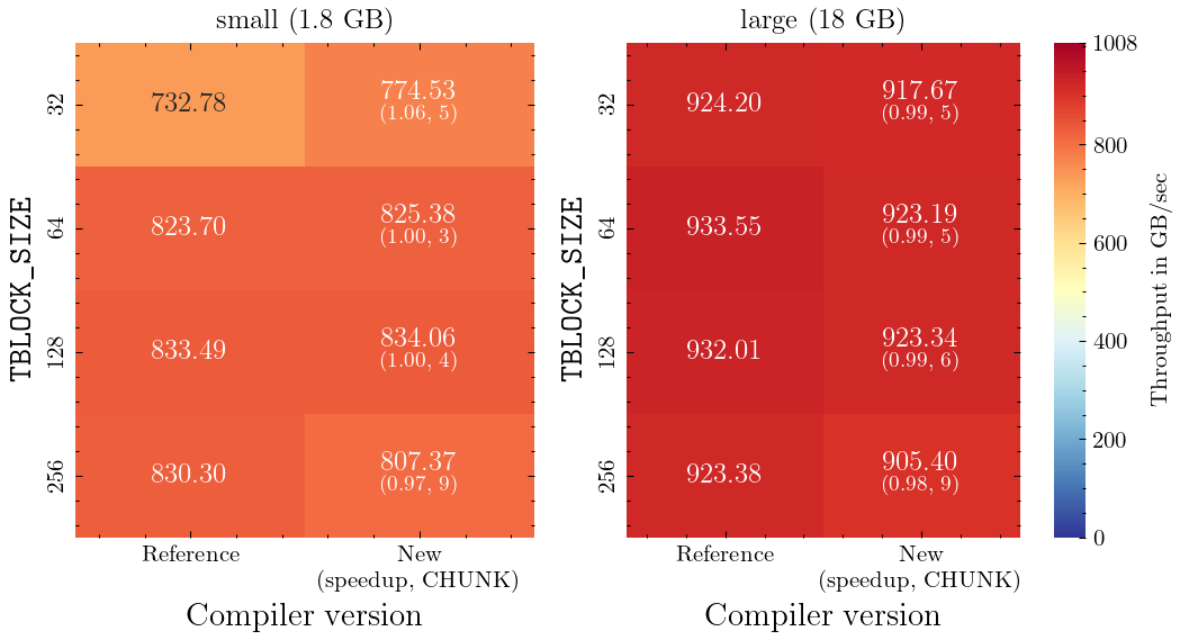

Figure 24: RTX4090, `mm-5x5.fut`, comparison with reference.

Save for the smallest `TBLOCK_SIZE` on the small input set, the two implementations are either equal in performance, or the reference implementation has a slight edge.

Note that, as with the A100, both implementations failed to launch the kernel for `TBLOCK_SIZE` $\geq$ 512.

RTX4090 round 2 results: `linear_function_composition.fut`

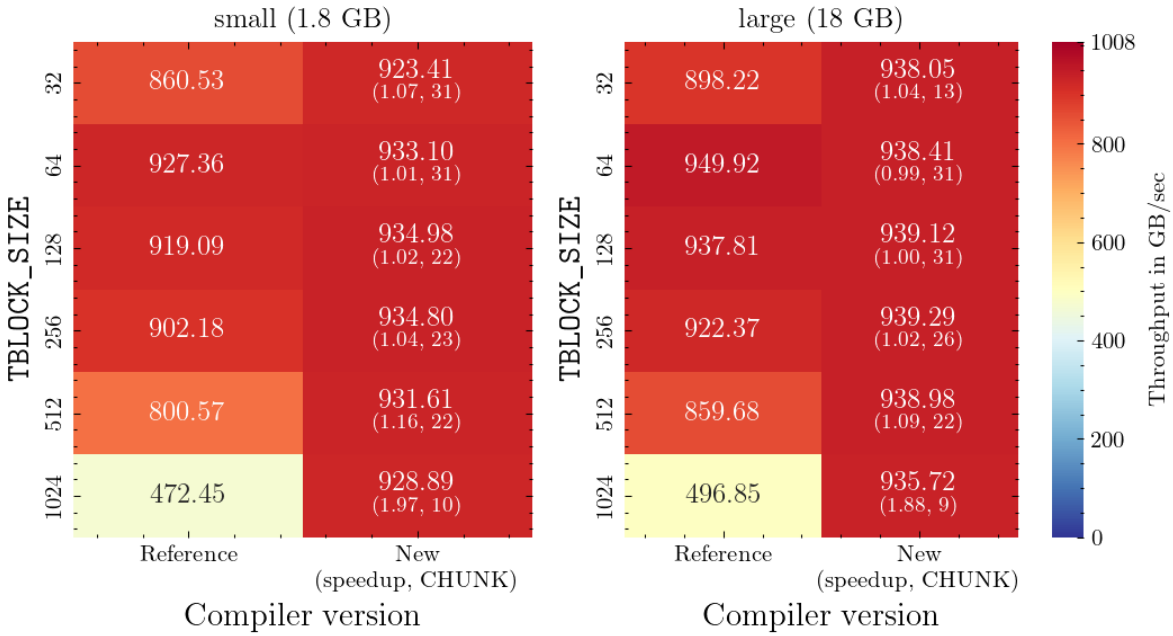

Figure 25: RTX4090, `linear_function_composition.fut`, comparison with reference.

For most `TBLOCK_SIZE`s, the two implementations perform about equally well, indicating again that the reference implementation was near optimal in certain cases for at least some GPUs. However, the new implementation is more consistent in measurements across `TBLOCK_SIZE`s, obtaining significant speedups for the larger `TBLOCK_SIZE`s on both input sets.

RTX4090 round 2 results: `mssp.fut`

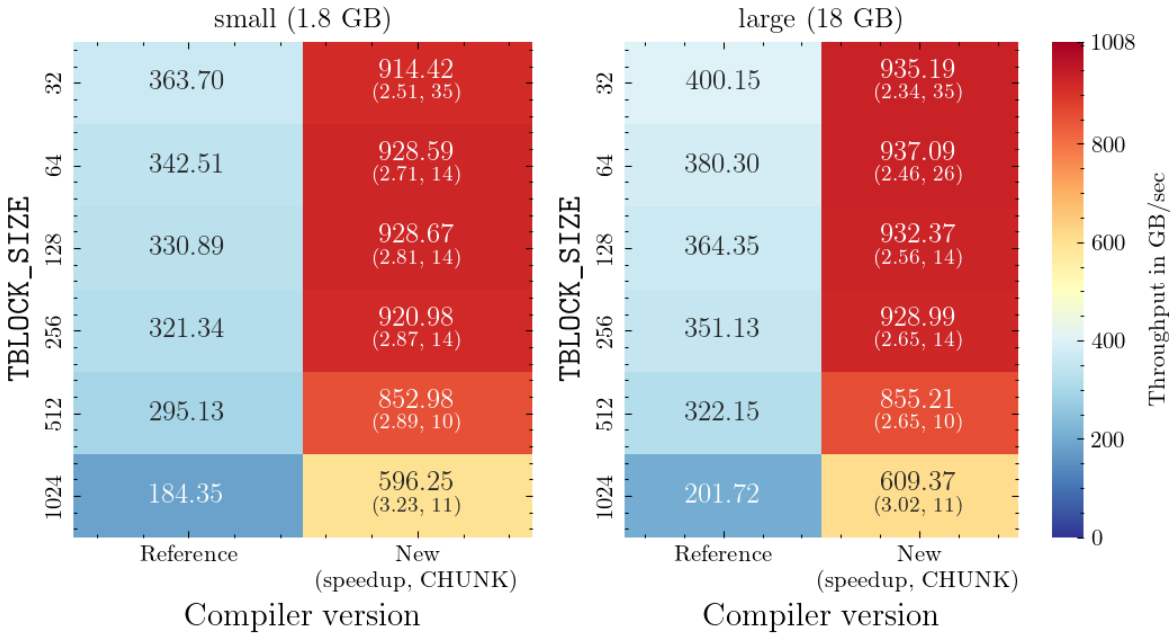

Figure 26: RTX4090, `mssp.fut`, comparison with reference.

Interestingly, this is the first plot of RTX4090 measurements in which we see very big speedups over the reference implementation. Evidently the reference implementation did not handle the `mss` operator very well, reaching only 39.7% of peak bandwidth, hence the big speedups.

As with most of the A100 measurements, throughput of the new implementation drops with `TBLOCK_SIZE`, while speedups grow.

RTX4090 round 2 results: `lssp.fut`

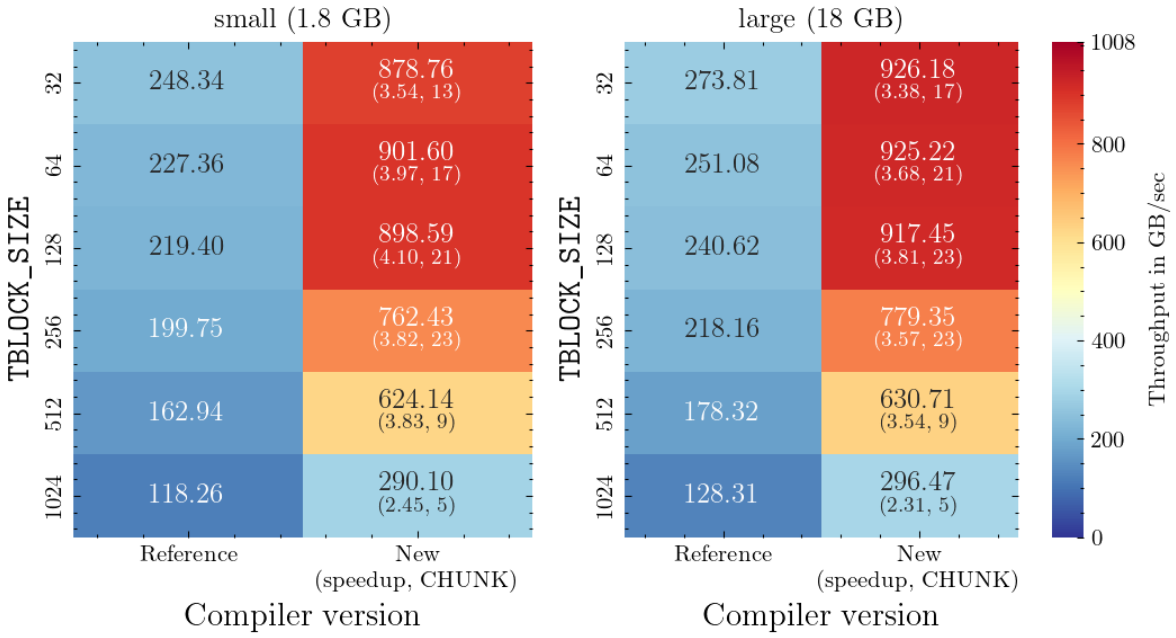

Figure 27: RTX4090, `lssp.fut`, comparison with reference.

As expected, we see largely the same pattern as for the RTX4090 `mssp.fut` comparison plot (fig. 26) – very big speedups due to the reference implementation not handling the `lss` operator well. In comparing with the RTX4090 `mssp.fut` tests, it seems that the new implementation drops in performance quicker as `TBLOCK_SIZE` grows than it did for `mssp.fut`, hence the speedups tend to *fall* as `TBLOCK_SIZE` grows, whereas for `mssp.fut`, the speedups *grew* with `TBLOCK_SIZE`.

5.4 Reproduction of benchmarking results

To reproduce benchmarking results, please first find the source code for our test programs and benchmarking test cases in appendix A.

As mentioned, all benchmarks of the new reduce kernel optimizations are run using commit `5234eb8` [6] of the Futhark compiler, so this must be installed. To force different `CHUNK` values, the code to automatically choose `CHUNK` must be disabled in the `CodeGen.ImpGen.GPU.SegRed` module before compiling the compiler.

All benchmarks of the reference reduce kernel implementation can be reproduced using commit `06732a5` of the compiler [7].

To reproduce the different test cases, please refer to the benchmark plan in section 5.1.

5.5 Benchmarking analysis

5.5.1 General benchmarking take-aways

Overall stability in `CHUNK` parameterization

In the following, we do not consider the stability or consistency of kernels for those values of `CHUNK` for which register spilling is perceived to occur.

For all the RTX4090 benchmarks except for `mssp.fut` and `lssp.fut`, the kernels seemed quite consistent in `CHUNK` parameterization, and hence most choices for `CHUNK` would result in good and near-optimal performance. This is less so the case for `mssp.fut` and `lssp.fut`, where `CHUNK` must be chosen a little more carefully to obtain good performance – for example, very low values as well as multiples of 8 (and, to some degree, 4) should be avoided.

For the A100 measurements, virtually all kernels appeared to be sensitive to multiples of 8 (and 4) and low `CHUNK`s. Outside of these values, there seemed to be some stability for all but `mssp.fut` and `lssp.fut`.

One pattern that *does* seem to show is that the lower the `TBLOCK_SIZE` is, the more stable the kernels are in `CHUNK` parameterization. This shows for results from both devices.

In conclusion, there unfortunately does not seem to be a general pattern in the consistency (or lack thereof) of performance across different parameterizations, except that the lower the `TBLOCK_SIZE`, the higher the chance of good performance, so long as the chosen `CHUNK` value is not very small, a multiple of 8 (or 4), or too high w.r.t. shared memory and register constraints (but the latter is easy to avoid given conservative bounds computed by the compiler).

Optimal `CHUNK` values

Unfortunately, there also does not seem to be a discernable pattern in optimal `CHUNK` value that we can generalize to use across different devices, programs, and values of `TBLOCK_SIZE`. For some device/program combinations, the best value seem to be in the upper ends of the `CHUNK` spectrum (ignoring `CHUNK`s for which register spilling occurred, obviously), while for others – in particular `mssp.fut` and `lssp.fut` for both devices – the optimal and near-optimal `CHUNK` values appear almost random and far between.

Best `TBLOCK_SIZE`s

For almost all device/program, the new kernel seemed to prefer the smaller `TBLOCK_SIZE`s. In some cases, a low `TBLOCK_SIZE` simply meant more headroom before register spilling and failure due to shared memory constraints, while for others the smaller `TBLOCK_SIZE`s simply gave better performance.

The smaller `TBLOCK_SIZE` gets, the higher we can push the sequentialization factor before e.g. register spilling starts (up to the hard limit of 255 registers per thread, as per [4]), so in this respect it is sensible. A lower `TBLOCK_SIZE` also means more shared memory per thread, but, as mentioned multiple times throughout this report, the shared memory constraint is rarely going to be the earliest limiting factor in the new reduce kernel.

In comparison with the reference implementation

For all A100 measurements, we saw significant improvement upon the reference implementation, with speedups generally in the range of x2-4 across the different programs and

parameterizations. However, some of these speedups should be taken with a grain of salt: For one, as discussed throughout section 5.3, the reference implementation did not perform particularly well for *any* of the six test programs on the A100. Second, all of the comparisons were made using the optimal `CHUNK` values, and while a good `CHUNK` was easy to obtain for most programs, the speedups over the reference implementation cannot be trivially guaranteed.

For the RTX4090, we did *not* see significant speedups for four out of six programs, namely `mm-2x2.fut`, `mm-3x3.fut`, `mm-5x5.fut`, and `linear_function_composition.fut`. As mentioned, this is largely because the reference implementation already did quite well for these programs on the RTX4090. However, it should be noted that the two most significant speeddowns were x0.92 and x0.93 (for the two input sets to `mm-3x3.fut` with `TBLOCK_SIZE` = 1024), that all other speeddowns were in the range of x0.97-0.99, and that the kernel was quite stable in `CHUNK` for all of these programs.

For the remaining two programs (`mssp.fut` and `lssp.fut`), we saw significant speedups, but as with most of the A100 measurements, speedups here were easy because the reference implementation did rather poorly. For `mssp.fut` the kernel was quite stable in parameterization, hence these speedups are reliable, however the case was less so for `lssp.fut`.

5.6 Benchmarking conclusion

The new kernel can unfortunately not be said to always be stable or reliable in parameterization without careful selection of `CHUNK` and `TBLOCK_SIZE`; as a general rule, low `TBLOCK_SIZE` values should be used whenever possible, but there are still programs and cases where performance varies greatly in `CHUNK` for even low `TBLOCK_SIZE`.

In summary across all devices and programs, it seems the new implementation did quite well as compared to the reference implementation as well as relative to peak memory bandwidth for the two devices. For those cases where the new implementation was equal to or even outperformed by the reference implementation on the RTX4090, the two implementations each measured relatively close to peak bandwidth, and because the new implementation was stable in parameterization for virtually all of these cases, we argue there are large benefits to be had from the new implementation in almost all cases, and that the performance loss seen in a select few cases is small and rare enough that it is outweighed by the benefits.

6 Conclusion and future work

We successfully implemented tblock-virtualization in the single-pass scan kernel as well as the optimization to non-commutative and primitive reductions as described in section 3. Both implementations were successfully validate tested. Benchmark testing of the new reduce kernel showed good and promising results for almost all test cases, however we also found the new kernel to not be completely stable in parameterization across different devices and test programs.

6.1 Future work

In this section, we explore and discuss possible future work. Some ideas here may not warrant the time and effort to implement, while others may may not at all be feasible for implementation into the Futhark compiler, but are still interesting to consider from a learning standpoint.

The future work discussed pertain to (code generation of) the new reduce kernel code generation unless otherwise noted.

More deliberate `CHUNK` selection

As mentioned in section 3.2.2, the compiler code used to choose `CHUNK` may benefit from revision. The overhead it adds is optimistic: For one, pointers should be assumed to be 64 bit, meaning an address takes two registers just to hold, so extra overhead should be added to account for the actual computations, which are not trivial, especially in the segmented case. Also, as the author of [3] states, the chosen amount of overhead assumes no extra registers are needed to apply the binary operator, although the compiler in most cases should be able to figure out how to reuse registers.

In a different sense, the computation is pessimistic: It is based on a constant `TBLOCK_SIZE` = 1024, such that the same `CHUNK` value is always used for a given reduction regardless of `TBLOCK_SIZE`. This means bounds on shared memory and registers are never broken, however it also sacrifices sequentialization, and potentially performance, if for a given reduction the optimal parameterization is one which uses a `CHUNK` larger than is possible for `TBLOCK_SIZE` = 1024. Ideally the `CHUNK` value should be based on varying `TBLOCK_SIZE`, which should be possible since the kernel is compiled between the time of kernel parameter configuration and kernel launch.

In addition to fine-tuning the overhead and basing `CHUNK` on varying `TBLOCK_SIZE`, we might still benefit from more deliberate `CHUNK` selection since, as we saw in section 5, the optimal `CHUNK` value is typically not the largest possible choice, and we also saw (sometimes significant) dips in performance when `CHUNK` was a multiple of 8. Hence to obtain optimal kernel parameterization universally, we might have to take deliberate action to e.g. avoid shared memory bank conflicts and to optimize for occupancy.

Further exploration of register spilling

In some relation to more deliberate `CHUNK` selection, it may be a good idea to explore methods to more reliably estimate the number of available thread registers for a given kernel parameterization, and more consistently avoid spilling of important registers (where a register is “important” if it is used in the virtualization loop whether or not it holds reduction operand(s)). As discussed in section 3, the amount of available shared memory puts a hard

cap on `CHUNK`, while the actual chosen value is almost always bounded by the (possibly pessimistic) estimate on the number of available registers, since the bound imposed by this is almost always smaller than the bound imposed by shared memory.

On the other hand, it is plausible that *some* register spilling is perfectly okay and may even be preferable in some cases, if e.g. the increase in sequentialization makes up for spilling certain non-essential variables. Using low-level kernel profiling tools one can gain insight into exactly which variables are spilled when increasing `CHUNK`, and which variables are spilled when performance degrades.

Optimize shared memory usage

As is, the reduce kernel reads elements from global to register memory; applies the map function (if any) in register memory; "effectualizes" collective copies by writing to shared memory and back to register memory; and finally, performs per-thread reductions of private chunks. This means that the collective copies and per-thread reductions are made on elements of the map-out type, meaning the computation of the sequentialization factor is based on the size of the map-out type. This is generally OK, however, as we saw in section 5, our implementation performed the worst (in terms of memory throughput) for those reductions which map input array elements to elements of a larger type going into the reduction, i.e. the `lssp` and `mssp` programs. If then, on the other hand, the map application is postponed until it is needed (i.e. until the point of applying the binary operator) whenever the map-out element type is larger than the map-in, then we may save significant register space and decrease the number of shared memory accesses, in other words increase the sequentialization factor, by not manifesting the larger map-out results in shared memory and per-thread chunk arrays.

We tested this using a hand-written CUDA prototype and saw the `mssp` reduction obtain performance similar to the other operators, but unfortunately did not document this.

As is, this is not feasible in Futhark, since in the Futhark IR the map function is represented as a kernel body (which can later be fused with e.g. the reduce kernel) which may contain arbitrary code and which does not have a concrete notion of *input/parameter* arrays or *output/result* arrays, as is more so the case for the IR representation of reduce; in fact, from a semantic viewpoint, it is not the map-out arrays which are copied to shared memory, but rather the *reduce-in* arrays.

Optimizations in the small-segments kernel

Our optimization to the reduce kernel touches only the non-segmented and large-segments segmented reduction kernels. This is because our optimizations touches only the stage one virtualization loop, and the small-segmented segmented kernel uses an entirely different algorithm from the two-stage algorithm. Hence it may be both interesting and beneficial to examine whether there is room for similar or other optimizations in the small-segments segmented case.

Benchmark test more different devices and test programs

As we saw in sections 5.2 and 5.3, performance of the new reduce kernel varied greatly between different combinations of the two devices (A100 and RTX4090) and test programs used. It may be beneficial to benchmark using more different devices and test programs to more accurately profile the performance and intricacies of the kernel.

Benchmarking of the single-pass scan kernel

During benchmarking we decided to forego testing the single-pass scan kernel for lack of time, and because the reduce kernel changes were deemed more important and interesting to properly benchmark and analyze. Even though our changes to the single-pass scan kernel were small and performance should not be significantly affected in most cases, the addition of tblock-virtualization can potentially have a big impact on performance if `NUM_TBLOCKS` is chosen poorly for a given scan program, and hence it is a good idea to profile the performance of the kernel under tblock-virtualization, as well as its sensitivity to `NUM_TBLOCKS` (and, perhaps to a lesser degree, `TBLOCK_SIZE`) parameterization.

7 References

[1] Duane Merrill and Michael Garland, March 2016. URL `https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf`.

[2] Andreas Nicolaisen and Marco Aslak Persson, November 2020. URL `https://futhark-lang.org/student-projects/marco-andreas-scan.pdf`.

[3] Morten Clausen, May 2021. URL `https://futhark-lang.org/student-projects/morten-msc-thesis.pdf`.

[4] NVIDIA Corporation, 12.3 edition. URL `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[5] The Futhark Hackers. Futhark GitHub repository (new single-pass scan implementation). `https://github.com/diku-dk/futhark/tree/f7a36ee`, 2023.

[6] The Futhark Hackers. Futhark GitHub repository (new reduce implementation). `https://github.com/diku-dk/futhark/tree/5234eb8`, 2023.

[7] The Futhark Hackers. Futhark GitHub repository (reference reduce implementation). `https://github.com/diku-dk/futhark/tree/06732a5`, 2023.

Appendix

A Benchmarking test programs

```
type f32_2x2 = ((f32, f32), (f32, f32))
def mm_2x2 (x: f32_2x2) (y: f32_2x2) =
  ( (x.0.0 * y.0.0  +  x.0.1 * y.1.0,
     x.0.0 * y.0.1  +  x.0.1 * y.1.1),
    (x.1.0 * y.0.0  +  x.1.1 * y.1.0,
     x.1.0 * y.0.1  +  x.1.1 * y.1.1))

def identity_2x2 = ((1f32, 0f32), (0f32, 1f32))

def from_arrs_2x2 [n] (xs: [n][4]f32): [n]f32_2x2 =
  map (\x -> ((x[0], x[1]),
              (x[2], x[3]))
      ) xs
-- ==
-- entry: A100
-- random input { [187500000][4]f32  }
-- random input { [1875000000][4]f32 }
entry A100 [n] (xs: [n][4]f32): f32_2x2 =
  from_arrs_2x2 xs |> reduce mm_2x2 identity_2x2
-- ==
-- entry: rtx_4090
-- random input { [112500000][4]f32 }
-- random input { [1125000000][4]f32 }
entry rtx_4090 [n] (xs: [n][4]f32): f32_2x2 =
  from_arrs_2x2 xs |> reduce mm_2x2 identity_2x2
```

Figure 28: `mm-2x2.fut`

```
type f32_3x3 = ((f32, f32, f32),
                (f32, f32, f32),
                (f32, f32, f32))

def mm_3x3 (x: f32_3x3) (y: f32_3x3): f32_3x3 =
  ( (x.0.0 * y.0.0  +  x.0.1 * y.1.0  +  x.0.2 * y.2.0,
     x.0.0 * y.0.1  +  x.0.1 * y.1.1  +  x.0.2 * y.2.1,
     x.0.0 * y.0.2  +  x.0.1 * y.1.2  +  x.0.2 * y.2.2),
    (x.1.0 * y.0.0  +  x.1.1 * y.1.0  +  x.1.2 * y.2.0,
     x.1.0 * y.0.1  +  x.1.1 * y.1.1  +  x.1.2 * y.2.1,
     x.1.0 * y.0.2  +  x.1.1 * y.1.2  +  x.1.2 * y.2.2),
    (x.2.0 * y.0.0  +  x.2.1 * y.1.0  +  x.2.2 * y.2.0,
     x.2.0 * y.0.1  +  x.2.1 * y.1.1  +  x.2.2 * y.2.1,
     x.2.0 * y.0.2  +  x.2.1 * y.1.2  +  x.2.2 * y.2.2))

def identity_3x3: f32_3x3 = ((1f32, 0f32, 0f32),
                             (0f32, 1f32, 0f32),
                             (0f32, 0f32, 1f32))

def from_arrs_3x3 [n] (xs: [n][9]f32): [n]f32_3x3 =
  map (\x -> ((x[0], x[1], x[2]),
              (x[3], x[4], x[5]),
              (x[6], x[7], x[8]))
      ) xs
-- ==
-- entry: A100
-- random input { [83333333][9]f32 }
-- random input { [833333333][9]f32 }
entry A100 [n] (inp: [n][9]f32) =
  from_arrs_3x3 inp |> reduce mm_3x3 identity_3x3
-- ==
-- entry: rtx_4090
-- random input { [50000000][9]f32  }
-- random input { [500000000][9]f32 }
entry rtx_4090 [n] (inp: [n][9]f32) =
  from_arrs_3x3 inp |> reduce mm_3x3 identity_3x3
```

Figure 29: `mm-3x3.fut`

```
type f32_5x5 = ((f32, f32, f32, f32, f32),
                (f32, f32, f32, f32, f32),
                (f32, f32, f32, f32, f32),
                (f32, f32, f32, f32, f32),
                (f32, f32, f32, f32, f32))
def mm_5x5 (a: f32_5x5) (b: f32_5x5): f32_5x5 =
  ( (a.0.0 * b.0.0  +  a.0.1 * b.1.0  +  a.0.2 * b.2.0  +  a.0.3 * b.3.0  +  a.0.4 * b.4.0,
     a.0.0 * b.0.1  +  a.0.1 * b.1.1  +  a.0.2 * b.2.1  +  a.0.3 * b.3.1  +  a.0.4 * b.4.1,
     a.0.0 * b.0.2  +  a.0.1 * b.1.2  +  a.0.2 * b.2.2  +  a.0.3 * b.3.2  +  a.0.4 * b.4.2,
     a.0.0 * b.0.3  +  a.0.1 * b.1.3  +  a.0.2 * b.2.3  +  a.0.3 * b.3.3  +  a.0.4 * b.4.3,
     a.0.0 * b.0.4  +  a.0.1 * b.1.4  +  a.0.2 * b.2.4  +  a.0.3 * b.3.4  +  a.0.4 * b.4.4),
    (a.1.0 * b.0.0  +  a.1.1 * b.1.0  +  a.1.2 * b.2.0  +  a.1.3 * b.3.0  +  a.1.4 * b.4.0,
     a.1.0 * b.0.1  +  a.1.1 * b.1.1  +  a.1.2 * b.2.1  +  a.1.3 * b.3.1  +  a.1.4 * b.4.1,
     a.1.0 * b.0.2  +  a.1.1 * b.1.2  +  a.1.2 * b.2.2  +  a.1.3 * b.3.2  +  a.1.4 * b.4.2,
     a.1.0 * b.0.3  +  a.1.1 * b.1.3  +  a.1.2 * b.2.3  +  a.1.3 * b.3.3  +  a.1.4 * b.4.3,
     a.1.0 * b.0.4  +  a.1.1 * b.1.4  +  a.1.2 * b.2.4  +  a.1.3 * b.3.4  +  a.1.4 * b.4.4),
    (a.2.0 * b.0.0  +  a.2.1 * b.1.0  +  a.2.2 * b.2.0  +  a.2.3 * b.3.0  +  a.2.4 * b.4.0,
     a.2.0 * b.0.1  +  a.2.1 * b.1.1  +  a.2.2 * b.2.1  +  a.2.3 * b.3.1  +  a.2.4 * b.4.1,
     a.2.0 * b.0.2  +  a.2.1 * b.1.2  +  a.2.2 * b.2.2  +  a.2.3 * b.3.2  +  a.2.4 * b.4.2,
     a.2.0 * b.0.3  +  a.2.1 * b.1.3  +  a.2.2 * b.2.3  +  a.2.3 * b.3.3  +  a.2.4 * b.4.3,
     a.2.0 * b.0.4  +  a.2.1 * b.1.4  +  a.2.2 * b.2.4  +  a.2.3 * b.3.4  +  a.2.4 * b.4.4),
    (a.3.0 * b.0.0  +  a.3.1 * b.1.0  +  a.3.2 * b.2.0  +  a.3.3 * b.3.0  +  a.3.4 * b.4.0,
     a.3.0 * b.0.1  +  a.3.1 * b.1.1  +  a.3.2 * b.2.1  +  a.3.3 * b.3.1  +  a.3.4 * b.4.1,
     a.3.0 * b.0.2  +  a.3.1 * b.1.2  +  a.3.2 * b.2.2  +  a.3.3 * b.3.2  +  a.3.4 * b.4.2,
     a.3.0 * b.0.3  +  a.3.1 * b.1.3  +  a.3.2 * b.2.3  +  a.3.3 * b.3.3  +  a.3.4 * b.4.3,
     a.3.0 * b.0.4  +  a.3.1 * b.1.4  +  a.3.2 * b.2.4  +  a.3.3 * b.3.4  +  a.3.4 * b.4.4),
    (a.4.0 * b.0.0  +  a.4.1 * b.1.0  +  a.4.2 * b.2.0  +  a.4.3 * b.3.0  +  a.4.4 * b.4.0,
     a.4.0 * b.0.1  +  a.4.1 * b.1.1  +  a.4.2 * b.2.1  +  a.4.3 * b.3.1  +  a.4.4 * b.4.1,
     a.4.0 * b.0.2  +  a.4.1 * b.1.2  +  a.4.2 * b.2.2  +  a.4.3 * b.3.2  +  a.4.4 * b.4.2,
     a.4.0 * b.0.3  +  a.4.1 * b.1.3  +  a.4.2 * b.2.3  +  a.4.3 * b.3.3  +  a.4.4 * b.4.3,
     a.4.0 * b.0.4  +  a.4.1 * b.1.4  +  a.4.2 * b.2.4  +  a.4.3 * b.3.4  +  a.4.4 * b.4.4))
let identity_5x5: f32_5x5 =
  ((1f32, 0f32, 0f32, 0f32, 0f32),
   (0f32, 1f32, 0f32, 0f32, 0f32),
   (0f32, 0f32, 1f32, 0f32, 0f32),
   (0f32, 0f32, 0f32, 1f32, 0f32),
   (0f32, 0f32, 0f32, 0f32, 1f32)
  )
def from_arrs_5x5 [n] (xs: [n][25]f32): [n]f32_5x5 =
  map (\x -> ((x[ 0], x[ 1], x[ 2], x[ 3], x[ 4]),
              (x[ 5], x[ 6], x[ 7], x[ 8], x[ 9]),
              (x[10], x[11], x[12], x[13], x[14]),
              (x[15], x[16], x[17], x[18], x[19]),
              (x[20], x[21], x[22], x[23], x[24])
             )) xs

-- ==
-- entry: A100
-- random input { [30000000][25]f32 }
-- random input { [300000000][25]f32 }
entry A100 [n] (inp: [n][25]f32) =
  from_arrs_5x5 inp |> reduce mm_5x5 identity_5x5
-- ==
-- entry: rtx_4090
-- random input { [18000000][25]f32 }
-- random input { [180000000][25]f32 }
entry rtx_4090 [n] (inp: [n][25]f32) =
  from_arrs_5x5 inp |> reduce mm_5x5 identity_5x5
```

Figure 30: `mm-5x5.fut`

```
def linear_function_composition ((a1, b1): (f32, f32))
                                ((a2, b2): (f32, f32))
                                : (f32, f32) =
  (a1 * a2, a1 * b2 + b1)
def ne = (1f32, 0f32)

-- ==
-- entry: rtx_4090
-- random input { [225000000]f32  [225000000]f32 }
-- random input { [2250000000]f32 [2250000000]f32 }
entry rtx_4090 [n] (as: [n]f32) (bs: [n]f32): (f32, f32) =
  zip as bs |> reduce linear_function_composition ne
-- ==
-- entry: A100
-- random input { [375000000]f32  [375000000]f32 }
-- random input { [3750000000]f32 [3750000000]f32 }
entry A100 [n] (as: [n]f32) (bs: [n]f32): (f32, f32) =
  zip as bs |> reduce linear_function_composition ne
```

Figure 31: `linear_function_composition.fut`

```
def mss (xs: []i32): i32 =
  let mapOp x = (i32.max x 0, i32.max x 0, i32.max x 0, x)
  let redOp (mssx, misx, mcsx, tsx) (mssy, misy, mcsy, tsy) =
    ( i32.max mssx (i32.max mssy (mcsx + misy)),
      i32.max misx (tsx+misy),
      i32.max mcsy (mcsx+tsy),
      tsx + tsy)
  let ne = (0, 0, 0, 0)
  in map mapOp xs |> reduce redOp ne |> (.0)

-- ==
-- entry: A100
-- random input { [750000000]i32 }
-- random input { [7500000000]i32 }
entry A100 (xs: []i32): i32 =
  mss xs
-- ==
-- entry: rtx_4090
-- random input { [250000000]i32 }
-- random input { [2500000000]i32 }
entry rtx_4090 (xs: []i32): i32 =
  mss xs
```

Figure 32: `mssp.fut`

```
def lss [n] 't (pred1: t -> bool) (pred2: t -> t -> bool) (xs: [n]t): i64 =
  let mapOp x =
    let xmatch = i64.bool (pred1 x)
    in (xmatch, xmatch, xmatch, 1i64, x, x)

  let redOp (lssx, lisx, lcsx, tlx, firstx, lastx)
            (lssy, lisy, lcsy, tly, firsty, lasty) =
    let connect = pred2 lastx firsty || tlx == 0 || tly == 0

    let newlss = i64.max (i64.bool connect * (lcsx + lisy)) (i64.max lssx lssy)
    let newlis = lisx + (i64.bool (lisx == tlx && connect) * lisy)
    let newlcs = lcsy + (i64.bool (lcsy == tly && connect) * lcsx)

    let first = if tlx == 0 then firsty else firstx
    let last  = if tly == 0 then lastx else lasty
    in (newlss, newlis, newlcs, tlx+tly, first, last)

  let ne = (0, 0, 0, 0, xs[0], xs[0])

  in map mapOp xs |> reduce redOp ne |> (.0)

-- ==
-- entry: rtx_4090
-- random input { [450000000]i32 }
-- random input { [4500000000]i32 }
entry rtx_4090 (xs: []i32): i64 =
  lss (const true) (<=) xs
-- ==
-- entry: A100
-- random input { [750000000]i32 }
-- random input { [7500000000]i32 }
entry A100 (xs: []i32): i64 =
  lss (const true) (<=) xs
```

Figure 33: `lssp.fut`

B Validation test program entry points and test cases

```
-- ==
-- entry: lss_validation lss_scan_validation
-- random input { [750000000]i32 } auto output
entry lss_validation (xs: []i32): i64 =
  lss (const true) (<=) xs
entry lss_scan_validation (xs: []i32): []i64 =
  lss (const true) (<=) xs
-- ==
-- entry: lss_seg_validation lss_segscan_validation
-- random input { [10][75000000]i32 } auto output
-- random input { [100][7500000]i32 } auto output
-- random input { [1000][750000]i32 } auto output
-- random input { [10000][75000]i32 } auto output
-- random input { [100000][7500]i32 } auto output
entry lss_seg_validation (xss: [][]i32): []i64 =
  map (lss (const true) (<=)) xss
entry lss_segscan_validation (xss: [][]i32): [][]i64 =
  map (lss_scan (const true) (<=)) xss
```

Figure 34: `lssp.fut`, additional validation entry points and test cases

```
def to_flat_arr (x: f32_5x5): [25]f32 =
  [x.0.0, x.0.1, x.0.2, x.0.3, x.0.4,
   x.1.0, x.1.1, x.1.2, x.1.3, x.1.4,
   x.2.0, x.2.1, x.2.2, x.2.3, x.2.4,
   x.3.0, x.3.1, x.3.2, x.3.3, x.3.4,
   x.4.0, x.4.1, x.4.2, x.4.3, x.4.4]
-- ==
-- entry: mm_5x5_validation mm_5x5_scan_validation
-- random input { [30000000][25]f32  } auto output
entry mm_5x5_validation [n] (inp: [n][25]f32) =
  from_arrs_5x5 inp |> reduce mm_5x5 identity_5x5 |> to_flat_arr
entry mm_5x5_scan_validation [n] (inp: [n][25]f32) =
  from_arrs_5x5 inp |> scan mm_5x5 identity_5x5 |> to_flat_arr
-- ==
-- entry: mm_5x5_seg_validation mm_5x5_segscan_validation
-- random input { [10][3000000][25]f32 } auto output
-- random input { [100][300000][25]f32 } auto output
-- random input { [1000][30000][25]f32 } auto output
-- random input { [10000][3000][25]f32 } auto output
-- random input { [100000][300][25]f32 } auto output
entry mm_5x5_seg_validation [] (inp: [][][25]f32) =
  map (from_arrs_5x5 >-> reduce mm_5x5 identity_5x5) inp |> map to_flat_arr
entry mm_5x5_segscan_validation [] (inp: [][][25]f32) =
  map (from_arrs_5x5 >-> scan mm_5x5 identity_5x5) inp |> map to_flat_arr
```

Figure 35: `mm-5x5.fut`, additional validation entry points and test cases

C Small input set benchmarking plots

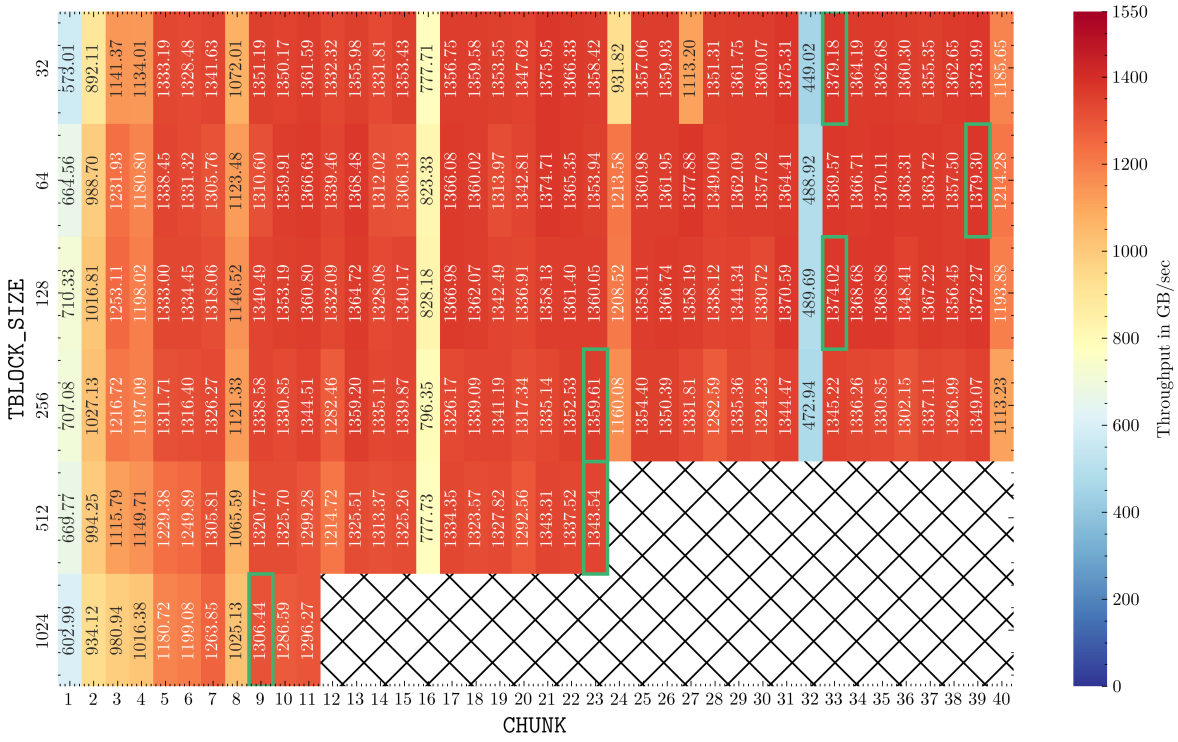

Figure 36: A100, `mm-2x2.fut`, small dataset (∼3.0 GB)

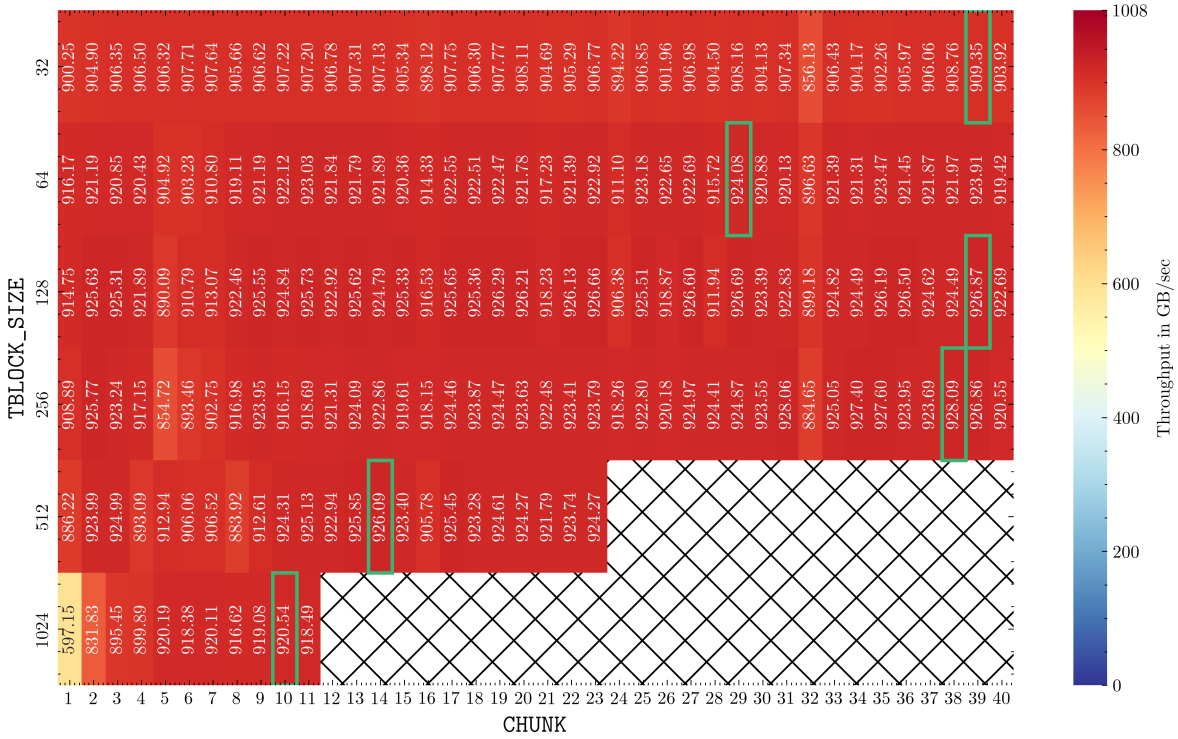

Figure 37: RTX4090, `mm-2x2.fut`, small dataset (∼1.8 GB)

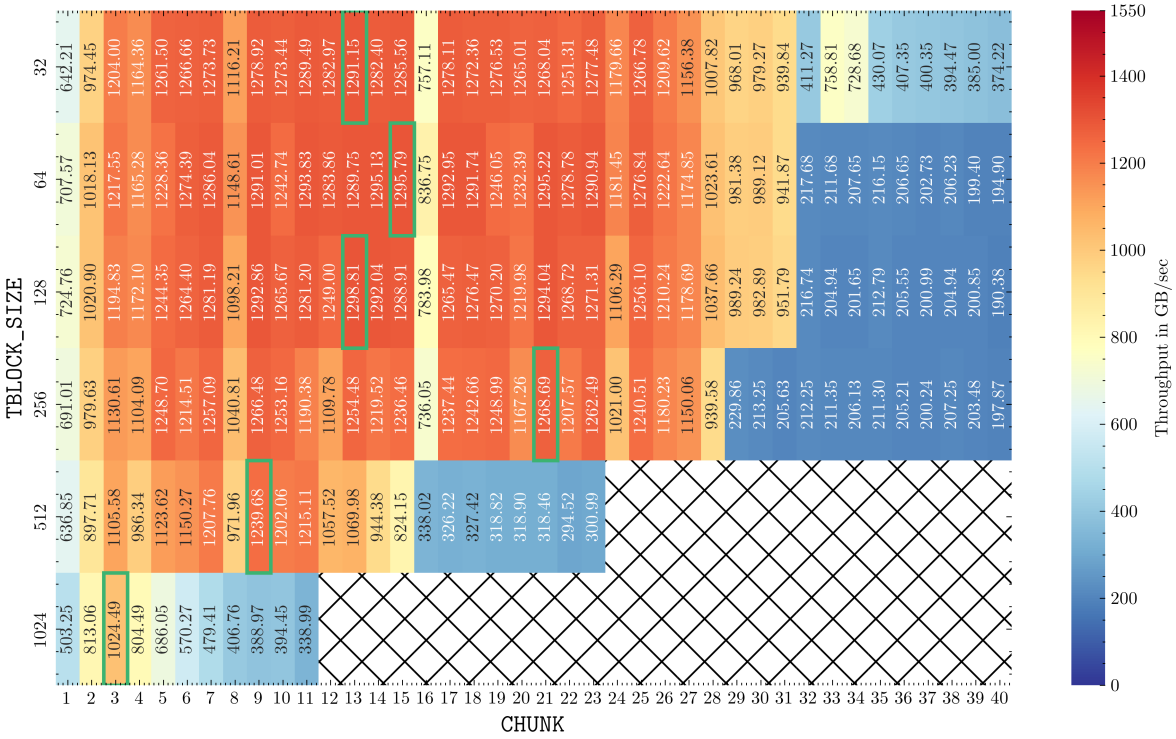

Figure 38: A100, `mm-3x3.fut`, small dataset (∼3.0 GB)

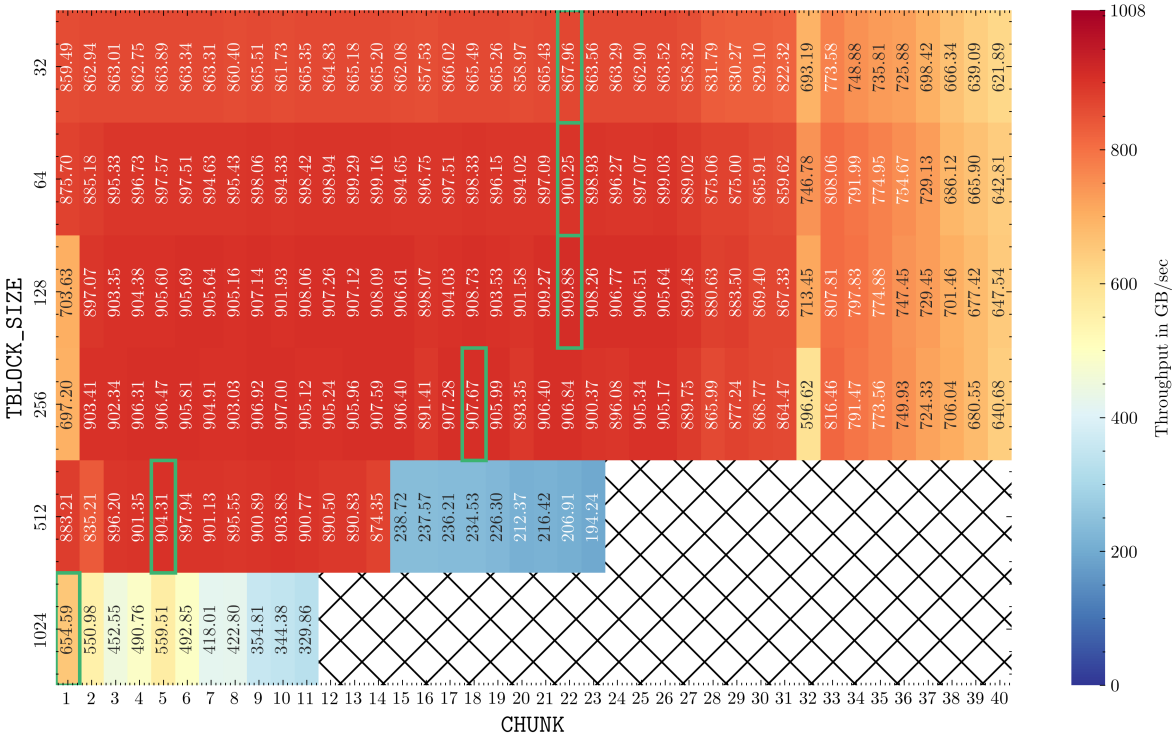

Figure 39: RTX4090, `mm-3x3.fut`, small dataset (∼1.8 GB)

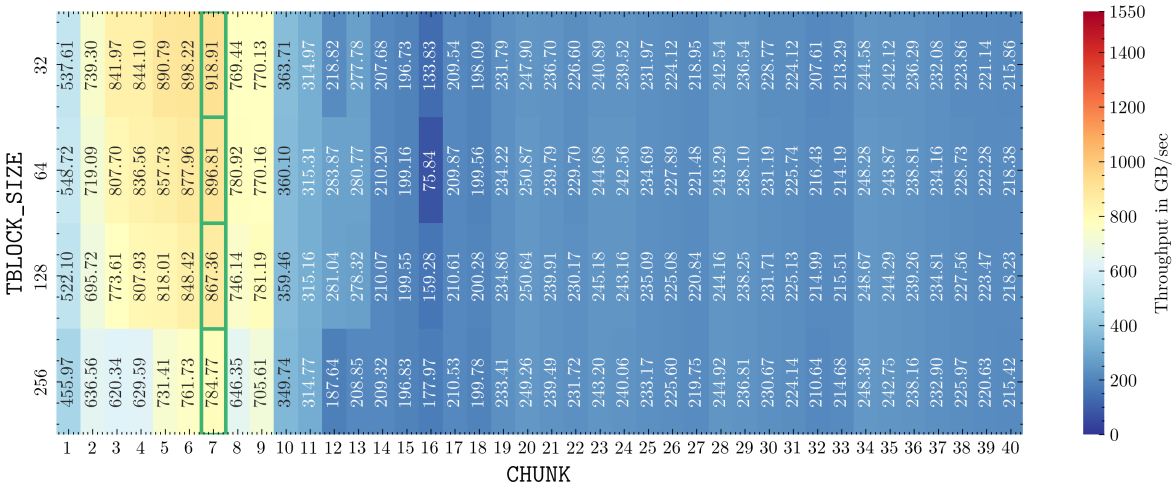

Figure 40: A100, `mm-5x5.fut`, small dataset (∼3.0 GB)

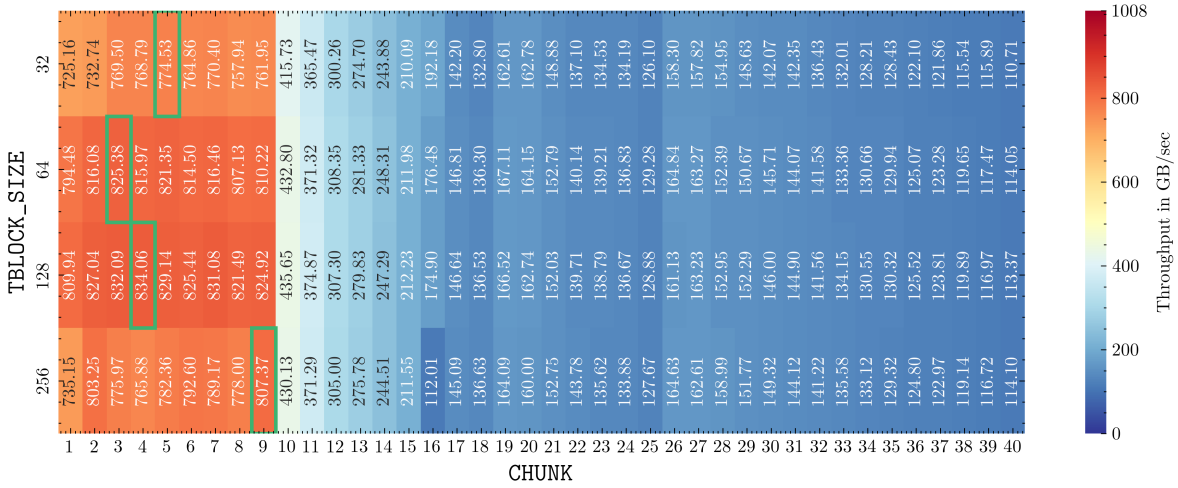

Figure 41: RTX4090, `mm-5x5.fut`, small dataset (∼1.8 GB)

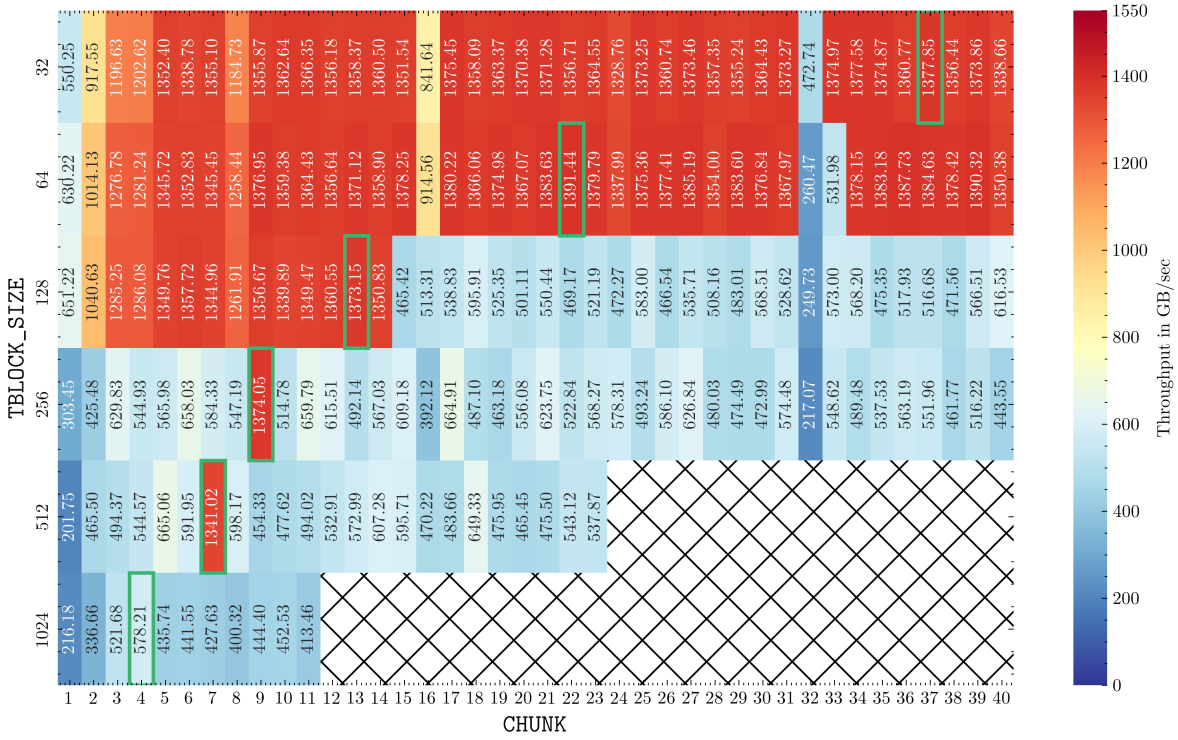

Figure 42: A100, `linear_function_composition.fut`, small dataset (∼3.0 GB)

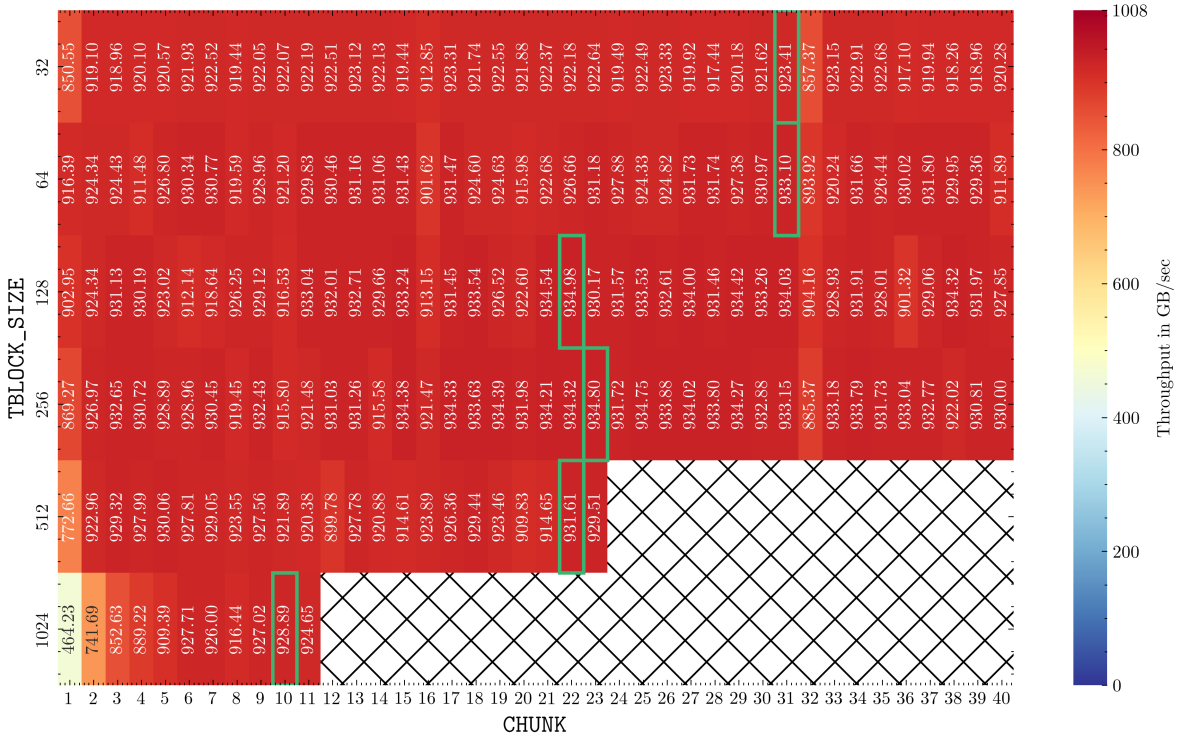

Figure 43: RTX4090, `linear_function_composition.fut`, small dataset (∼1.8 GB)

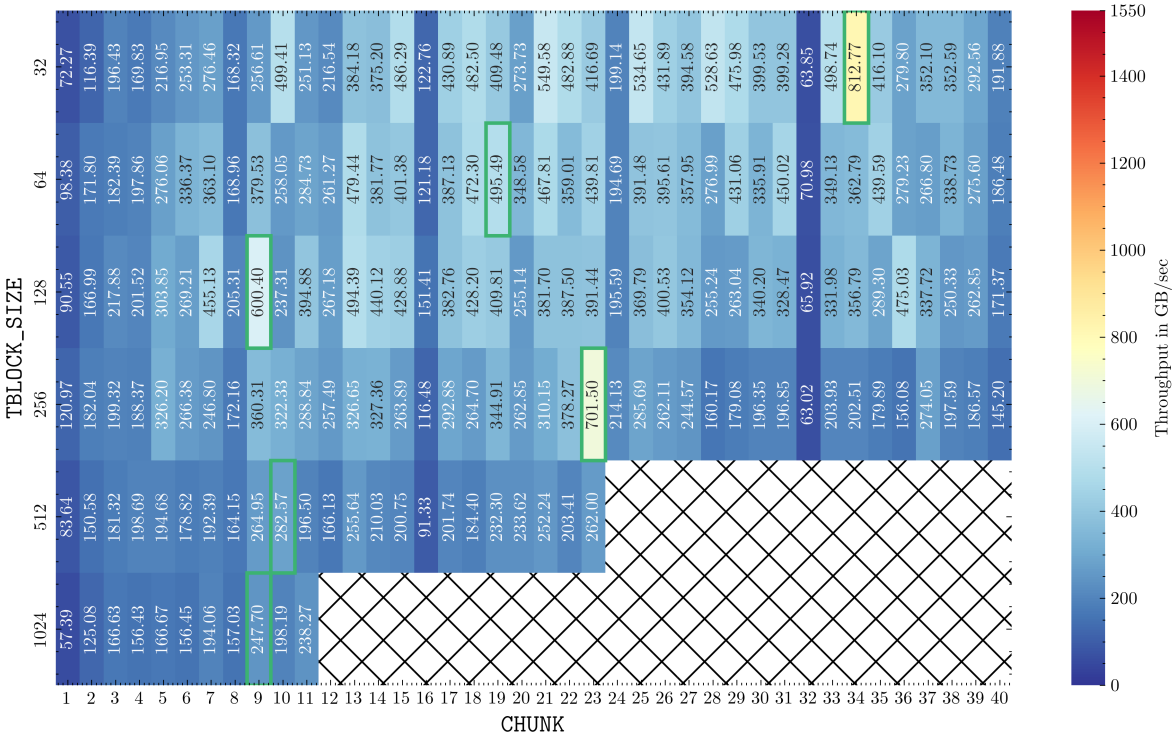

Figure 44: A100, `mssp.fut`, small dataset (∼3.0 GB)

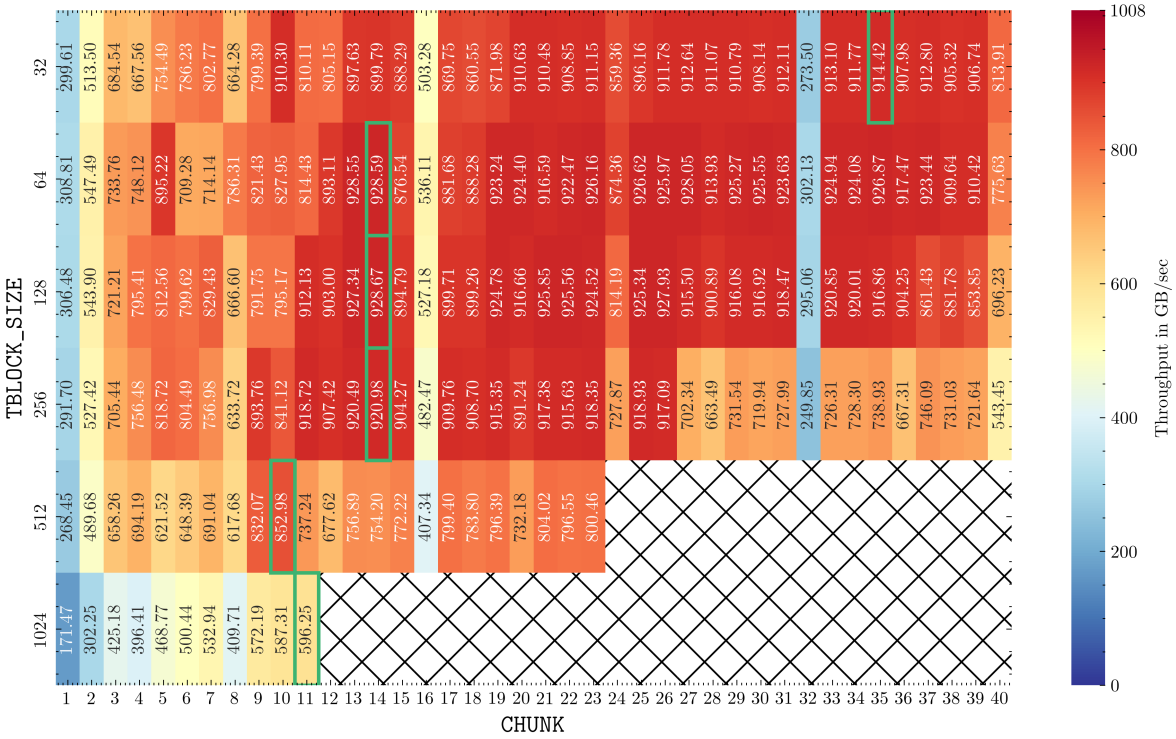

Figure 45: RTX4090, `mssp.fut`, small dataset (∼1.8 GB)

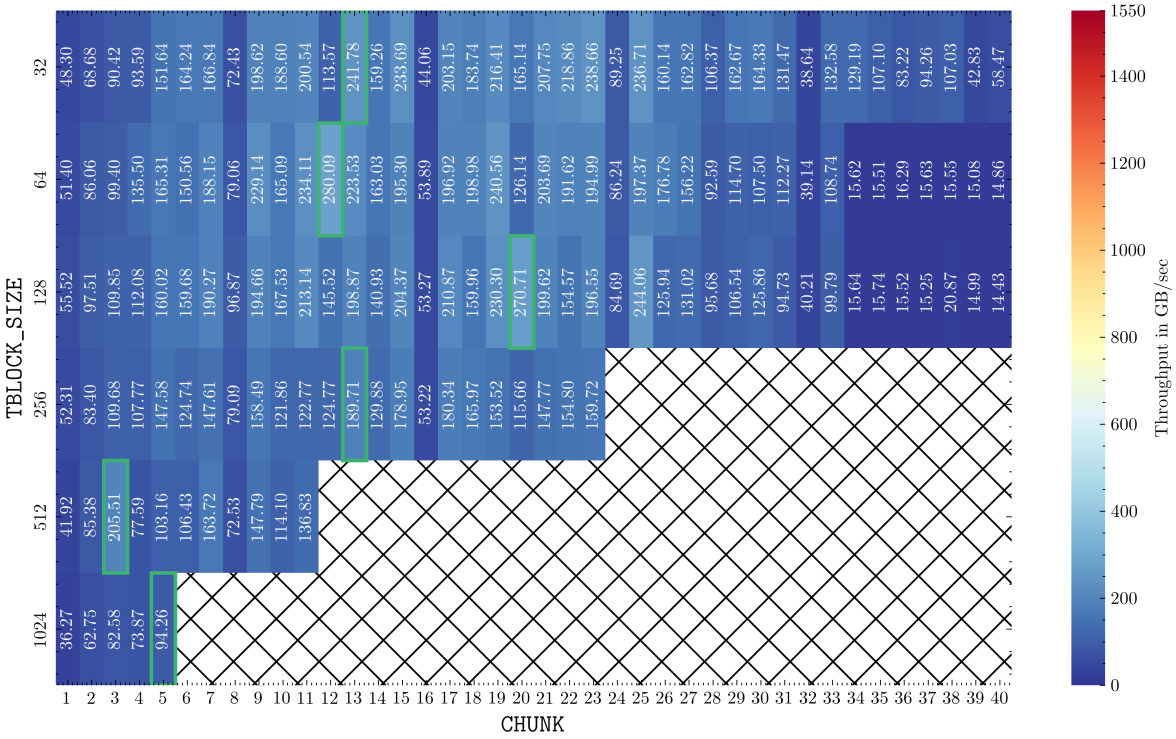

Figure 46: A100, `lssp.fut`, small dataset (∼3.0 GB)

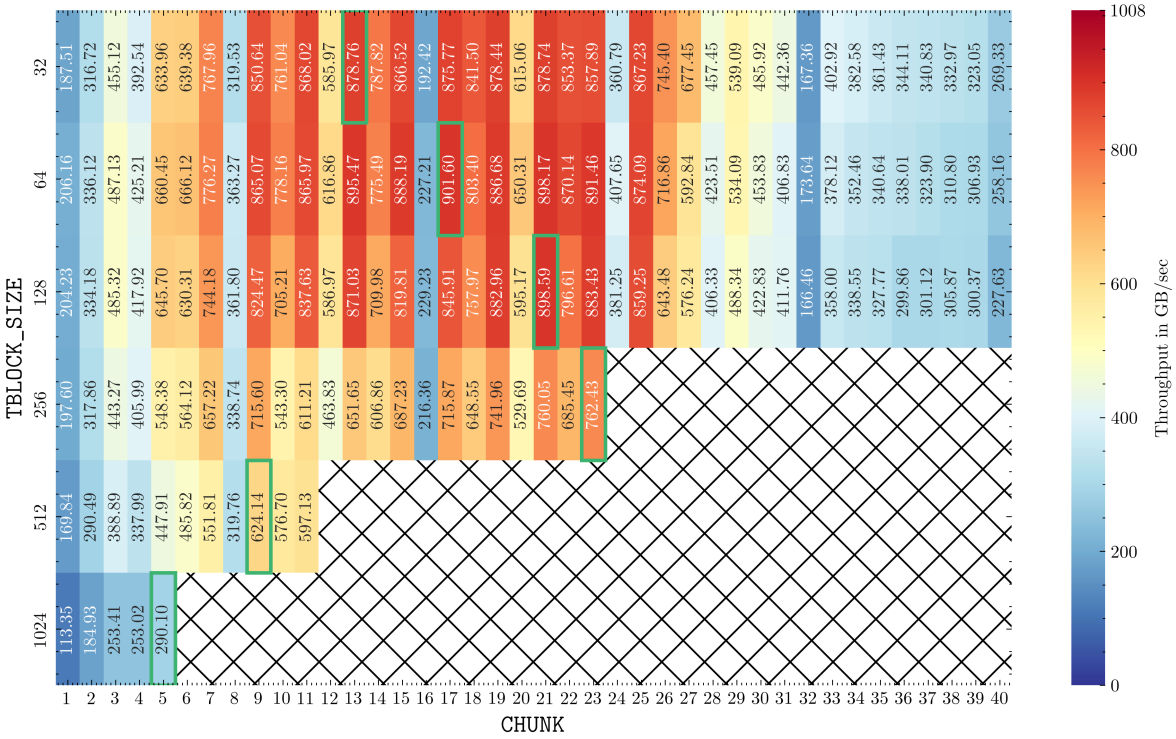

Figure 47: RTX4090, `lssp.fut`, small dataset (∼1.8 GB)