Rank-Polymorphism for Shape-Guided Blocking

Rank-Polymorphism for Shape-Guided Blocking

Artjoms Šinkarovs
A.Sinkarovs@hw.ac.uk
Heriot-Watt University
Edinburgh, Scotland

Thomas Koopman
Thomas.Koopman@ru.nl
Radboud University
Nijmegen, Netherlands

Sven-Bodo Scholz
SvenBodo.Scholz@ru.nl
Radboud University
Nijmegen, Netherlands

Abstract

Many numerical algorithms on matrices or tensors can be formulated in a blocking style which improves performance due to better cache locality. In imperative languages, blocking is achieved by introducing additional layers of loops in a nested fashion alongside with suitable adjustments in index computations. While this process is tedious and error-prone, it is also difficult to implement a *generically* blocked version that would support arbitrary levels of blocking.

At the example of matrix multiply, this paper demonstrates how rank-polymorphic array languages enable the specification of such generically blocked algorithms in a simple, recursive form. The depth of the blocking as well as blocking factors can be encoded in the structure of array shapes. In turn, reshaping arrays makes it possible to switch between blocked and non-blocked arrays. Through rank-polymorphic array combinators, any specification of loop boundaries or explicit index computations can be avoided.

Firstly, we propose a dependently-typed framework for rank-polymorphic arrays. We use it to demonstrate that all blocked algorithms can be naturally derived by induction on the argument shapes. Our framework guarantees lack of out-of-bound indexing, and we also prove that all the blocked versions compute the same results as the canonical algorithm. Secondly, we translate our specification to the array language SaC. Not only do we show that we achieve similar conciseness in the implementation, but we also observe good performance of the generated code. We achieve a 7% improvement compared to the highly-optimised OpenBLAS library, and 3% compared to Intel's MKL library when running on a 32-core shared-memory system.

CCS Concepts: • **Theory of computation** → **Data structures design and analysis**; **Algorithm design techniques**; • **Software and its engineering** → **Source code generation**; *Functional languages*; **Data types and structures**.

FHPNC '23, September 4, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0296-9/23/09.
https://doi.org/10.1145/3609024.3609410

Keywords: Array Programming, Rank-Polymorphism, Dependent Types, HPC, Matrix Multiply

ACM Reference Format:
Artjoms Šinkarovs, Thomas Koopman, and Sven-Bodo Scholz. 2023. Rank-Polymorphism for Shape-Guided Blocking. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '23), September 4, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3609024.3609410

1 Introduction

Blocked implementations of numerical algorithms are commonplace in high performance computing. Blocking makes it possible to keep data that is necessary for the computation within fast(er) levels of the memory hierarchy, ameliorating the effect of memory bottlenecks. Typically, deriving a blocked version of a given algorithm is a non-trivial task. Especially in the context of imperative languages such as Fortran or C where one has to talk explicitly about nested loops, references to memory and index computations. Many numerical algorithms have gone through extensive analyses, leading to highly tuned codes containing several blocking variants alongside with inline assembly code to enable the highest possible levels of performance. Using such implementations through libraries such as OpenBLAS or MKL, while being convenient, may leave us feeling uneasy for mainly two reasons. Firstly, it is difficult to verify that such highly-tuned blocked versions *actually* implement the original algorithm. Secondly, any modification of the blocked implementation that introduces further levels of blocking or that change blocking factors are non-trivial and error-prone.

It turns out that the task of deriving blocked algorithms becomes significantly easier if we view this problem from within a rank-polymorphic array theory. By equipping arrays with a multi-dimensional indexing structure described through shapes, blocking algorithms appear almost automatically by induction over the shapes. Modifications of the blocking structure boils down to reshaping argument arrays prior to applying the algorithm. We find this result aesthetically pleasing, and with this paper, we take you to the journey through the envisioned framework where blocked algorithms can be defined, proven correct, and turned into code that runs efficiently on state-of-the-art parallel architectures.

We use a dependently-typed system to introduce the array theory, implement the algorithm and its blocked version using shape recursion, and we prove that these two versions compute the same results. We use Agda for the actual implementation, but any implementation of Martin-Löf type theory can be used instead. After that, a dependently-typed formulation is turned into a SaC program. SaC is an array language that admits rank-polymorphic architecture-agnostic specifications, yet its compiler is capable of generating highly optimised parallel code.

We use blocked matrix multiplication as a running example to demonstrate the process. Some stages are fully automated, others are performed manually. The latter indicate immediate future work. While not being anywhere near a "press-button" tool chain, we manage to match performance of hand-optimised state of the art libraries such as OpenBLAS or MKL from high-level specifications, without ever writing a single line of assembly.

The main contributions of this paper are:

- The observation that rank-polymorphism enables very elegant specifications of shape-guided blocking;
- The specification of blocked matrix multiplication as a shape-recursive program within a rank-polymorphic array theory;
- A formal proof that blocking preserves the semantics of the canonical algorithm;
- A shape-recursive blocked matrix multiplication in SaC;
- Performance analysis and tuning of the SaC version so that it matches performance of OpenBLAS and MKL.

2 Blocked Matrix Multiplication

Suppose A is an $M \times K$ matrix and B is an $K \times N$ matrix. Then the canonical definition of their product is given by the following formula:

$$(AB)_{i,j} = \sum_{p<K} A_{i,p} \cdot B_{p,j}.$$

The product AB represents the composition of the linear functions that A and B represent.

Now, if we can split A and B into blocks, we can define a blocked version of the same algorithm. Assuming that $M = \hat{M} b_M$, $K = \hat{K} b_K$ and $N = \hat{N} b_N$, we can subdivide A in submatrices of size $b_M \times b_N$, B into blocks of size $b_N \times b_K$ and AB into blocks of size $b_M \times b_K$. In this case, matrix multiplication is given as:

$$\begin{pmatrix} A_{0,0} & \cdots & A_{0,\hat{K}} \\ \vdots & \ddots & \vdots \\ A_{\hat{M},0} & \cdots & A_{\hat{M},\hat{K}} \end{pmatrix} \begin{pmatrix} B_{0,0} & \cdots & B_{0,\hat{K}} \\ \vdots & \ddots & \vdots \\ B_{\hat{K},0} & \cdots & B_{\hat{K},\hat{N}} \end{pmatrix} =$$

$$\begin{pmatrix} \sum_{p<\hat{K}} A_{0,p} B_{p,0} & \cdots & \sum_{p<\hat{K}} A_{0,p} B_{p,\hat{N}} \\ \vdots & \ddots & \vdots \\ \sum_{p<\hat{K}} A_{\hat{M},p} B_{p,0} & \cdots & \sum_{p<\hat{K}} A_{\hat{M},p} B_{p,\hat{N}} \end{pmatrix}$$

The interesting observation here is that we can iterate this process by applying blocking within the inner matrix multiplications. We can do this up to the moment we get to the matrices of size 1×1, in which we can stop and use a regular element product to compute the result. Putting these observations together we obtain a recursive algorithm where the base case is a regular product, and the recursive step is the blocked decomposition described above.

3 Discovering Blocked Algorithms

The initial point of our framework is a dependently-typed language. We use Agda, but none of the constructions presented below are Agda-specific, so any implementation of Martin-Löf type theory can be used instead. Our formalisation introduces a minimalist rank-polymorphic array theory, derives blocked matrix multiplication and proves its correctness. This section is written in literate Agda, so all the expressions have been typechecked. We assume some familiarity with Agda syntax, otherwise consider folllowing any of these[1] tutorials.

When working with dependent types it is important to chose the right encoding for the data that we are reasoning about. A well-chosen encoding can either make all further reasoning pleasant or turn it into a nightmare.

In our case, we have to chose representation for multi-dimensional arrays. We want to be able to reason about rank-polymorphic functions on these arrays and guarantee safe indexing. We are not interested in capturing low-level implementational details such as memory allocation or synchronisation of threads. For such cases, it is a common practice to represent arrays as functions. For the purposes of rank-polymorphism, we have to chose a convenient representation for array shapes and array indices.

Array shapes are sequences of natural numbers. While these can be represented as lists, we encode them as finite binary trees, where leaves are natural numbers. Such encoding makes it possible to describe non-trivial traversal of the shape sequence inductively. We call the data type for shapes S, and we define it inductively using two constructors.

```
data S : Set where
  ι     : ℕ → S
  _⊗_ : S → S → S
```

Leaves of the shape tree are constructed with ι which takes one argument. The _⊗_ constructor creates a tree out of two subtrees which are the arguments of the constructor. Note that underscores in _⊗_ specify the positions where the arguments go, therefore ⊗ is a binary infix operation.

Array indices (positions) are given by the dependent type P which is indexed by array shapes. A position for the given shape has exactly the same tree structure as the shape, but all

[1]The list of introductory tutorials to Agda can be found here: https://agda.readthedocs.io/en/v2.6.3/getting-started/tutorial-list.html

the leaves are natural numbers that are bound by the shape leaf. Again, P is given inductively using the following two constructors.

```
data P : S → Set where
  ι     : Fin n → P (ι n)
  _⊗_ : P s → P p → P (s ⊗ p)
```

The type of arrays is indexed by the element type and the shape. We use a C-like syntax with square brackets after the element type. Objects of this type are functions from array indices into array elements.

```
_⟦_⟧ : Set → S → Set
X ⟦ s ⟧ = P s → X
```

There is no need for special syntax for creating arrays, as this can be achieved by a lambda term. Neither there is a need for the syntax for array selections, as this can be achieved with function application. Yet, for aesthetic purposes, we add a C-like syntax for array selections:

```
_[_] : X ⟦ s ⟧ → (P s → X)
a [ ix ] = a ix
```

Summation. As a prerequisite for defining matrix multiplication, we have to define array summation. We define a 1-dimensional version of summation called sum_1. After that, we will generalise it to arbitrary shapes.

We start with a few helper operations: ιsuc increases the given 1-dimensional index by 1; hd is the head (first element of the 1-dimensional array); tl is the tail (the 1-dimensional array without the first element).

```
ιsuc : P (ι n) → P (ι (suc n))
ιsuc (ι i) = ι (suc i)

hd : X ⟦ ι (suc n) ⟧ → X
hd a = a [ ι zero ]

tl : X ⟦ ι (suc n) ⟧ → X ⟦ ι n ⟧
tl a i = a [ ιsuc i ]
```

With these operations in place, sum_1 is defined recursively. The function has two arguments: the binary function, the initial element and the 1-dimensional array that we sum. For the empty array we return the initial element, and for non-empty arrays we add the head of the array to the sum of the tail.

```
sum₁ : (X → X → X) → X → X ⟦ ι n ⟧ → X
sum₁ {n = zero}  f e a = e
sum₁ {n = suc n} f e a = f (hd a) (sum₁ f e (tl a))
```

Array Combinators. It is useful to invest a little time in defining common array combinators. As our arrays are functions, we can easily define: K x to produce an array where all the elements are x; map f a to apply f to all the elements of a; and zipWith f a b to apply the binary operation f to arrays a and b point-wise, given that a and b are of the same shape.

```
K : X → X ⟦ s ⟧
K x i = x

map : (X → Y) → X ⟦ s ⟧ → Y ⟦ s ⟧
map f a i = f (a [ i ])

zipWith : (X → Y → Z) → X ⟦ s ⟧ → Y ⟦ s ⟧ → Z ⟦ s ⟧
zipWith f a b i = f (a [ i ]) (b [ i ])
```

The nest and unnest combinators make it possible to treat the same array as a nested one (array of arrays); or as an array of a product shape. Note that both operations change the order of the shape components because our array notation (_⟦_⟧) is a postfix operation.

```
nest : X ⟦ s ⊗ p ⟧ → X ⟦ p ⟧ ⟦ s ⟧
nest a i j = a [ i ⊗ j ]

unnest : X ⟦ p ⟧ ⟦ s ⟧ → X ⟦ s ⊗ p ⟧
unnest a (i ⊗ j) = a [ i ] [ j ]
```

Generalised Sum. We define sum for arbitrarily shaped array by induction on the shape. For 1-d case, we use sum_1. For product shapes we apply sum to all the sub-arrays and sum all the partial results.

```
sum : (X → X → X) → X → X ⟦ s ⟧ → X
sum {s = ι x}     f e a = sum₁ f e a
sum {s = s ⊗ p} f e a = sum   f e $ map (sum f e) $ nest a
```

Note that this definition of sum traverses the array bottom-up, and it adds the neutral element after each 1-dimensional traversal. We do not impose any restrictions on the binary operation or the initial element in this definition.

Canonical Matrix Multiplication. Having the sum operation at hand, we define a canonical version of matrix multiplication. In order to keep matrix multiplication generic, we parametrise it by the types X, Y and Z and the operations (_⊠_ : $X \to Y \to Z$), (ϵ : Z) and (_⊞_ : $Z \to Z \to Z$) that correspond to multiplication, initial element and addition.

```
mm-canon : X ⟦ s ⊗ p ⟧ → Y ⟦ p ⊗ q ⟧ → Z ⟦ s ⊗ q ⟧
mm-canon a b (i ⊗ j) =
  sum _⊞_ ϵ λ k → a [ i ⊗ k ] ⊠ b [ k ⊗ j ]
```

For now, there is no assumptions about the properties of _⊠_, ϵ or _⊞_. Note that mm-canon is not restricted to two-dimensional arrays where the shape is a product of two singletons (*e.g.* ι m ⊗ ι n), it is rank-polymorphic after all. We come back to this question later in this section.

Blocking. The essence of blocked algorithms comes from the ability to "block" arrays into smaller sub-arrays. That is, we cut the matrix into sub-matrices that preserve all the local neighbours within the block. For example, consider an array of shape 4×6 blocked into an array of shape 2×2 where each element is a 2×3 block.

$$\begin{pmatrix} a & a & a & a & a & a \\ b & b & b & b & b & b \\ c & c & c & c & c & c \\ d & d & d & d & d & d \end{pmatrix} \Rightarrow \left(\begin{array}{ccc|ccc} a & a & a & a & a & a \\ b & b & b & b & b & b \\ \hline c & c & c & c & c & c \\ d & d & d & d & d & d \end{array}\right)$$

For our arrays, we can define conversion functions that perform blocking over rows and columns as presented above.

```
block : X ⟦ (s ⊗ p) ⊗ (q ⊗ r) ⟧ → X ⟦ (s ⊗ q) ⊗ (p ⊗ r) ⟧
block a ((i ⊗ j) ⊗ (k ⊗ l)) = a [ (i ⊗ k) ⊗ (j ⊗ l) ]

unblock : X ⟦ (s ⊗ q) ⊗ (p ⊗ r) ⟧ → X ⟦ (s ⊗ p) ⊗ (q ⊗ r) ⟧
unblock = block
```

Note that block is a self-inverse, but we will will define the unblock to indicate the intention.

Next, we can make a cut over the columns of the matrix. For the example above of shape 4×6, we get an array of shape 3 where the elements are blocks of shape 4×2

$$\begin{pmatrix} a & a & a & a & a & a \\ b & b & b & b & b & b \\ c & c & c & c & c & c \\ d & d & d & d & d & d \end{pmatrix} \Rightarrow \left(\begin{array}{cc|cc|cc} a & a & a & a & a & a \\ b & b & b & b & b & b \\ c & c & c & c & c & c \\ d & d & d & d & d & d \end{array}\right)$$

We call this operation block_v (vertical cuts), and similarly to block it is a self-inverse, but we still introduce unblock_v to indicate the intention.

```
block_v : X ⟦ s ⊗ (p ⊗ q) ⟧ → X ⟦ p ⊗ (s ⊗ q) ⟧
block_v a (i ⊗ (j ⊗ k)) = a [ j ⊗ (i ⊗ k) ]

unblock_v : X ⟦ p ⊗ (s ⊗ q) ⟧ → X ⟦ s ⊗ (p ⊗ q) ⟧
unblock_v = block_v
```

Finally, we can block over rows (horizontal cuts), and for our 4×6 example we have an array of 2 where elements are blocks of shape 2×6.

$$\begin{pmatrix} a & a & a & a & a & a \\ b & b & b & b & b & b \\ c & c & c & c & c & c \\ d & d & d & d & d & d \end{pmatrix} \Rightarrow \left(\begin{array}{cccccc} a & a & a & a & a & a \\ b & b & b & b & b & b \\ \hline c & c & c & c & c & c \\ d & d & d & d & d & d \end{array}\right)$$

In this case, the function is not a self-inverse as we have to rearrange the structure of the shape

```
block_h : X ⟦ (s ⊗ p) ⊗ q ⟧ → X ⟦ s ⊗ (p ⊗ q) ⟧
block_h a (i ⊗ (j ⊗ k)) = a [ (i ⊗ j) ⊗ k ]

unblock_h : X ⟦ s ⊗ (p ⊗ q) ⟧ → X ⟦ (s ⊗ p) ⊗ q ⟧
unblock_h a ((i ⊗ j) ⊗ k) = a [ i ⊗ (j ⊗ k) ]
```

Note that while our pictures are two- and four-dimensional, the blocking functions are rank-polymorphic. Free variables s, p, q and r can be of arbitrary ranks.

Blocked Matrix Multiplication. We are ready to define blocked matrix multiplication that we call mm and that has the following type for some shapes s, p and q:

```
mm : X ⟦ s ⊗ p ⟧ → Y ⟦ p ⊗ q ⟧ → Z ⟦ s ⊗ q ⟧
```

Now we define mm by pattern-match on the structure of s, p and q. This gives us $2 \cdot 2 \cdot 2 = 8$ variants that are given below. The base-case of our algorithm is the variant where s, p and q are all singletons, in which case we simply apply the canonical matrix multiplication.

```
mm {s = ι m} {ι n} {ι k} = mm-canon
```

In all the other cases, we block the input matrices in such a way that the shapes of the blocks are appropriate for the subsequent mm calls. Then we apply mm to these blocks and reduce partially-multiplied blocks using sum that is adjusted to add arbitrarily-shaped arrays. We use the colon (:) notation to annotate the type of the reblocked arrays before applying mm recursively. Finally, we have to unblock the result that we get from the recursive call.

```
mm {s = ι m} {ι n} {p ⊗ q} a b =
  let b' = nest $ block_v b             : Y ⟦ ι n ⊗ q ⟧ ⟦ p ⟧
      c' = map (mm a) b'                : Z ⟦ ι m ⊗ q ⟧ ⟦ p ⟧
  in  unblock_v $ unnest c'

mm {s = ι m} {p ⊗ q} {ι n} a b =
  let a' = nest $ block_v a             : X ⟦ ι m ⊗ q ⟧ ⟦ p ⟧
      b' = nest $ block_h b             : Y ⟦ q ⊗ ι n ⟧ ⟦ p ⟧
  in  sum (zipWith _⊞_) (K ϵ) λ k → mm (a' [ k ]) (b' [ k ])

mm {s = ι m} {p ⊗ q} {r ⊗ w} a b =
  let a' = nest $ block_v a             : X ⟦ ι m ⊗ q ⟧ ⟦ p ⟧
      b' = nest $ block b               : Y ⟦ q ⊗ w ⟧ ⟦ p ⊗ r ⟧
      c' = λ (i : P r) → sum (zipWith _⊞_) (K ϵ)
                  λ k → mm (a' [ k ]) (b' [ k ⊗ i ])
  in  unblock_v $ unnest c'


mm {s = s ⊗ p} {ι m} {ι n} a b =
  let a' = nest $ block_h a             : X ⟦ p ⊗ ι m ⟧ ⟦ s ⟧
      c' = map (flip mm b) a'           : Z ⟦ p ⊗ ι n ⟧ ⟦ s ⟧
  in  unblock_h $ unnest c'

mm {s = s ⊗ p} {ι m} {q ⊗ r} a b =
  let a' = nest $ block_h a             : X ⟦ p ⊗ ι m ⟧ ⟦ s ⟧
      b' = nest $ block_v b             : Y ⟦ ι m ⊗ r ⟧ ⟦ q ⟧
      c' = (λ { (i ⊗ j) → mm (a' [ i ])
                  (b' [ j ]) }) : Z ⟦ p ⊗ r ⟧ ⟦ s ⊗ q ⟧
  in  unblock $ unnest c'


mm {s = s ⊗ p} {q ⊗ r} {ι m} a b =
  let a' = nest $ block a               : X ⟦ p ⊗ r ⟧ ⟦ s ⊗ q ⟧
      b' = nest $ block_h b             : Y ⟦ r ⊗ ι m ⟧ ⟦ q ⟧
      c' = λ (i : P s) → sum (zipWith _⊞_) (K ϵ)
```

```
                                λ k → mm (a’ [ i ⊗ k ]) (b’ [ k ])
  in unblock_h $ unnest c’

mm {s = s ⊗ p} {q ⊗ r} {u ⊗ w} a b =
  let a’ = nest $ block a                : X ⟦ p ⊗ r ⟧ ⟦ s ⊗ q ⟧
      b’ = nest $ block b                : Y ⟦ r ⊗ w ⟧ ⟦ q ⊗ u ⟧
      c’ = λ { (i ⊗ j) → sum (zipWith _⊞_) (K ϵ)
                                λ k → mm (a’ [ i ⊗ k ])
                                         (b’ [ k ⊗ j ]) }
  in unblock $ unnest c’
```

It is satisfyingly surprising that all these cases appear naturally by following the structure of the argument shapes.

Correctness. While there is a strong evidence that blocked mm implements matrix multiplication, we would like to prove this formally. We introduce an equivalence relation on arrays, which is defined as point-wise equality of the array elements.

```
_≈_a_ : X ⟦ s ⟧ → X ⟦ s ⟧ → Set
a ≈_a b = ∀ i → a [ i ] ≡ b [ i ]
```

We can formulate and prove our main theorem called blocked-ok that says: for any two matrices a and b (of the matching shapes) mm a b computes the same array as mm-canon a b.

```
blocked-ok : (a : X ⟦ s ⊗ p ⟧) → (b : Y ⟦ p ⊗ q ⟧)
           → mm a b ≈_a mm-canon a b
```

For space purposes, we omit the proof itself, but it can be found in the sources of this paper [25]. The proof itself is straight-forward, as the representation of our arrays have very good computational properties. Term normalisation inlines most of the operations, and typically, all we have to do is to apply the following lemma:

```
sum-commutes : (a : Z ⟦ p ⟧ ⟦ s ⟧)
             → sum (zipWith _⊞_) (K ϵ) a
               ≈_a λ i → sum _⊞_ ϵ λ j → a [ j ] [ i ]
```

It says that for the nested array a, it does not matter whether we apply sum with the operation lifted by zipWith and the inital element lifted by K, or we apply the non-lifted sum to all the “columns” along the p shape. The proof of this lemma follows the definition of sum.

From this proof we see that blocking does not depend on the properties of the operations used in matrix multiplication. If the definition of the sum is fixed, blocking does not change the order in which the elements are multiplied and added.

Blocking Factors. The blocking behaviour of mm is fully determined by the shape structure of s, p and q. These sub-shapes encode the blocking factors, forcing us to use arrays of ranks that are larger than two. We can always obtain such arrays from regular matrices by choosing multipliers for rows and columns and organising these multipliers into sub-shapes. For example, the (rank 4) matrix of shape $(\iota\ m \otimes \iota\ n) \otimes (\iota\ k \otimes \iota\ l)$ is isomorphic to the (rank 2) matrix of shape $\iota\ (m * n) \otimes \iota\ (k * l)$. We can define conversion functions flatten-2d and unflatten-2d that make it possible to switch between the matrices.

```
flatten : S → ℕ
flatten (ι n)     = n
flatten (s ⊗ p) = flatten s * flatten p

ιflatten : S → S
ιflatten s = ι (flatten s)

pflatten : P s → P (ιflatten s)
punflatten : P (ιflatten s) → P s
```

The flatten function computes the product of all the leaves in the shape tree, performing multiplications in the order that is determined by the tree structure. Index translations pflatten and punflatten perform multiplication or division with remainder (bodies of these functions are omitted).

Array flattenings simply apply index translations contravariantly.

```
flatten-2d : X ⟦ s ⊗ p ⟧ → X ⟦ ιflatten s ⊗ ιflatten p ⟧
flatten-2d a (i ⊗ j) = a (punflatten i ⊗ punflatten j)

unflatten-2d : X ⟦ ιflatten s ⊗ ιflatten p ⟧ → X ⟦ s ⊗ p ⟧
unflatten-2d a (i ⊗ j) = a (pflatten i ⊗ pflatten j)
```

With these functions we can turn any matrix of rank 2, into the matrix of a higher rank by choosing blocking factors for its rows and columns. Then we can obtain blocking behaviour by applying mm on the matrix of a higher rank. We know from blocked-ok that we will obtain the same elements as if we were applying mm-canon that does not do any blocking, but it uses all the blocking factors at summation.

If we want to guarantee that blocked algorithm computes the same result as the canonical one applied to the rank 2 matrix, we have to guarantee that _⊞_ is associative and ϵ is the neutral element. Here is the key theorem that captures this fact. We omit the proof for space purposes, but it can be found in [25].

```
-- Assuming that ⊞ is associative, and ϵ is ⊞-neutral
mm-flat-ok : (a : X ⟦ s ⊗ p ⟧) (b : Y ⟦ p ⊗ q ⟧)
           → mm-canon (flatten-2d a) (flatten-2d b)
             ≈_a flatten-2d (mm-canon a b)
```

Pre-blocking. We can ask ourselves whether it is possible to separate the blocking steps from the actual recursive matrix multiplication. That is, would it be possible to pre-block the array, apply matrix multiplication recursively, and unblock the entire array back. To do so, we need to construct an evidence that the arrays are blocked in a such a way so that all the recursive calls of mm become applicable. In other words, we have to capture recursive unfolding of the mm for the given arrays.

To illustrate this, we define a relation that captures one particular blocking style — the last case of mm, where blocking happens on both the rows and the columns. Additionally, we chose the outer shape to always be of the form ($\iota\ m \otimes \iota\ n$) to prevent further decomposition on the outer side. We capture these assumptions in the inductive relation called R.

```
data R : S → S → S → Set where
  ι : R (ι m ⊗ ι n) (ι n ⊗ ι k) (ι m ⊗ ι k)
  σ : R s p q
    → R ((ι m ⊗ ι n) ⊗ s) ((ι n ⊗ ι k) ⊗ p) ((ι m ⊗ ι k) ⊗ q)
```

This relation describes all the valid shapes of the two input matrices and the output matrix, in case they were pre-blocked in the way we have described above. The base case ι says that any singletons shapes of the form $m \times n$ $n \times k$ and $n \times k$ satisfy our assumption. The σ case says that if s, p and q are related by R, then we can use matrices of the shape $m \times n$ $n \times k$ and $n \times k$, where the elements are of the shape s, p and q correspondingly.

In this case, if we have a witness for the inner structure of the argument shapes as described above, we can define the blocked matrix multiplication inductively as follows:

```
mmx : R s p q → X ⟦ s ⟧ → Y ⟦ p ⟧ → Z ⟦ q ⟧
mmx ι     a b = mm-canon a b
mmx (σ r) a b = let
  a' = nest a
  b' = nest b
  c' = λ { (i ⊗ j) → sum (zipWith _⊞_) (K ϵ)
                         λ k' → mmx r (a' [ i ⊗ k' ])
                                      (b' [ k' ⊗ j ]) }
  in unnest c'
```

Essentially, we assume that arrays a and b are pre-blocked (witnessed by the first argument of mmx) such that we always take the last case of mm in the non-pre-blocked scenario.

Pre-blocking correctness. In order to see that the pre-blocked version mmx and mm compute the same results, we have to define conversion functions between the original arrays and the pre-blocked ones.

For the given shapes s, p, and q that are related by R (*i.e.* for pre-blocked arrays), we can compute the product shapes s', p' and q' that will be suitable for mm. Also, we compute the conversion functions from s to $s' \otimes p'$, from p to $p' \otimes q'$, and from $s' \otimes q'$ into q.

```
toinv : ∀ s p q → R s p q
  → Σ[ (s' , p' , q') ∈ S × S × S ]
       ( (∀ {X} → X ⟦ s ⟧ → X ⟦ s' ⊗ p' ⟧)
         × (∀ {X} → X ⟦ p ⟧ → X ⟦ p' ⊗ q' ⟧)
         × (∀ {X} → X ⟦ s' ⊗ q' ⟧ → X ⟦ q ⟧))
toinv (ι m ⊗ ι n) (ι n ⊗ ι k) (ι m ⊗ ι k) ι =
  (ι m , ι n , ι k) , id , id , id
toinv (ι m ⊗ ι n ⊗ s) (ι n ⊗ ι k ⊗ p) (ι m ⊗ ι k ⊗ q) (σ r) = let
  (s' , p' , q') , s⇒s'p' , p⇒p'q' , s'q'⇒q = toinv s p q r
  in (ι m ⊗ s' , ι n ⊗ p' , ι k ⊗ q')
    , (λ a → unblock $ unnest $ map s⇒s'p' $ nest a)
    , (λ a → unblock $ unnest $ map p⇒p'q' $ nest a)
    , (λ a → unnest $ map s'q'⇒q $ nest $ block a)
```

With the given conversions available, we can show that switching between the pre-blocked version and the blocked one does not affect the computation.

```
mmx≈mm : (r : R s p q) (a : X ⟦ s ⟧) (b : Y ⟦ p ⟧)
       → let _ , fromA , fromB , toC = toinv s p q r
         in mmx r a b ≈ₐ toC (mm (fromA a) (fromB b))
```

You can find the proof in [25]. Technically, we have to compute the conversions in the other direction, but these conversions are actually isomorphisms. In order to show this formally we would have to extend our framework a little further. We do this in the proof, but not in the paper.

4 Shape Recursion in SaC

We now port this formally proved, blocked algorithm manually into the array language SaC, in hope that the optimising compiler `sac2c` will produce efficiently executable, parallel code. We assume some familiarity with SaC syntax, otherwise consider [20, 31] as an introduction to the language.

In SaC array shapes are represented as 1-dimensional arrays. While we can access all the components of the shape using selection, we cannot encode an arbitrary splitting pattern as we did in Agda specification — SaC shapes are flattened trees. However, as SaC makes it possible to overload functions based on shapes, it is easy to define regular shape-recursive traversals such as the one that consumes shapes sequence left-to-right or right-to-left. This is exactly what is needed to implement our running example.

4.1 Blocking

We replicate blocking across rows and columns, similar to `block`/`unblock` in the specification. To do so, we have to make an assumption about the structure of the array shape that we are dealing with. For matrices a and b of shapes $m \times n$ and $n \times k$ correspondingly, we assume that m, n, k can be split into equal number of multipliers. We illustrate this idea with three multipliers, but the number can be arbitrary:

$$xyz = m \qquad uvw = n \qquad spq = k$$

In this case we can reshape a into an array of shape $[x, y, z, u, v, w]$ and b into an array of shape $[x, y, z, u, v, w]$. Reshaping preserves all the elements and their order under row-major flattening. In SaC, such reshapes have zero overhead at runtime, as all the SaC arrays are flattened. These new shapes will be only used to guide the recursion.

New shapes always have the even number of elements. Blocking splits the shape "in the middle", picks the first element from each part, and moves them to the front. Unblocking will perform the same operations in reverse.

We can define such blocking/unblocking operations concisely, if we consider shapes having a multi-dimensional structure. The following combinators will help to view shapes as 2-dimensional arrays:

```
// [x,y,z,...,u,v,w,...] => [[x,u],[y,v],[z,w],...]
int[.,.] cut_half(int[.] s)
{
  return transpose(reshape([2, len(s)/2], s));
}

// [x,u, y,v, z,w,...] => [[x,y,z,...],[u,v,w,...]]
int[.,.] cut_pairs(int[.] s)
{
  return transpose(reshape([len(s)/2, 2], s));
}
```

As blocking/unblocking operations do not change the rank of the array, it only shuffles the shape components. We can define a helper operation that we call ptranspose that is similar to the dyadic transpose in APL:

```
int[*] ptranspose(int[.] pv, int[*] a)
{
  return { iv -> a[invpermute(pv, iv)]
         | iv < permute(pv, shape(a))};
}
```

The function takes the permutation vector of the shape of the second argument. As every permutation has an inverse, we can trivially compute the mapping between shapes and indices.

Our strategy in implementing blockings is to compute the permutation mask using 2d view of the array shape and pass this mask to ptranspose. The block operation cuts the shape "in the middle", makes a 2d array of two halves and splits on the first column, which is moved to the front.

```
// [x,xs,y,ys] => [x,y,xs,ys]
double[*] block(double[*] a)
{
  ijs, ivs = takedrop(cut_half(iota(dim(a))), 1);
  m = flatten(ijs) ++ flattrans(ivs);
  return ptranspose(m, a);
}
```

Unfortunately, in SaC, block is not a self-inverse, because *xs* and *ys* can be vectors of arbitrary lengths.

The unblock function splits the shape after the first two elements. The left-hand side is prepended to the first column of the 2d view of the right-hand side.

```
// [x,y,xs,ys] => [x,xs,y,ys]
double[*] unblock(double[*] a)
{
  ijs, ivs = takedrop(iota(dim(a)), 2);
  m = flattrans([ijs] ++ cut_half(ivs));
  return ptranspose(m, a);
}
```

Finally, we can create pre-blocking and its inverse (as computed by toinv) needed for the mmx version of matrix multiplication. We apply cut_half and cut_pairs as follows:

```
// [x,y,z..., u,v,w,...] => [x,u, y,v, z,w,...]
double[*] preblock(double[*] a)
{
  m = flatten(cut_half(iota(dim(a))));
  return ptranspose(m, a);
}

// [x,u, y,v, z,w,...] => [x,y,z,..., u,v,w,...]
double[*] unpreblock(double[*] a)
{
  m = flatten(cut_pairs(iota(dim(a))));
  return ptranspose(m, a);
}
```

Note that all the blockings presented above simply rearrange array elements according to the changes in the shape. After compiler optimisations, we end-up with computations of offsets into the arrays that we are reshaping. Therefore, there are many equivalent ways to specify these. In the tradition of array languages, we express these transformations in a combinatorial style.

4.2 Matrix Multiplication with In-place Blocking

Per our assumption, we are always blocking on rows and columns. Therefore, mm specification reduces to recursive application of a single case (the last case in the pattern-matching). No other cases are applicable to the chosen shape structure.

In the SaC implementation of mm, we use function overloading to drive the recursion over shapes. We assume that we start with already reshaped arrays, in which case mm can be expressed as follows:

```
double[*] mm(double[*] a, double[*] b)
{
  a = block(a);
  b = block(b);
  c = {[i,j] -> sum({[k] -> mm(a[i,k], b[k,j])})};
  return unblock(c);
}
```

As SaC arrays have the notion of the distinct empty shape, the base case of the matrix multiplication can be given for arrays of the empty shapes (also known as scalars).

```
double[] mm(double[] a, double[] b)
{
  return a*b;
}
```

4.3 Matrix Multiplication with Pre-blocking

A version of the algorithm that operates on pre-blocked arrays (mmx in the specification) can be implemented as follows:

```
double[*] mmx(double[*] a, double[*] b)
{
  return {
    [i,j] -> sum({[k] -> mmx(a[i,k], b[k,j])})
  };
}
```

The notation of the outer array comprehension implicitly implies that i and j are scalars. Summation happens over the array comprehension indexed by k, which is also a scalar. Each recursive calls "chops off" the first two dimensions from the a and b shapes.

As we assume that shapes of a and b are of the same length, and they have even number of elements, the base case for mmx is when both arrays are of the empty shape. As before, we use regular multiplication for the empty-shaped matrices.

```
double[] mmx(double[] a, double[] b)
{
  return a * b;
}
```

We also notice that pre-blocking can be expressed in a shape-recursive manner by repeated application of block/unblock as follows.

```
// [x,y,z,... u,v,w,...] => [x,u, y,v, z,w,...]
double[*] preblock(double[*] a)
{
  return {[i,j] -> preblock(block(a)[i,j])};
}

double[] preblock(double[] a)
{
  return a;
}
```

In this case shape recursion is left-to-right, and the base case is the identity function.

Inverse of pre-blocking is given as follows:

```
// [x,u, y,v, z,w,...] => [x,y,z,..., u,v,w,...]
double[*] unpreblock(double[*] a)
{
  return unblock({[i, j] -> unpreblock(a[i, j])});
}

double[] unpreblock(double[] a)
{
  return a;
}
```

This is right-to-left shape recursion, and the base case is the identity function. Such a blocked pre-blocking is likely to have a better runtime behaviour, as some of the operations get better cache locality.

Algorithm Application. We consider applying both algorithms (mm and mmx) to the matrices a and b of respective shapes $[m, n]$ and $[n, k]$. We assume that m, n and k can be split into equal number of multipliers. We use three multipliers just as an example:

$$xyz = m \qquad uvw = n \qquad spq = k.$$

In this case, we start with reshaping of a and b:

```
a1 = reshape([x,y,z,u,v,w] a);
b1 = reshape([u,v,w,s,p,q] b);
```

After that, mm can be applied to $a1$ and $b1$, but the obtained result needs to be reshaped into the expected shape $[m, k]$:

```
c = reshape([m,k], mm(a1, b1));
```

Application of mmx requires pre-blocking $a1$ and $b1$, and unpreblocking the result. Finally, the result needs to be reshaped into the expected shape $[m, k]$:

```
c = reshape([m,k], unpreblock(mmx(preblock(a1),
                                  preblock(b1))));
```

Weak Typing. Note that we cannot express most of the assumptions about the structure of array shapes in the type system of SaC. Instead, we have to maintain them using programming discipline.

5 Runtime Evaluation

In the previous sections we demonstrated generic specifications of the blocked matrix multiplication. Now, we want to apply these algorithms on a particular architecture and demonstrate, explain how to chose blocking factors and compare our results with state of the art implementations: OpenBLAS [32], BLIS [30] and MKL [11].

5.1 Experimental Setup

We use a multicore machine consisting of two 16 core AMD EPYC 7313 CPUs running at a 3.0 GHz frequency. The machine has private 32KiB L1D and 1MiB L2 caches. Every four cores share a 16 MiB L3 cache. We use OpenBLAS version 0.3.20, SaC 1.3.3 with GCC version 11.3.0, BLIS 0.9.0 and Intel MKL version 2020.4.304. The peak performance of this machine is 1536 Gflops/s.

Each core can do two 32 byte reads and one 32 byte write per clock cycle. That means that the L1 read bandwidth is $3 \cdot 2 \cdot 32 \cdot 32 = 6144$ GB/s. The L2 bandwidth is half this, 3072 GB/s. Every clock cycle a L2 cache can load in 32 bytes from L3, so the L3 bandwidth is $3 \cdot 32 \cdot (32/4) = 768$ GB/s. We have 8 times 16 GB of RAM, running at 3200 MT/s over 8 channels, resulting in 204.8 GB/s bandwidth between RAM and CPU.

Bandwidth vs Compute Rate. The execution speed is limited by how fast we can do operations, and by how quickly we can get data from RAM into the CPU registers. For our machine the latter is:

$$\frac{204.8\text{GB/s}}{8\text{B} \cdot 3\text{Ghz}} \approx 8.5 \text{ doubles (8 bytes) per clock cycle}$$

Our machine can do $32 \cdot 4 \cdot 2 \cdot 2 = 512$ flops (cores, SIMD, fma, instructions per clock). So if we only use each double once, we can expect no more than $\frac{8.5}{512} \approx \frac{1}{60}$th of the peak performance. Therefore, in order to achieve peak performance we are forced to reuse each double we obtain from the memory multiple times. For example, if we reuse each double 30 times, we get half of the peak performance, *etc.*

In principle we have an intensity of $2MKN/(MK + KN)$ flops per double which tells us the problem is compute bound. Of course in practice it is hard to achieve this performance because we cannot store the entire matrix in registers. For this reason we look at the reuse of elements before they are evicted from their respective level in the memory hierarchy. So the unblocked implementation will have an effective intensity of less than $\frac{2 \cdot K}{K} = 2$ as each calculation $(AB)_{ij} = \sum_{p=0}^{K} A_{ip} B_{pj}$ will see the jth column of B being evicted from cache.

5.2 Blocking Factors

In order to achieve peak performance, we use the approach from [7, 30] to find the block sizes that agree with the cache sizes for multiplying $A \in \mathbb{R}^{M \times K}$ by $B \in \mathbb{R}^{K \times N}$. When choosing block sizes we took measurements ranging from filling about half of the cache to the entire cache. We then picked the point where increasing the block size did not further improve runtimes. This way the inner blocks stay small, and we have some more leeway when choosing the outer blocks.

First Blocking. We are going to chose our block sizes in a bottom-up fashion. We specify the size of a matrix as a superscript. Firstly, we notice that the matrix of size 6×8 can be stored entirely in SIMD registers. Our machine has 16 YMM registers, 32 bytes (4 doubles) each. Also, this leaves us at least three registers to do other operations. This means that for a multiplication of a $6 \times k$ and $k \times 8$ matrix, we store only 48 doubles on $2 \cdot 6 \cdot 8 \cdot k$ flops. For $A^{6 \times k}$ and $B^{k \times 8}$ we get a 6-fold reuse of B and an 8-fold reuse of A. Our processor can execute 2 fma instructions per clock cycle, and the L1, L2 cache can provide these operands in 2 and 4 cycles respectively. As 6 and 8 are larger than 4, it suffices to have our matrices in either L1 or L2. Shapes so far: $[6, 250]$ for A and $[250, 8]$ for B.

Second Blocking. The second blocking level stacks all the $A^{6 \times k}$ (denoted A' here) into $A^{m \cdot 6 \times k}$, performing the following operation:

$$A^{m \cdot 6 \times k} = \begin{pmatrix} A'_1 \\ \vdots \\ A'_m \end{pmatrix} \qquad AB = \begin{pmatrix} A'_1 B \\ \vdots \\ A'_m B \end{pmatrix}.$$

We chose m and k such that B fits into L1 and A fits into L2. At the same time, we want to chose m such that B can be streamed into L1 (*i.e.* $m \cdot 6 > 60$) so that we can make the next blocking step. We chose $m = 15$ and $k = 250$. Shapes so far: $[15, 1, 6, 250]$ for A and $[1, 1, 250, 8]$ for B.

Third Blocking. After that, we consider stacking previously defined blocks $A^{90 \times 250}$ and $B^{250 \times 8}$ (denoted A' and B') into column and row vectors:

$$AB = \begin{pmatrix} A'_1 \\ \vdots \\ A'_k \end{pmatrix} \begin{pmatrix} B'_1 & \cdots & B'_n \end{pmatrix} \quad \text{or} \quad (AB)_{ij} = A'_i B'_j.$$

Each of these matrix-multiplications is a smaller matrix-multiplication described above, and this can be calculated at peak compute rate whenever A'_i is in L2 cache and B'_j can be streamed into L1 fast enough. As each element of B'_j is reused $90 > 60$ times, this should be the case. However, we cannot guarantee that the entirety of B'_{j+1} will be prefetched while we are calculating $A'_i B'_p$. Also, loading in a block of A has some latency. So in practice we should to stack only as many blocks of A and B as we can fit into L3 cache in order to accelerate loading these blocks into L1 and L2 cache. We choose to stack 125 such blocks of B and 3 of A. Shapes so far: $[3, 1, 15, 1, 6, 250]$ for A and $[1, 125, 1, 1, 250, 8]$ for B.

Remainder. For matrices larger than $A^{270 \times 250}$ and $B^{250 \times 1000}$, we compute a 'normal' matrix-multiplication in terms of these blocks (denoted A' and B'):

$$AB = \begin{pmatrix} A'_{1,1} & \cdots & A_{1,\frac{K}{250}} \\ \vdots & \ddots & \vdots \\ A'_{\frac{M}{270},1} & \cdots & A'_{\frac{M}{270},\frac{K}{250}} \end{pmatrix} \begin{pmatrix} B'_{1,1} & \cdots & B_{1,\frac{N}{1000}} \\ \vdots & \ddots & \vdots \\ B'_{\frac{K}{250},1} & \cdots & B'_{\frac{K}{250},\frac{N}{1000}} \end{pmatrix}$$

Final shapes: $[\frac{M}{270}, \frac{K}{250}, 3, 1, 15, 1, 6, 250]$ for A and $[\frac{K}{250}, \frac{N}{1000}, 1, 125, 1, 1, 250, 8]$ for B.

5.3 Adjustments

Our initial experiments demonstrated the following two problems. Firstly, when SaC is making a recursive call with a submatrix, it does not detect read-only access to the subblock, and makes a copy. Secondly, neither SaC, nor the underlying C compilers can generate the matrix multiplication kernel that operates in registers. Sadly as it can be, we have to introduce the following workarounds.

Good Kernel. We use the FFI mechanism of SaC and implement the kernel using GCC vector extensions [6, 23]. This gives us a nice and easy formulation of the kernel that is portable across all the architectures supported by GCC.

```
// MR = 6; KC = 250; NR = 8
typedef double vect
  __attribute__((__vector_size__(32), aligned(8)));
void matmul(double **cp, double *a, double *b,
            int offset_a, int offset_b)
{
  double *c = malloc(MR * NR * sizeof(double));
  *cp = c; a = a + offset_a; b = b + offset_b;

  real (*as)[MR][KC  ] = (real (*)[MR][KC  ])a;
  vect (*bv)[KC][NR/4] = (vect (*)[KC][NR/4])b;
  vect (*cv)[MR][NR/4] = (vect (*)[MR][NR/4])c;

  vect zero = {0.,0.,0.,0.};
  vect c00 = zero, c01 = zero;
  vect c10 = zero, c11 = zero;
  vect c20 = zero, c21 = zero;
  vect c30 = zero, c31 = zero;
  vect c40 = zero, c41 = zero;
  vect c50 = zero, c51 = zero;
  for (size_t k = 0; k < KC; k++) {
    vect b0 = (*bv)[k][0];
    vect b1 = (*bv)[k][1];
    real a0 = (*as)[0][k];
    real a1 = (*as)[1][k];
    real a2 = (*as)[2][k];
    real a3 = (*as)[3][k];
```

```
        real a4 = (*as)[4][k];
        real a5 = (*as)[5][k];

        c00 += a0 * b0; c01 += a0 * b1;
        c10 += a1 * b0; c11 += a1 * b1;
        c20 += a2 * b0; c21 += a2 * b1;
        c30 += a3 * b0; c31 += a3 * b1;
        c40 += a4 * b0; c41 += a4 * b1;
        c50 += a5 * b0; c51 += a5 * b1;
    }
    (*cv)[0][0] = c00; (*cv)[0][1] = c01;
    (*cv)[1][0] = c10; (*cv)[1][1] = c11;
    (*cv)[2][0] = c20; (*cv)[2][1] = c21;
    (*cv)[3][0] = c30; (*cv)[3][1] = c31;
    (*cv)[4][0] = c40; (*cv)[4][1] = c41;
    (*cv)[5][0] = c50; (*cv)[5][1] = c51;
}
```

Bad Memory. We eliminate the need of copying by a slight modification in the implementation of mmx. We call this version mmy: instead of passing a sub-block as one argument, we pass the main array and the index into the block. The overloading of mmy happens on the last two arguments that increase on each recursive step by two elements. We know that we are dealing with 8-dimensional array, so the base case of our overloading calls our kernel.

```
inline double[*]
mmy(double[*] a, int[.] ia,
    double[*] b, int[.] ib)
{
  sha = drop(shape(ia), shape(a));
  shb = drop(shape(ib), shape(b));
  shco,shci = takedrop(mmshape(sha, shb), 2);

  return { [i,j] -> with { // Inlined sum
                     ([0] <= [l] < [sha[1]]):
                       mmy (a, (ia ++ [i,l]),
                            b, (ib ++ [l,j]));
                   }: fold(+, genarray(shci, 0d))
         | [i,j] < shco};
}

inline double[.,.]
mmy(double[*] a, int[6] ia,
    double[*] b, int[6] ib)
{
  return matmul(a, b,
                row_major(ia++[0,0], shape(a)),
                row_major(ib++[0,0], shape(b)));
}
```

This problem has been encountered before and studied in [19]. The authors believe it should be possible to extend this optimisation to be applicable to matrix-multiplication as well.

5.4 Timings

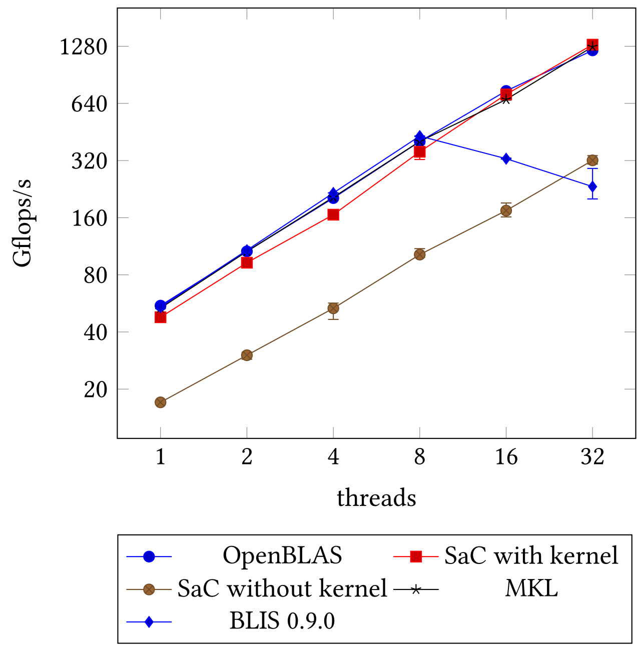
We have measured a multiplication of a 8640×10000 matrix by a 10000×10000 matrix. We provide two SaC versions. The first one is blocked as described in Subsection 5.2, with a handwritten kernel for the two-dimensional computation. The second one does not use a handwritten kernel. As GCC cannot use blocking at the register level, we use squarish blocks: 40×40 blocks to target L1, and 240×200 for A, C and 200×200 for B to target L2. We compare it to the state of the art implementation OpenBLAS. The timings are averaged over 10 runs, and we draw error bars from the slowest to the fastest run. These results are summarised on the left-hand side of Figure 1. The axes are log-scaled, so linear speedup is also linear in the graph. The fastest programs are rather close to each other, so we also graph the efficiencies for these (on the right-hand side of Figure 1) compared to the sequential performance to OpenBLAS. The astute reader may have noticed that OpenBLAS and MKL perform better than the 48 Gflops/s theoretical peak performance on one core. It could be due to Strassen's algorithm as explained in [10], but we have not found a confirmation of this in the literature or OpenBLAS's codebase. The alternative is that the actual clock speeds are higher than 3 GHz, even though we have verified that the settings cap it at 3 GHz.

OpenBLAS achieves 55 Gflops/s on one core, MKL achieves 53 Gflops/s and SaC with kernel 48 Gflops/s. A reason that the SaC kernel is slower, is that it allocates memory for the 6×8 block of C instead of directly writing to the memory of C. MKL and OpenBLAS, opposed to the SaC version, does not have blocks stored in contiguous memory, so they need to pack these blocks. We see that this causes the SaC version to scale better and catch up to OpenBLAS higher core counts: 1305 Gflops/s on 32 cores for SaC, 1218 Gflops/s for OpenBLAS, and 1272 Gflops/s for MKL. The reason this only happens at higher core counts is that packing takes a little bandwidth, which is more contested the more cores are used. SaC in contrast does not have to pack, as we have benchmarked the preblocked version. The SaC version without a kernel is about a factor four slower, and has less speedup going from 16 to 32 cores. On 32 cores, compared to the fastest sequential time by OpenBLAS, we have efficiencies of 69%, 72%, 74%, 18% for OpenBLAS, MKL, SaC with kernel, SaC without kernel, respectively. The unblocked version is unsurprisingly very slow, 12 Gflops/s on 32 cores, a factor 100 slower than SaC with kernel, and 27 times slower than the blocked SaC version without kernel. We reuse each element once before it is evicted from cache, so we cannot expect more than $\frac{1532}{60} \approx 26$ Gflops/s. Combined with the strided access of B, it gives the poor result.

For BLIS we use use OpenMP for the backend and thread-binding strategy `OMP_PROC_BIND=close` and `OMP_PLACES=cores` as recommended by the authors. It does well up to including 8 cores, and then experiences slowdown.

5.5 Blocking and Unblocking

The SaC compiler is not required to materialize A, B in memory, block them, and then run the matrix-multiplication. Instead, it could immediately generate A, B in blocked form. For this reason we did not include the blocking cost in Figure 1. In the following table we list those times for cases where the SaC compiler cannot immediately generate A, B in



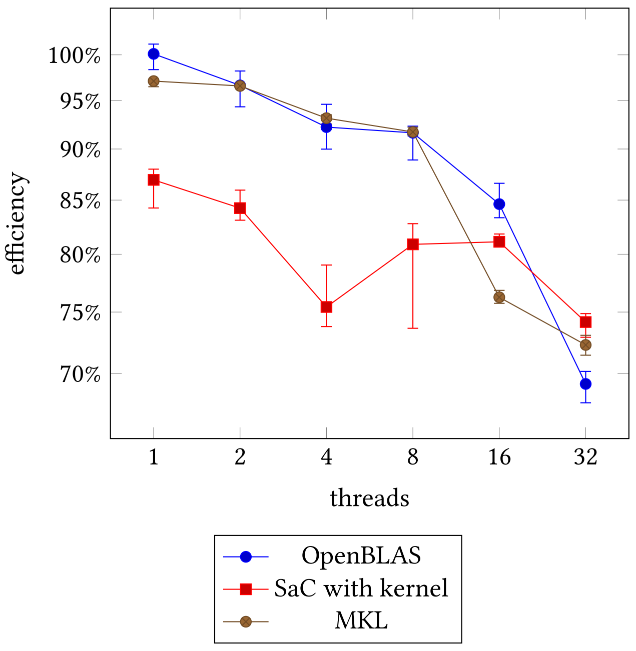


Figure 1. Performance of computing AB for matrix A of size 8640×10000 and B of size 10000×10000. On the left-hand side the absolute figures are presented, on the right hand side we show the efficiencies (per core) compared to OpenBLAS's sequential performance. The SaC version uses four-level blocking described above.

blocked form. As before we use $A^{8640\times 10^4}$, $B^{10^4\times 10^4}$ splitting the runtime between blocking, matrix-multiplication, and unblocking in seconds, averaging the time over 10 runs.

Proc	Block	σ	Matmul	σ	Unblock	σ
1	5.317	0.118	38.78	0.19	2.799	0.081
2	2.675	0.011	19.60	0.06	1.494	0.010
4	1.439	0.208	9.98	0.05	0.770	0.022
8	0.714	0.044	5.00	0.02	0.417	0.055
16	0.464	0.009	2.53	0.01	0.246	0.022
32	0.302	0.007	1.44	0.01	0.124	0.029

6 Related Work

The idea to use blocking to improve performance of numerical algorithms dates back to the 1960s [15], roughly coinciding with the introduction of memory hierarchies. Since then, a large body of research has investigated [12, 13] ways on how to implement blocked linear algebra operations efficiently. This work provides guidelines on how to structure blocked algorithms, assuming manual implementations. The canonical results in this area are described in [7] which constitutes the basis for today's state of the art hand-written libraries such as OpenBLAS [32] and BLIS [30]. In contrast to our work, the manual encodings in low-level languages of these approaches raises potential concerns about their correctness and their maintainability.

Another area of research in the context of blocking aims to enable compilers to introduce blocking into blocking-free implementations. This work is based on dependency analyses such as the ones in [1] and the polyhedral model [16] which provides a mathematical framework for analysing and transforming loop nests. More recent work in this context [5] includes possible data layout transformations as well, which, at least in principle, enables compiler-driven transformations into code very similar to the code that is generated from our shape-generic specification.

One of the key challenges of the compiler-introduced blocking approach is the need to find suitable parameters. Analytical models have been proposed [14], but there are also attempts to tune the parameters based on exhaustive experiments or by applying some form of machine learning. ATLAS [2] is an implementation of BLAS that automatically tunes architecture specific parameters. Further examples of autotuning can be found in [35] and [28]. Lift [27] uses autotuning, by applying rewrite rules to a high-level functional specification of the algorithm. While our approach does not solve the parameter finding problem either, it is possible for the programmer to experiment with different blocking scenarios without rewriting the rank-polymorphic matrix multiply at all. This is similar in spirit to approaches such as those of Halide [17] or FLAME [34], which offer the programmer to specify separately which transformations should be introduced by the compiler. Here, the key difference is that our approach allows the blocking to be encoded in the array shapes and, thus, in the source language itself. Another difference is that the dependently typed setting offers formal correctness guarantees which are not available in the aforementioned approaches.

Closer to our work is the work on Multi-Dimensional Homomorphisms [18]. The formulation as homomorphisms provides identity guarantees similar to ours and the tiling can be steered by the injection of tiling operators. The main difference to our work is the dependently typed setup in our case which enables further guarantees such as the absence of out-of-bounds accesses.

Elliott proposes [3] a Haskell framework that uses structured views on data to give rise to natural functional specifications. This work is also very close to our approach; however, we can also guarantee correctness of our transformations by using dependent types. Furthermore, Elliott's work focusses on the specification aspects, but does not elaborate on how to generate high-performance code. Sequoia [4] is a language that allows for a specification close to ours, the main difference being that Sequoia does not have rank-polymorphic arrays, and hence requires explicit creations of subarrays in the recursive call. Some advanced type system for array languages have been proposed [9, 26]. However, these do not capture the rank-polymorphism that is crucial to our approach. Dependent types for array languages have been proposed in [29, 33]. The closest-related treatment of arrays within a dependently-typed setting is proposed in [21] where arrays are described as polynomial functors.

Finally, this work relates to earlier work on shape-guided parallel implementations of scan operations described in [31]. The difference is that here we use the shape to guide the blocking instead of the parallelism, and we start from a dependently typed specification.

7 Conclusion

This paper proposes to implement blocked numerical algorithms through rank-polymorphic functions on arrays. This allows for a concise formulation of arbitrary blocking levels, enabling programmers to encode the desired blocking factors through array shapes. Besides conciseness, the ability to express arbitrary blocking through a single generic algorithm gives the opportunity to prove the correctness of all possible blocking variants mechanically. We demonstrate this idea at the example of matrix multiplication. Starting from a high-level specification in Agda, we verify that all possible blocking variants compute the same result as the canonical, unblocked version. A simple manual transliteration into SaC demonstrates that it is possible to generate very efficient parallel code from such a generic specification.

While several manual steps are required to achieve the transition from Agda to code that is competitive with hand-tuned high-performance implementations, a mechanisation of this approach seems within reach. Dependent types prove to be a good starting point of the specification. We can encode rank-polymorphism, shape algebras, guarantee lack of out-of-bound indexing, and prove correctness of blocking. Additional invariants concerning the structure of the array or its shape, in principle, are expressible. The standard Martin-Löf type theory is powerful enough, and we do not need quotients or higher inductive types. Treating arrays as functions is very powerful for verification purposes.

Currently, the translation from Agda to SaC is performed manually. In [22] we show a way to automate such a translation using reflection mechanisms of Agda. Mainly, this translation is a technical task, but two aspects require further investigation: (i) how to avoid runtime representation of types that are not needed; (ii) how to control excessive inlining of arrays triggered by normalisation. Quantitative type theory seem to be a good answer for the former. Effect systems or the ideas from [8] should help with the latter.

SaC-like array languages seem to be a good choice for the "backend". Being a functional rank-polymorphic array language, dependently-typed specifications can be translated into SaC programs almost one-to-one, by stripping out most of the type guarantees and the proof-related parts. Our generically blocked algorithm is expressed with less than 10 lines of code. The SaC compiler leverages shape specialisation and advanced optimisation techniques to achieve the high-performance parallel code. However, the availability of further guarantees in Agda, at least in principle, opens up possibilities for providing the compiler with invariance properties that could help improving the code generation further. Ideally, we would want to formally verify that the SaC programs preserve semantics of the original specification, but this would require not only a verified Agda to SaC translation, but also a verified SaC to C compiler as well as a verified C compiler. This would take enormous efforts which seems out of reach at this time.

When striving for the highest levels of performance, we see that some further optimisations within the SaC compiler would be desirable. Specifically, the ability to implement subarray selections without copying elements seems crucial in this context. While this mainly constitutes an engineering challenge, the paper shows that we can achieve these effects by means of a workaround in the form of a re-write and a 40-line kernel written in C. Jointly, these tweaks lead to parallel performance that is competitive with state-of-the-art libraries BLIS, OpenBLAS and MKL.

Data-Availability Statement

Both, the Agda proof and scripts for repeating our experiments are available as artefact [24]. A description on how to use the artefact can be found in the appendix of this paper.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their constructive feedback which helped improving the paper. This work is partially funded by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

A Artefact

A.1 Abstract

This artefact describes the experiments that were conducted for the FHPNC'2023 paper *Rank-Polymorphism for Shape-Guided Blocking*.

The following workflow makes use of scripts to compile, run, and evaluate presented benchmarks in Figure 1 and the blocking table of section 5. The figure showing performance relative to the peak is not reproduced as this depends on the machine that is used for running the experiments.

A.1.1 Agda Specification Check-list.

- **Agda version:** 2.6.3, available at https://agda.readthedocs.io/.
- **Agda standard library:** version 1.7.2, available at https://github.com/agda/agda-stdlib.

A.1.2 Runtime Experiments Check-list.

- **Compilation:** GCC C compiler 11.3.0, sac2c compiler v1.3.3-1079-1 (https://sac-home.org/download:main), make. Optionally: a SLURM environment.
- **Hardware:** a multithreaded machine.
- **Libraries:** OpenBLAS 0.3.20, BLIS 0.9.0, MKL 2020.4.304.
- **Output:** a Latex installation that supports pgfplots and tikz.

A.2 Description

The artefact is hosted on the following GitLab server instance: https://gitlab.sac-home.org/sac-group/2023-array.

Files that are related to the Agda specification are in the root directory. Files that are needed to reproduce runtime figures are in the directory named artefact.

A.3 Reproducing Proofs

The paper is written using literate Agda, which means that all the definitions in the code have been type-checked. The Agda file that corresponds to Section 3 can be found in the file `model.lagda`. Some additional proofs such as of mmx≈mm in Section 4 can be found in the file `proof.agda`. We introduce inductive reshapes which make the proofs easier to articulate but which are are orthogonal to the content of the paper.

Reproducing the proofs requires to pass the agda/lagda files to the `agda` binary as follows:

1. `agda model.lagda`
2. `agda proof.agda`

A.4 Reproducing Runtime

1. Ensure that all the dependencies listed in the check-list are installed.
2. Clone the repository and enter the artefact directory
   ```
   $ cd 2023-array/artifact
   ```
3. Replace `$HOME/sac2c/build/sac2c_d` in the Makefile with your build of the SaC compiler.
4. **If using SLURM:** Replace the lines
   ```
   #SBATCH -account=csmpi
   #SBATCH -partition=csmpi_fpga_long
   ```
   with your SLURM setup using *e.g.*
   ```
   $ sed -i 's/csmpi_fpga_long/PARTITION/g' *.sh
   $ sed -i 's/csmpi/ACCOUNT/g' *.sh
   ```
5. Build the binaries with `make` if not using SLURM, and `sbatch build.sh` is using SLURM.
6. Run the following scripts, using `sbatch` only if using SLURM.
   ```
   $ [sbatch] ./bench_blis.sh 8640 10000 10000 1 32 10
   $ [sbatch] ./bench_openblas.sh 8640 10000 10000 1 32 10
   $ [sbatch] ./bench_mkl.sh 8640 10000 10000 1 32 10
   $ [sbatch] ./bench.sh 1 32 10
   $ [sbatch] ./bench_naive.sh 1 32 10
   $ [sbatch] ./split.sh 1 32 10
   ```
   If on a system with less or more than 32 cores, you can replace 32 with your maximum number of cores.
7. Build the graphs with
   ```
   $ pdflatex graph.tex
   ```
   The result will be in a pdf file `graph.pdf`.

References

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[2] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35. https://doi.org/10.1016/S0167-8191(00)00087-9 New Trends in High Performance Computing.

[3] Conal Elliott. 2017. Generic Functional Parallel Algorithms: Scan and FFT. *Proc. ACM Program. Lang.* 1, ICFP, Article 7 (aug 2017), 25 pages. https://doi.org/10.1145/3110251

[4] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 4–4. https://doi.org/10.1109/SC.2006.55

[5] Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-Performance Generalized Tensor Operations: A Compiler-Oriented Approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (sep 2018), 27 pages. https://doi.org/10.1145/3235029

[6] GNU Project. [n. d.]. Using Vector Instructions through Built-in Functions. https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html, Last accessed on 2023-05-30.

[7] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (may 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[8] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. 2022. Controlling unfolding in type theory.

arXiv:2210.05420 [cs.LO]

[9] Troels Henriksen and Martin Elsman. 2021. Towards Size-Dependent Types for Array Programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3460944.3464310

[10] Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. Van De Geijn. 2016. Strassen's Algorithm Reloaded. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 690–701. https://doi.org/10.1109/SC.2016.58

[11] Intel. [n. d.]. Intel oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html, Last accessed on 2023-05-31.

[12] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. 1997. Data-Centric Multi-Level Blocking. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 346–357. https://doi.org/10.1145/258915.258946

[13] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not.* 26, 4 (apr 1991), 63–74. https://doi.org/10.1145/106973.106981

[14] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (aug 2016), 18 pages. https://doi.org/10.1145/2925987

[15] A. C. McKellar and E. G. Coffman. 1969. Organizing Matrices and Matrix Operations for Paged Memory Systems. *Commun. ACM* 12, 3 (mar 1969), 153–165. https://doi.org/10.1145/362875.362879

[16] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. 2007. Iterative Optimization in the Polyhedral Model: Part I, One-Dimensional Time. In *International Symposium on Code Generation and Optimization (CGO'07)*. 144–156. https://doi.org/10.1109/CGO.2007.21

[17] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2017. Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Commun. ACM* 61, 1 (dec 2017), 106–115. https://doi.org/10.1145/3150211

[18] Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2019. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 354–369. https://doi.org/10.1109/PACT.2019.00035

[19] Sven-Bodo Scholz. 1998. With-loop-folding in Sac — Condensing Consecutive Array Operations. In *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers (Lecture Notes in Computer Science, Vol. 1467)*, Chris Clack, Tony Davie, and Kevin Hammond (Eds.). Springer, 72–92. https://doi.org/10.1007/BFb0055425

[20] Sven-Bodo Scholz and Artjoms Šinkarovs. 2021. Tensor Comprehensions in SaC. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages* (Singapore, Singapore) *(IFL '19)*. Association for Computing Machinery, New York, NY, USA, Article 15, 13 pages. https://doi.org/10.1145/3412932.3412947

[21] Artjoms Šinkarovs. 2020. Multi-dimensional Arrays with Levels. In *Proceedings Eighth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2020, Dublin, Ireland, 25th April 2020 (EPTCS, Vol. 317)*, Max S. New and Sam Lindley (Eds.). 57–71. https://doi.org/10.4204/EPTCS.317.4

[22] Artjoms Sinkarovs and Jesper Cockx. 2021. Choosing is Losing: How to combine the benefits of shallow and deep embeddings through reflection. *CoRR* abs/2105.10819 (2021). arXiv:2105.10819 https://arxiv.org/abs/2105.10819

[23] Artjoms Šinkarovs and Sven-Bodo Scholz. 2012. Portable Support for Explicit Vectorisation in C. In *16th Workshop on Compilers for Parallel Computing (CPC'12)*.

[24] Artjoms Šinkarovs, Sven-Bodo Scholz, and Thomas Koopman. 2023. Artefact for Rank-Polymorphism for Shape-Guided Blocking. https://doi.org/10.1145/3580402

[25] Artjoms Šinkarovs, Sven-Bodo Scholz, and Thomas Koopman. 2023. Sources of Rank-Polymorphism for Shape-Guided Blocking. https://gitlab.sac-home.org/sac-group/2023-array.

[26] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46.

[27] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/2784731.2784754

[28] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P. Sadayappan, and Fabrice Rastello. 2023. Autotuning Convolutions Is Easier Than You Think. *ACM Trans. Archit. Code Optim.* 20, 2, Article 20 (mar 2023), 24 pages. https://doi.org/10.1145/3570641

[29] Kai Trojahner and Clemens Grelck. 2011. Descriptor-Free Representation of Arrays with Dependent Types. In *Implementation and Application of Functional Languages*, Sven-Bodo Scholz and Olaf Chitil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 100–117.

[30] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. https://doi.acm.org/10.1145/2764454

[31] Artjoms Šinkarovs and Sven-Bodo Scholz. 2022. Parallel Scan as a Multidimensional Array Problem. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (San Diego, CA, USA) *(ARRAY 2022)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3520306.3534500

[32] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/2503210.2503219

[33] Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732

[34] Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. 2008. Scalable Parallelization of FLAME Code via the Workqueuing Model. *ACM Trans. Math. Softw.* 34, 2, Article 10 (mar 2008), 29 pages. https://doi.org/10.1145/1326548.1326552

[35] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

Received 2023-05-31; accepted 2023-06-28