Automatic Parallelization and OpenMP Offloading of Fortran Array Notation

Ivan R. Ivanov[1,2] [0000−0003−0356−3768],
Jens Domke[2] [0000−0002−5343−414X],
Toshio Endo[1] [0000−0001−7297−6211], and
Johannes Doerfert[3] [0000−0001−7870−8963]

[1] Tokyo Institute of Technology
[2] RIKEN Center for Computational Science
[3] Lawrence Livermore National Laboratory

Abstract. The Fortran programming language is prevalent in the scientific computing community with a wealth of existing software written in it. It is still being developed with the latest standard released in 2023. However, due to its long history, many old code bases are in need of modernization for new HPC systems. One advantage Fortran has over C and C++, which are other languages broadly used in scientific computing, is the easy syntax for manipulating entire arrays or subarrays. However, this feature is underused as there was no way of offloading them to accelerators and support for parallelization has been unsatisfactory. The new OpenMP 6.0 standard introduces the `workdistribute` directive which enables parallelization and/or offloading automatically by just annotating the region the programmer wishes to speed up. We implement `workdistribute` in the LLVM project's Fortran compiler, called Flang. Flang uses MLIR as its intermediate representation which allows for a structured representation that captures the high level semantics of array manipulation and OpenMP. This allows us to build an implementation that performs on par with more verbose manually parallelized OpenMP code. By offloading linear algebra operations to vendor libraries, we also enable software developers to easily unlock the full potential of their hardware without needing to write verbose, vendor-specific source code.

Keywords: OpenMP · Fortran · offloading · parallelization.

1 Introduction

The most substantial compute power found in most modern HPC systems is in their accelerators, namely GPUs [1]. Thus, it is important to achieve a high GPU utilization in order to maximize performance of scientific computing applications.

Fortran is still prevalent in the scientific community and there are vast amounts of important applications written in it. Previous Fortran codes were not written with accelerators in mind, so enabling scientists to easily make use of modern hardware with minimal effort is an important goal.

OpenMP has been widely used as a way to accelerate these programs, and the OpenMP 6.0 Specification [2]—scheduled to be released in late 2024—introduces a new directive with this goal in mind, called `workdistribute`.

In this work, we present a proof-of-concept implementation of `workdistribute` in LLVM's MLIR-based Flang compiler, and show how code, which utilises array notation, can be automatically parallelized and offloaded to accelerators, with just simple annotations by the programmer. We will introduce the Flang compiler and the MLIR infrastructure it is built upon in Section 2. Then, we will present our approach in Section 4 and evaluate it in Section 5.

2 Background

In this section, we will outline some Fortran features which pertain to this work before introducing the new `workdistribute` OpenMP 6.0 directive. We will also briefly describe the LLVM project's MLIR compiler infrastructure and Fortran compiler Flang.

2.1 Array Notation in Fortran

Array notation provides easy syntax for the programmer to interact with multi-dimensional arrays in a concise and intuitive fashion, as shown in Fig. 1 **(a)**. With the new `workdistribute` construct, parallelization and offloading of such array notation requires only minimal source changes, as illustrated in Fig. 1 **(b)**. Programmers can also use the *slicing* notation (e.g., `y(1:n/2,1:n)`) to specify specific portions of the arrays to be operated on.

An intricacy of this feature is that the expression on the right-hand side (RHS) of assignments must be evaluated before being assigned to the left-hand side (LHS). An example of why this is required is shown in Fig. 2 where an array notation implementation of array reversal is shown in Fig. 2c. If the result of the RHS is computed element-wise and stored directly in the LHS as shown in Fig. 2b, some of the element-wise computation will not use the original values in

```
integer :: n
logical :: any_less
real, dimension(n, n) :: x, y, tmp

y(1:n/2,1:n) = 1.0
y = y + x
tmp = n * matmul(x, y + 1.0)
any_less = any(tmp(1:n/2,1:n/3) < 1.0)
```

(a) Fortran array notation

```
integer :: n
logical :: any_less
real, dimension(n, n) :: x, y, tmp
!$omp target teams workdistribute
    y(1:n/2,1:n) = 1.0
    y = y + x
    tmp = n * matmul(x, y + 1.0)
    any_less = any(tmp(1:n/2,1:n/3) < 1.0)
!$omp end target teams workdistribute
```

(b) OpenMP `workdistribute`

Fig. 1: **(a)** Fortan array notation allows operating on entire arrays or slices of arrays (discussion in Section 2.1) and **(b)** OpenMP `workdistribute` directive instructs the compiler to automatically parallelize the computation and optionally offload it to a target device (further discussed in Section 2.2)

```
integer :: n
real, dimension(n) :: x

x = x(n:1)
```

(a) Array reversal (original code)

```
integer :: i, n
real, dimension(n) :: x

do i = 1, n
  x(i) = x(n + 1 - i)
enddo
```

(b) Incorrect compiler-generated code

```
integer :: n
real, dimension(n) :: x, tmp

do i = 1, n
  tmp(i) = x(n + 1 - i)
enddo
do i = 1, n
  x(i) = tmp(i)
enddo
```

(c) Correct compiler-generated code

Fig. 2: Using array notation in Fortran may require additional implicit allocations as storing the result of the RHS directly in the LHS when they alias would overwrite other elements that may be used later in the same array operation.

`x`, but values overwritten earlier by the same array operation. Thus, correctly generated code for Fig. 2a is shown in Fig. 2c, where we allocate a temporary array for the RHS expression before assigning it to the LHS.

In general, in order to preserve correctness, the compiler must allocate intermediate temporary arrays for all expressions that appear in the RHS. For best performance, optimizations then try to eliminate temporaries when they are not required for correctness.

2.2 The OpenMP `workdistribute` directive

An example of the `workdistribute` directive is shown in Fig. 1 **(b)**. This directive must be nested directly in a `teams` directive which can in turn be nested in a `target` directive. The `target` directive specifies that the code can be run on a target device, while the `teams` directive indicates that a *league* of teams will be launched and they will work in parallel. The `teams` directive corresponds to the outermost level of parallelism in OpenMP.The `workdistribute` directive specifies that all teams share the work contained in it, while preserving the semantics of the Fortran code. This means that, for example, the ordering of statements is enforced and the RHS of assignments must be completed prior to the assignment to the LHS, as we discussed in Section 2.1.

Statements allowed inside the `workdistribute` region are array assignments, calls to array elemental operations (e.g. element-wise multiplication, math functions), calls to intrinsic functions operating on arrays (e.g. `matmul`, `transpose`, `any`, etc.), scalar operations and assignments. This means that this construct is a single block without control flow [2].

Each array element in an elemental-wise computation and assignment is a unit of work, as is each individual scalar operation. Parallelization is performed across units of work and compilers are at liberty to choose how intrinsics are implemented.

2.3 Multi-Level Intermediate Representation (MLIR)

A sub-project of the LLVM project, MLIR [3] is a compiler development framework that enables a structured intermediate representation that is easily composable and extensible. It allows abstractions on different levels to be freely mixed and contained in a single representation. The basic building blocks of MLIR are *operations* and structure in MLIR is represented through region-carrying operations. For example, an `if` statement can be represented as an operation that takes a condition as an operand and contains two regions, `then` and `else`, one of which is executed according to the condition value at runtime. A group of operations and types that pertain to a specific purpose is called a *dialect*.

2.4 The Flang Compiler

Flang [4] is LLVM's Fortran compiler and it uses MLIR for its intermediate representation. This allows it to use a progressive lowering pipeline which starts at an abstract high level representation that captures the Fortran semantics and then progressively lowers that to a more concrete compute-oriented representation in LLVM IR.

Flang's optimization pipeline is generally comprised of three stages which are illustrated in Fig. 3. They roughly correspond to the dialects used at the respective stages. At the beginning, the `hlfir` (High-Level Fortran IR) dialect is used in conjunction with `fir` (Fortran IR) and `omp` (OpenMP) dialects to express a high-level abstract representation of the program.[4]

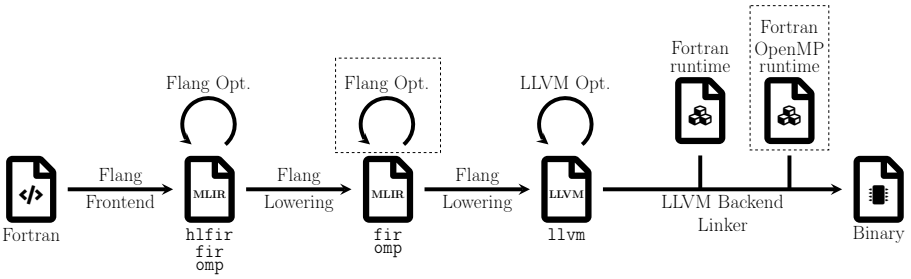


Fig. 3: Flang employs a gradual lowering strategy to preserve high-level Fortran semantics (in the `hlfir` dialect) before lowering to the `fir` dialect and finally LLVM IR. Optimizations suitable for the representation are performed at each step. The Fortran runtime provides the implementation of the array intrinsics. Most of the transformations described in this work are performed at the `fir` level and execution is supported by our OpenMP-enabled Fortran runtime, both of which are highlighted in the figure.

[4] Note that these are not the only dialects in use, however, they are the ones that characterize the compilation stage and its associated optimizations.

`hlfir` represents the array notation computation and array intrinsics in a way that allows high-level simplifications, for example merging multiple elemental computations, or simplifying sequences of intrinsics. `fir` is a lower level dialect that directly represents the computation while still preserving higher-level notions than traditional IRs such as loops and `if` statements and Fortran semantics for pointers and operations. This is Finally lowered to LLVM IR, where a standard optimization pipeline is run.

Fortran array intrinsics such as `matmul` and `transpose` are implemented as calls to a runtime library provided by Flang.

Flang's MLIR representation allows OpenMP constructs to be represented hierarchically. This hierarchical and gradual-lowering pipeline allows us to implement the necessary transformations for performant offloading.

3 Related Work

Automatic GPU offloading has been explored in many applications and programming languages. For example, it is a standard feature in many machine learning frameworks in high-level languages. However, these still require the programmer to adopt the framework's programming style and APIs.[5, 6] All arrays to be operated on also need to be converted to the types provided by the frameworks. This is also the case with NVIDIA's thrust [7], which provides high level C++ types to operate on arrays.

In C/C++ and Fortran, which are the most common languages for high performance scientific computing, accelerated libraries for similar patterns of computations that `workdistribute` allows exist [8, 9, 10], however, they still require the programmer to make extensive changes to their code.

High Performance Fortran, or HPF [11], is a Fortran extension that emerged in the 90's which supports automatic parallization and distributing of computation with annotations to native syntax. However, while it was met with enthusiasm, it failed to achieve success and wide adoption. One of the reasons for this was immature compiler technology and slow development speed at the time [12]. Conversely, recently, auto-parallelization and offloading of Fortran code was implemented for stencil style computation in Flang [13], showing the potential of the flexible intermediate representation.

Similarly to HPF, `workdistribute` allows the programmer to use the existing language syntax and only annotate what part of the code they want accelerated. In addition, because it is a part of the OpenMP standard, it allows easy integration into existing OpenMP code and/or extending the code with more generic computation than what `workdistribute` allows.

4 Automatic Parallelization and Offloading

In this section, we will present our approach and outline how owing to the structured high level representation employed by Flang, we are able to produce

a high performance implementation advantageous in comparison to traditional compilation approaches found elsewhere, for example, in clang.

4.1 Shortcomings of a Trivial Implementation

In order to preserve the Fortran semantics (Section 2.1), a trivial implementation would perform the following for each statement:

- Allocate temporary arrays for each expression in the RHS
- Execute a separate kernel for each expression in the RHS
- Copy the result of the RHS to the LHS.
- Deallocate the temporary arrays

Room for improvement exists w.r.t. unnecessary memory movement and allocations, cross-kernel simplifications (e.g. fusion), and using high-performance, vendor-provided libraries (e.g. cublas or rocblas for GPUs).

4.2 Overview of Our Approach

The statements that can be included in a `workdistribute` region (ref. Section 2.2) are be modelled by `hlfir` and optimizations on the `hlfir` level have an opportunity to greatly impact the performance of the resulting code. Thus, we would like to perform the division into units of work described above only after we have performed high-level optimizations in `hlfir`.

We achieve this by preserving the high-level structure of the `workdistribute` block and deferring materializing the separate kernels until a later stage. More specifically, we perform handling of the `workdistribute` directive at the `fir` stage of the Flang pipeline, see Section 2.4 and Fig. 3 for details.

4.3 MLIR `workdistribute` Operation

In order to make use of the `hlfir` optimizations, we need to preserve the information about what part of the program comes from the `workdistribute` directive through the `hlfir` pipeline. We know that `workdistribute` is comprised of a single Fortran block which contains no control flow (ref. Section 2.2). We introduce a new `workdistribute` MLIR operation which is a container for a single MLIR block. This block does not allow code to move outside it and allows us to precisely encapsulate the extent of our `workdistribute`.

We adapt the existing frontend code generation to emit the contents of the `workdistribute` Fortran construct in this operation. The MLIR we get at the beginning of the Flang MLIR pipeline is sketched in Fig. 4a.

Then, we reuse the existing high-level optimizations to optimize the contents of the `workdistribute` and then bufferize `hlfir`. This will already give us many of the optimizations that we want (e.g., buffer elimination, kernel merging). This leaves us in the state shown in Fig. 4b.

```
omp.target {
  omp.teams {
    omp.workdistribute {
      %a = hlfir.elemental {...}
      %b = hlfir.matmul %a ...
      hlfir.assign %b %c
      %d = hlfir.elemental {...}
      hlfir.assign %e %d
    }
  }
}
```

(a) The `omp` and `hlfir` dialects emitted by the frontend capture the high-level semantics and structure.

```
target {
  teams {
    workdistribute {
      fir.do_loop ... unordered {
        ...
      }
      fir.allocmem ...
      call RT_Matmul(...)
      fir.do_loop ... unordered {
      }
      call RT_Assign(...)
      fir.freemem ...
    }
  }
}
```

(b) Lowering to `fir` materializes loops and allocations, and adds runtime calls.

```
target {
  teams { workdistribute {
      fir.do_loop ... unordered {
        ...
      }
  }}
  teams { workdistribute {
      fir.allocmem ...
      call RT_Matmul(...)
  }}
  teams { workdistribute {
      fir.do_loop ... unordered {
        ...
      }
  }}
  teams { workdistribute {
      call RT_Assign(...)
      fir.freemem ...
  }}
}
```

(c) We fission the `teams(wd)` nests to outline separate parallel regions.

```
target {
  teams { distribute {
    parallel { wsloop {...} }
  }}
  fir.allocmem ...
  call RT_Matmul(...)
  teams { distribute {
    parallel { wsloop {...} }
  }}
  call RT_Assign(...)
  fir.freemem ...
}}
```

(d) We convert each `teams(wd)` nest to its corresponding OpenMP construct.

```
target_data {
  target { teams { distribute {
    parallel { wsloop {...} }
  }}}
  fir.allocmem ...
  call RT_OMP_Matmul(...)
  target { teams { distribute {
    parallel { wsloop {...} }
  }}}
  call RT_OMP_Assign(...)
  fir.freemem ...
}
```

(e) The `target` operation is fissioned to allow for host code execution and gets wrapped in a `target_data` to preserve the overall memory movement. Note that code in the `target_data` is executed on the host. The Fortran runtime calls get replaced with OpenMP enabled versions called from the host.

Fig. 4: Our transformation pipeline takes as an input a `omp.workdistribute` nested in `omp.teams` and optionally `omp.target`, and represents it in terms of concrete parallel OpenMP operations. We omit dialect names (e.g. `omp`) and shorten `workdistribute` to `wd` for brevity.

4.4 Lowering `workdistribute` to existing OpenMP Constructs

To lower the single-block `workdistribute` operation to concrete parallel OpenMP constructs we chunk the computation into appropriate kernels. Because we want to be able to replace intrinsic calls such as `matmul` and temporary memory allocations with appropriate runtime calls from the host, we need to split the target region and execute host code in-between. Splitting the target kernels is also required to synchronize across teams.

Then, we fission `workdistribute` into what will eventually become different `target` regions (i.e. kernels) or will turn into host-side runtime calls (Fig. 4c).

Now, we need to transform the `teams{workdistribute{...}}` nests. A `workdistribute loop` nest can be converted to a `distribute parallel do` nest, whereas a `workdistribute{<intrinsic>}` nest becomes just `<intrinsic>` (as the sharing of work happens *inside* the intrinsic) (Fig. 4d).

This is semantically sound, as it describes the computation that needs to happen on the target, however, it is invalid OpenMP as `teams` must be strictly nested in `target`. It also cannot be directly lowered as we do not have implementations of the intrinsics we can use on the device.

Thus, we need to fission the target region around our `teams` and transfer the rest of the computation to the host, where we can call our intrinsic functions, which will in turn perform the computation on the target device. The result of this transformation is shown in Fig. 4e. With this, we have successfully converted a `workdistribute` statement to existing OpenMP constructs which can be lowered to LLVMIR.

4.5 Enabling `hlfir` Optimizations: Alias Analysis

As we discussed in Section 2.2 and Section 4.1, implementing `workdistribute` naively results in excessive allocation of intermediate expressions and memory copying. This is especially problematic in the context of `workdistribute` as we are operating on entire arrays. In order to minimize the required intermediates, alias analysis is required to proof the correctness of omitting them. In our case, however, temporaries may be allocated on the target, while other arrays may be passed into the target kernel from the host. Thus, we need to be able to reason about arrays that cross the host-target boundary.

We extend Flang's alias analysis to follow memory references across host-target boundaries. This allows the existing Flang `hlfir` lowering pipeline to avoid unnecessary temporary allocations (see Section 2.1).

4.6 OpenMP-Enabled Fortran Runtime

Fortran array intrinsics such as `matmul` and `transpose` are implemented as runtime calls to a library provided by Flang (shown in Fig. 3). We provide OpenMP enabled versions of the runtime functions that take as arguments arrays that are *already* on the target.

For intrinsics that have high performance implementations by the vendor, such as `matmul` which is provided by cublas and rocblas on NVIDIA and AMD GPUs, we internally defer to them. This way, we abstract the low-level detail and boilerplate associated with using them directly and enable programmers to use the native language.

4.7 Memory Movement

Memory movement for OpenMP target offloading in Fortran can be automatically generated as the array types contain information about their bounds. This is in contrast with C/C++ where the sizes of the arrays must be specified by the programmer. When the automatically generated memory movement is not performant enough, it can be further optimized using the standard OpenMP memory movement utilities by the programmer due to `workdistribute` being part of the OpenMP infrastructure

5 Evaluation and Discussion

We conduct the evaluation of our approach using two experiments. Firstly, in Section 5.2, we compare array notation `workdistribute` against OpenMP offloading code that uses loops to express array operations to make sure our approach is up to par with idiomatic OpenMP and can benefit from vendor accelerated libraries. Secondly, in Section 5.3, we evaluate how our approach compares against the straight-forward, trivial implementation discussed in Section 4.1.

5.1 Experiment Setup and Benchmarks

Setup We use a dual-socket system with 8-core 16-thread Xeon Silver 4215 CPUs and an AMD MI210 GPU. We run all benchmarks a total of 3 times, each with an additional single warm up iteration and take the mean runtime. For GPU evaluations we measure the memory movement time between host and device, and computation time and report both separately.

Benchmarks For our proof-of-concept, we focus on two benchmarks shown in Fig. 5a and Fig. 5b. The first one is the `axpy` linear algebra operation which is a scaled vector addition, and the second one is a matrix multiplication. These are memory and compute bound, respectively.

5.2 Comparison to Loop-Based OpenMP code

For each benchmark, we compare the array notation implementation on a single CPU core (labeled: `cpu`) and a simple straight-forward loop-based OpenMP offloading implementation (`omp-traditional`) against an implementation utilizing `workdistribute` (`omp-workdistribute`). Note how the required programmer effort for the `workdistribute` implementation is very low, involving just wrapping the array notation block in OpenMP directives. Our results are shown in Fig. 6.

```
integer :: n
real :: a
real, dimension(n, n) :: x, y, z

z = a * x + y
```

cpu

```
!$omp target teams distribute \
!     parallel do collapse(2)
do i = 1, n
  do j = 1, n
    z = a * x(j, i) + y(i, j)
  enddo
enddo
```

omp-traditional

```
!$omp target teams workdistribute
  z = a * x + y
!$omp end target teams workdistribute
```

omp-workdistribute

(a) AXPY ($n = 4 \times 10^8$)

```
integer :: n
real, dimension(n, n) :: x, y, z

z = matmul(x, y)
```

cpu

```
!$omp target teams distribute \
!     parallel do collapse(2)
do i = 1, n
  do j = 1, n
    z(j, i) = 0
    do k = 1, n
      z(j, i) = z(j, i) +
                x(j, k) * y(k, i)
    enddo
  enddo
enddo
```

omp-traditional

```
!$omp target teams workdistribute
  z = matmul(x, y)
!$omp end target teams workdistribute
```

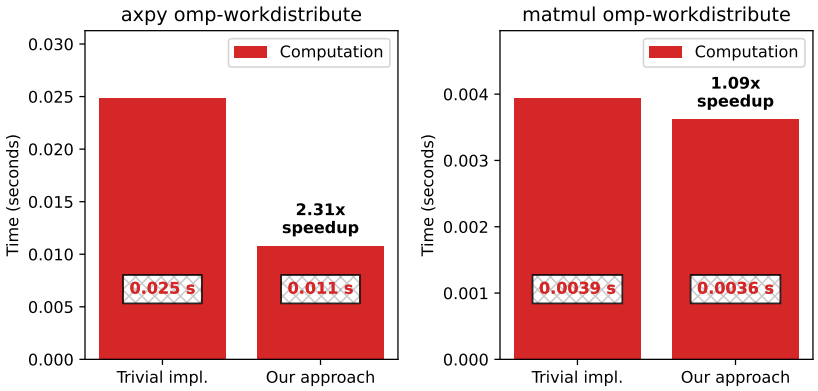
omp-workdistribute

(b) Matrix multiplication ($n = 4096$)

Fig. 5: We use three implementations of two common BLAS routines, *AXPY* and *Matrix multiplication*, to evaluate our approach. An array notation implementation which runs on a single core of the cpu for reference (**cpu**), a trivial implementation using traditional OpenMP offloading constructs (**omp-traditional**), and the simple array notation version, wrapped in `workdistribute` (**omp-workdistribute**). Note how `workdistribute` allows the programmer to avoid the verbosity of loop-based OpenMP code.

		all	computation	memory
matmul	cpu	61.6	61.6	0.00
	omp-traditional	0.440	0.312	0.128
	omp-workdistribute	0.131	0.00352	0.128
axpy	cpu	0.811	0.811	0.00
	omp-traditional	3.25	0.0110	3.24
	omp-workdistribute	3.25	0.0108	3.24

Fig. 6: We (**omp-workdistribute**) achieve superior or comparable performance compared to traditional OpenMP loop-based implementation (**omp-traditional**) on two common linear algebra operations (Fig. 5). The automatically parallelized and offloaded array-notation code (**omp-workdistribute)** is overwhelmingly more performant than the original array notation code executed on a CPU (**cpu**).



(a) `axpy` ($n = 4 \times 10^8$) (b) Matrix multiplication ($n = 4096$)

Fig. 7: A trivial implementation that needs to allocate temporary arrays for each expression and assignment (see Section 4.1) performs significantly worse than our approach which enables high-level optimizations. See Fig. 5 for the benchmarks.

We can see how the computation portion of the benchmark is on-par with the traditional OpenMP code in the `axpy` case, because the generated code is essentially the same since we are able to omit all of the redundant allocations and fuse the separate kernels. On the other hand, in the matrix multiplication case, we are able to use vendor libraries (here *rocblas* to target our AMD GPU) which boosts the performance by close to 8× while making it easier to write. Using vendor-provided linear algebra routines is cumbersome, verbose, and non-portable. On the other hand, using `workdistribute` hides the implementation details and uses abstractions available in the base language.

We can observe that on memory bound applications (e.g. `axpy`), straight-forward usage of `workdistribute` will not improve performance due to the overwhelming overhead of the memory movement between the host and the device. However, as we discussed in Section 4.7, this can be further optimized using the existing OpenMP infrastructure and the arrays can be kept on the device across multiple target regions if the applications allows it. On the other hand, as we can see from the matrix multiplication results, for compute-heavy tasks that accelerators excel at, even the straight-forward unoptimized host-device memory movement can improve performance at close to no programmer effort.

5.3 Comparison to a Trivial Implementation

In order to evaluate the benefits of our approach, we disable the optimizations related to removal of temporary allocations (see Section 2.1 and Section 4.5), which results in generated code resembling that of a trivial implementation discussed in Section 4.1.

We plot the results in Fig. 7. We can see how eliminating unnecessary temporary memory allocations is especially important in memory-bound computation, cf. Fig. 7a, where we achieve a 2.3× speedup over the trivial implementation. This stems from the code in Fig. 5, i.e., we need to allocate one temporary array `tmp1` for the result of `a * x` which we compute in one kernel, then we need another temporary allocation for `tmp1 + y` which we compute in another kernel, and finally a assignment to `z` which is a redundant memory copy. Using our approach, this gets reduced to a single kernel which computes `a * x(i,j) + y(i,j)` and stores the result directly in `z(i,j)`.

For the matrix multiplication benchmark (cf. Fig. 7b), the trivial implementation still uses the vendor libraries. The relatively small speedup we get is because we eliminate the redundant allocation for the RHS and assignment to LHS and instead store directly into it.

6 Conclusion

We presented a proof-of-concept implementation of the `workdistribute` OpenMP construct which provides an easy way to automatically parallelize and offload Fortran array notation. Our approach is implemented on top of LLVM's new Fortran compiler – Flang.[5] By employing a progressive abstraction lowering strategy enabled by the MLIR intermediate representation Flang uses, we are able to perform high-level optimizations to ensure high-performance code generation. This allows us to achieve over 2× speedup on a benchmark compared to code generated by a trivial implementation.

Hiding the parallelization details also allows us to use high-performance vendor libraries to accelerate array operations without programmers needing to manually call into cumbersome APIs.

This work can enable easy GPU offloading for existing Fortran codebases and improve utilization of modern HPC systems.

Acknowledgements

This work was supported by JST SPRING, Grant Number JPMJSP2106 and the RIKEN Junior Research Associate Program.

References

[1] TOP500. *June 2024*. URL: `https://www.top500.org/lists/top500/2024/06/` (visited on 06/21/2024).

[2] OpenMP Architecture Review Board. *OpenMP ARB Releases Technical Report 12*. URL: `https://www.openmp.org/press-release/openmp-arb-releases-technical-report-12/` (visited on 01/25/2024).

[5] The source is made available at `https://github.com/ivanradanov/llvm-project/tree/flang_workdistribute_iwomp_2024`.

[3] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[4] LLVM. *The Flang Compiler*. URL: https://flang.llvm.org/docs/ (visited on 06/20/2026).

[5] 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". In: *Advances in neural information processing systems* 32 (2019).

[6] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling machine learning programs via high-level tracing". In: *Systems for Machine Learning* 4.9 (2018).

[7] Nathan Bell and Jared Hoberock. "Thrust: A productivity-oriented library for CUDA". In: *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.

[8] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. "An updated set of basic linear algebra subprograms (BLAS)". In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[9] NI V'yukova, VA Galatenko, and SV Samborskii. "Support for parallel and concurrent programming in C++". In: *Programming and Computer Software* 44 (2018), pp. 35–42.

[10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[11] Harvey Richardson. "High Performance Fortran: history, overview and current developments". In: *Thinking Machines Corporation* 14 (1996), p. 17.

[12] Ken Kennedy, Charles Koelbel, and Hans Zima. "The rise and fall of High Performance Fortran: an historical object lesson". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 7–1–7–22. ISBN: 9781595937667. DOI: 10.1145/1238844.1238851. URL: https://doi.org/10.1145/1238844.1238851.

[13] Nick Brown, Maurice Jamieson, Anton Lydike, Emilien Bauer, and Tobias Grosser. "Fortran performance optimisation and auto-parallelisation by leveraging MLIR-based domain specific abstractions in Flang". In: *Pro-*

ceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23. , Denver, CO, USA, Association for Computing Machinery, 2023, pp. 904–913. ISBN: 9798400707858. DOI: 10.1145/3624062.3624167. URL: https://doi.org/10.1145/3624062.3624167.