Generating High Performance Code for Irregular Data Structures using Dependent Types

Federico Pizzuti
The University of Edinburgh
Edinburgh, Scotland, United Kingdom
federico.pizzuti@ed.ac.uk

Michel Steuwer
The University of Edinburgh
Edinburgh, Scotland, United Kingdom
michel.steuwer@ed.ac.uk

Christophe Dubach
McGill University
Canada
christophe.dubach@mcgill.ca

Abstract

Parallel architectures offer high performance but are challenging to program. Data parallel functional languages offer a solution by providing a high-level programming model to work with accelerators such as GPUs. Existing languages are designed to work with dense arrays, limiting their usefulness in expressing irregular data structures, such as graphs and sparse matrices important in many application domains.

This paper addresses this limitation by extending a data-parallel language with *limited* dependent types, including position dependent arrays and dependent pairs to model irregular data structures. The approach is demonstrated through three case studies: dense to sparse matrix conversion, sparse matrix-vector multiplication, and parallel breadth-first search.

Experimental results show that this approach outperforms state-of-the-art implementations on GPUs. Compared to Nvidia's cuSparse, our automatically generated code achieves an average speedup of 1.2× for dense to sparse matrix conversion and 1.3× for sparse matrix-vector multiplication.

CCS Concepts: • **Software and its engineering → Compilers**; **Parallel programming languages**; • **Theory of computation → Type theory**.

Keywords: Irregular Data Structures, Dependent Types

ACM Reference Format:
Federico Pizzuti, Michel Steuwer, and Christophe Dubach. 2021. Generating High Performance Code for Irregular Data Structures using Dependent Types. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (FHPNC '21), August 22, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3471873.3472977

1 Introduction

Modern parallel hardware offers great opportunities for performance. Programming such platforms is a however a challenging task even for the experienced programmer, particularly as they expose a low-level interface. Parallel programming is an inherently complex task, further complicated by each platform's specific details. The lack of higher-level abstractions in these low-level languages gives the programmer few tools to control this complexity.

A trend in addressing these problems has been the development of specialized high-level functional languages, targeting such devices. Many examples have been developed in recent years, including *Accelerate*[11],*Futhark* [8], *Single Assignment C* [17] *Lift* [18, 19] and *Rise* [5]. These languages combine functional and array programming paradigms, leveraging expressive type systems to achieve both efficient code generation in the parallel context, and correctness in the presence of the targeted platforms' many constraints.

Traditionally, such languages have focused on computations over data with regular layouts that are used pervasively in many application domains. Simple yet simultaneously flexible, regular multi-dimensional arrays allow for a convenient and well-understood programming style.

However, there are important domains that naturally operate on irregularly structured data, many of interest to the high-performance computing community. Such domains include Sparse Linear Algebra and Graph Algorithms. For applications belonging to these domains, being limited to dense, regular data structures imposes unacceptable performance and expressiveness penalties.

Recent work has shown how it is possible to express programs operating on irregular arrays with a statically-known shape expressed using a limited form of dependent types [14]. This work has been extended to cover computations over sparse data structures with dynamic shapes [15], while still preserving a compilation model that can produce high-performance implementations.

Unfortunately, the prior work has a significant limitation that prevents the expression of many irregular applications: so far, irregular data structures can only be consumed as input but never be produced as an output of a program. This limitation comes from the dependent types that are used to model irregular data structures: *dependent functions* and *position dependent arrays* are used together to express the dependencies between multiple parameters that are passed separately to the program representing the values stored in the data structure and metadata explaining the shape of the irregular data structure. Returning a *position dependent array* from a program in a type-safe way is not possible, as it contains universally quantified identifiers introduced by the *dependent functions*.

This paper overcomes this significant limitation and — for the first time — provides a universal solution for generating high-performance code for applications with irregular data structures by leveraging dependent types. We built upon the prior work and add a limited form of *dependent pairs* that still permits high-performance code generation. This existentially qualified type allows irregular data structures to be produced and returned as an output from a program. We demonstrate their use and interactions with other dependent types to implement algorithms operating over irregular data structures, including programs not previously expressible. Our implementation of dependent pairs differs from the canonical versions found in mainstream dependently typed languages such as *Idris* [2] or *Agda* [13], as it includes specific solutions enabling the generation of high-performance parallel GPU code. Finally, we demonstrate using three case studies competitiveness with state-of-the-art low-level handwritten implementations.

In summary, this paper makes the following contributions:

- present the first data parallel functional programming language with universal support for irregular data structures by using dependent types (Sections 2 & 3),
- demonstrate the use of the language to implement programs consuming and generating irregular data structures in three case studies: *dense to csr matrix conversion*, *csr sparse matrix vector multiplication* and *breath first search* frontier expansion (Section 4),
- discuss compilation challenges and techniques to overcome them, focusing on GPU compilation (Section 5),
- experimentally evaluate the case studies by comparing the generated code against state-of-the-art low-level handwritten implementations (Section 6).

2 Language Overview

In this section, we present our functional array programming language that builds upon prior work presented in [5, 14] and [15]. We introduce the language's type system and primitives here as technical background, highlighting the existing dependent types. We extend them in Section 3 with dependent types and primitives that allow to produce and return irregular data structures from programs. The language presented in this section and the following one provides universal support for generating high-performance code for applications operating on irregular data structures.

We start by presenting the type system before introducing common data parallel primitives and their types.

2.1 Type System

Figure 1 shows the well-formedness rules of our type system. We use *Kinds* (1a) for introducing a limited form of dependent types that still permits high-performance code generation. Δ introduced in Figure 1b is the *kinding environment* containing mappings of type variables and their kinds.

The first kind are *natural numbers* (nat). We consider two nats equal if they are equal in their intepretation as natural numbers. Figure 1d shows the well-formedness of natural numbers, including literals, binary expressions, and the results of a type-level function application.

We introduce two different kinds of type-level functions (Figure 1e): functions mapping natural numbers to natural numbers (nat→nat) and functions mapping natural numbers to data types (nat→data).

Figure 1f shows the data types: *scalar types*, *index types* $\mathsf{idx}[n]$ representing a number smaller than n, *array types* $n{\bullet}T$ representing n elements of type T, *pair types*, nat→data type-level function *application*, and, finally, *position dependent array types*.

Position Dependent Array Types. Expressed as $n{\bullet\bullet}i \mapsto T$, these are arrays in which the element type depends on its position within the array. The dependence is statically known: $i \mapsto T$ is the notation for a type-level function nat→data.

As arrays are the only data type parametric on nat expressions and the language disallows matching over the number to return disjoint types, *Position dependent arrays* are implemented as contiguous array of the innermost scalar element type: a *zero-cost abstraction* without additional metadata.

Position dependent arrays are a superset of ordinary arrays. The language therefore provides a primitive, **asDepArray**, which interprets an ordinary array to a dependent one, allowing for the use of a position-aware version of common primitives such as **map**.

Figure 1g shows the remaining types of kind 'type': *function types*, *dependent function types*, and we consider all data types to be part of this kind as well. The kinding structure presented here, particularly the separation of data and function types, ensures that we cannot represent functions as data that we need to store in memory – something that is not easily supported on GPUs.

Dependent Function Types. These function types of the form $(n : \mathsf{nat}) \to T$ are parameterized with a nat parameter that may appear in T. This type allows polymorphism over nat, in practice often used for programs that compute arrays of dependent size.

Figure 2 shows typing rules for our type system. These rules are standard, except that types in the *typing environment* Γ must be well-kinded in the kinding environment Δ following the rules in Figure 1, and for the Conv rule, which

$$\kappa ::= \mathsf{nat} \mid \mathsf{nat}{\to}\mathsf{nat} \mid \mathsf{nat}{\to}\mathsf{data} \mid \mathsf{data} \mid \mathsf{type}$$

(a) Kinds

$$\frac{x : \kappa \in \Delta}{\Delta \vdash x : \kappa}$$

(b) Kinding Structural Rules

$$\frac{\models \forall \sigma : dom(\Delta) \to \mathbb{N}.\sigma(N) = \sigma(M)}{\Delta \vdash N \equiv M : \mathsf{nat}}$$

(c) Type Equality

$$\frac{}{\Delta \vdash \underline{\ell} : \mathsf{nat}} \qquad \frac{\oplus \in (+, *, \ldots) \quad \Delta \vdash N : \mathsf{nat} \quad \Delta \vdash M : \mathsf{nat}}{\Delta \vdash N \oplus M : \mathsf{nat}} \qquad \frac{\Delta \vdash N : \mathsf{nat} \quad \Delta \vdash F_N : \mathsf{nat}{\to}\mathsf{nat}}{\Delta \vdash F_N\ N : \mathsf{nat}}$$

(d) Natural Numbers

$$\frac{\Delta, n : \mathsf{nat} \vdash N : \mathsf{nat}}{\Delta \vdash n \mapsto N : \mathsf{nat}{\to}\mathsf{nat}} \qquad \frac{\Delta, n : \mathsf{nat} \vdash T : \mathsf{data}}{\Delta \vdash n \mapsto T : \mathsf{nat}{\to}\mathsf{data}}$$

(e) Type-level Functions

$$\frac{}{\Delta \vdash \mathsf{f32} : \mathsf{data}}\ \textsc{Prim} \qquad \frac{\Delta \vdash N : \mathsf{nat}}{\Delta \vdash \mathsf{idx}[N] : \mathsf{data}}\ \textsc{Index} \qquad \frac{\Delta \vdash N : \mathsf{nat} \quad \Delta \vdash T : \mathsf{data}}{\Delta \vdash N_{\bullet}T : \mathsf{data}}\ \textsc{Array} \qquad \frac{\Delta \vdash S : \mathsf{data} \quad \Delta \vdash T : \mathsf{data}}{\Delta \vdash S \times T : \mathsf{data}}\ \textsc{Pair}$$

$$\frac{\Delta \vdash F_T : \mathsf{nat}{\to}\mathsf{data} \quad \Delta \vdash N : \mathsf{nat}}{\Delta \vdash F_T\ N : \mathsf{data}}\ \textsc{NatToDataApp} \qquad \frac{\Delta \vdash N : \mathsf{nat} \quad \Delta \vdash F_T : \mathsf{nat}{\to}\mathsf{data}}{\Delta \vdash N_{\bullet\bullet}F_T : \mathsf{data}}\ \textsc{PosDepArray}$$

(f) Data Types

$$\frac{\Delta \vdash \theta_1 : \mathsf{type} \quad \Delta \vdash \theta_2 : \mathsf{type}}{\Delta \vdash \theta_1 \to \theta_2 : \mathsf{type}}\ \textsc{Fun} \qquad \frac{\Delta, x : \kappa \vdash \theta : \mathsf{type} \quad \kappa \in (\mathsf{nat}, \mathsf{data}, \mathsf{nat}{\to}\mathsf{data})}{\Delta \vdash (x{:}\kappa) \to \theta : \mathsf{type}}\ \textsc{DepFun} \qquad \frac{\Delta \vdash \theta : \mathsf{data}}{\Delta \vdash \theta : \mathsf{type}}\ \textsc{Data}$$

(g) Types

Figure 1. Well-formedness of Types

$$\frac{x : \theta \in \Gamma}{\Delta \mid \Gamma \vdash x : \theta}\ \textsc{Var} \qquad \frac{\Delta \mid \Gamma \vdash P : \theta_1 \quad \Delta \vdash \theta_1 \equiv \theta_2 : \mathsf{type}}{\Delta \mid \Gamma \vdash P : \theta_2}\ \textsc{Conv} \qquad \frac{\mathbf{prim} : \theta \in \textsc{Primitives}}{\Delta \mid \Gamma \vdash \mathbf{prim} : \theta}\ \textsc{Prim}$$

(a) Structural Rules

$$\frac{\Delta \mid \Gamma, x : \theta_1 \vdash P : \theta_2}{\Delta \mid \Gamma \vdash \lambda x.P : \theta_1 \to \theta_2}\ \textsc{Lam} \qquad \frac{\Delta \mid \Gamma_1 \vdash P : \theta_1 \to \theta_2 \quad \Delta \mid \Gamma_2 \vdash Q : \theta_1}{\Delta \mid \Gamma_1, \Gamma_2 \vdash P\ Q : \theta_2}\ \textsc{App}$$

$$\frac{\Delta, x : \kappa \mid \Gamma \vdash P : \theta \qquad x \notin fv(\Gamma)}{\Delta \mid \Gamma \vdash \Lambda x.P : (x{:}\kappa) \to \theta}\ \textsc{TLam} \qquad \frac{\Delta \mid \Gamma \vdash P : (x{:}\kappa) \to \theta \quad \Delta \vdash \tau : \kappa}{\Delta \mid \Gamma \vdash P\ \tau : \theta[\tau/x]}\ \textsc{TApp}$$

(b) Abstraction and Application Rules

Figure 2. Typing Rules

$$\begin{aligned}
\texttt{map} &: (n : \mathsf{nat}) \to (t_1 : \mathsf{data}) \to (t_2 : \mathsf{data}) \to (t_1 \to t_2) \to n_{\bullet}t_1 \to n_{\bullet}t_2 \\
\texttt{map} &: (n : \mathsf{nat}) \to (ft_1 : \mathsf{nat}{\to}\mathsf{data}) \to (ft_2 : \mathsf{nat}{\to}\mathsf{data}) \to ((i : \mathsf{nat}) \to ft_1(i) \to ft_2(i)) \to n_{\bullet\bullet}ft_1 \to n_{\bullet\bullet}ft_2 \\
\texttt{fold} &: (n : \mathsf{nat}) \to (t_1 : \mathsf{data}) \to (t_2 : \mathsf{data}) \to (t_1 \to t_2 \to t_1) \to t_1 \to n_{\bullet}t_2 \to t_1 \\
\texttt{zip} &: (n : \mathsf{nat}) \to (t_1 : \mathsf{data}) \to (t_2 : \mathsf{data}) \to n_{\bullet}t_1 \to n_{\bullet}t_2 \to n_{\bullet}t_1 \times t_2 \\
\texttt{split} &: (n : \mathsf{nat}) \to (m : \mathsf{nat}) \to (t : \mathsf{data}) \to nm_{\bullet}t \to n_{\bullet}m_{\bullet}t \\
\texttt{split} &: (n : \mathsf{nat}) \to (ft : \mathsf{nat}{\to}\mathsf{nat}) \to (t : \mathsf{data}) \to \textstyle\sum_{i=0}^{n} ft(i)_{\bullet}t \to n_{\bullet\bullet}i \mapsto ft(i)_{\bullet}t \\
\texttt{join} &: (n : \mathsf{nat}) \to (m : \mathsf{nat}) \to (t : \mathsf{data}) \to n_{\bullet}m_{\bullet}t \to nm_{\bullet}t \\
\texttt{join} &: (n : \mathsf{nat}) \to (ft : \mathsf{nat}{\to}\mathsf{nat}) \to (t : \mathsf{data}) \to n_{\bullet\bullet}i \mapsto ft(i)_{\bullet}t \to \textstyle\sum_{i=0}^{n} ft(i)_{\bullet}t \\
\texttt{transpose} &: (n : \mathsf{nat}) \to (m : \mathsf{nat}) \to (T : \mathsf{data}) \to n_{\bullet}m_{\bullet}T \to m_{\bullet}n_{\bullet}T \\
\texttt{asDepArray} &: (n : \mathsf{nat}) \to (t : \mathsf{data}) \to n_{\bullet}t \to n_{\bullet\bullet}i \mapsto t \\
\texttt{liftNat} &: \mathsf{int} \to (T : \mathsf{data}) \to ((n : \mathsf{nat}) \to T) \to T \\
\texttt{which} &: (n : \mathsf{nat}) \to (m : \mathsf{nat}) \to n_{\bullet}\mathsf{bool} \to m_{\bullet}\mathsf{idx}[n]
\end{aligned}$$

Figure 3. Common data-parallel primitives. Type variables in curly braces are implicit and inferred during type inference.

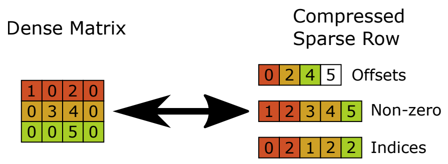

Figure 4. Visual representation of conversion between dense and sparse matrix formats

applies the type equality established in Figure 1c during type checking.

2.2 Data Parallel Primitives

Figure 3 shows some of the data-parallel primitives supported in our language, which are quite common in array programming languages. The **map** primitives exists in two forms. One applies a function to each element of an ordinary array, the other applies a dependent function to each element of a position-dependent array. **fold** reduces an array to a single value, by pairwise combination starting on the left. **split** and **join** introduce and remove an additional dimension to an array, and are also supported for position dependent arrays. **asDepArray**interprets an ordinary array as a position dependent array. The **liftNat** primitive is used to scope an int, lifting it as a type level nat. Finally, **which**, returns the indices of the first m true elements in an array of bool, defaulting to index 0 if m exceeds the input's length.

3 Language Extensions

3.1 Limitations

Prior work[14] has demonstrated how to express irregular data structures, as shown in the following code, implementing triangular matrix-vector multiplication. The matrix is encoded as a *position dependent array* with type $n_{\bullet\bullet} i \mapsto i{+}i_{\bullet}\mathrm{f32}$. n indicates the number of rows in the matrix; $i+1$ expresses the triangular shape. The index i ranges from 0 up to n, so the first row has one element, the second row has two, etc.

```
TmVM (n:nat) => (matrix:n..i->i+1.f32) =>
 (vector:n.f32) = matrix
    |> map(idx => row =>
        let v = take (idx+1) vector
        zip(row, v) |> map(*) |> fold (0, +))
```

However, it is not clear how to express programs operating over sparse data structures. For example, consider the conversion between dense matrices and sparse matrices, expressed in the *CSR* format, focusing on the type signature of each operation. Figure 4 illustrates the formats and conversion.

We start by transforming a *sparse* matrix to a *dense* one. How can the operation's type be expressed?

Each component of the *CSR* matrix could be modeled as a separate parameter: the row offsets as an array of int, and the data and column indices as an array of pairs. However, it is impossible to write out the second array's inner size:

$$\kappa ::= \ldots \mid \mathsf{nats} \mid \mathsf{nats}{\to}\mathsf{data} \mid \mathsf{nats}{\to}\mathsf{nat}$$

(a) Extension to Figure 1a

$$\frac{\Delta \vdash Ns : \mathsf{nats} \quad \Delta \vdash N : \mathsf{nat}}{\Delta \vdash Ns@N : \mathsf{nat}}$$

(b) Extension to Figure 1d

Figure 5. Extensions to Figure 1

$\texttt{csr2dense} : (n : \mathrm{nat}) \to (m : nat) \to n.int \to (n_{\bullet}?_{\bullet}\mathrm{f32} \times \mathrm{idx}[m]) \to n_{\bullet}m_{\bullet}\mathrm{f32}$

As the inner dimension ? varies across rows, a regular array is not sufficient. Instead, a position dependent array is necessary. We introduce some unknown nat→nat function f mapping the index of each row to its length:

$\texttt{csr2dense} : (n : \mathrm{nat}) \to (m : \mathrm{nat}) \to n_{\bullet}\mathrm{int} \to (n_{\bullet\bullet}(i \mapsto f(i)_{\bullet}(\mathrm{f32} \times \mathrm{idx}[m]))) \to n_{\bullet}m_{\bullet}\mathrm{f32}$

What should f be? From the definition of *CSR*, $f(i)$ should be the ith row length, computed from the offsets array. But how can this operation be modeled at the type level?

3.2 Nats

The crucial observation is that, while an int can be expressed at the type level as nat, there is no such equivalent construct for an *array* of int, which is being used in the running example to store the *CSR* row offsets.

Figure 5a introduces the new kind nats, representing a type-level array of nat, together with an indexing operation. We also extend the language with various supporting constructs: dependent functions $(ns : \mathrm{nats}) \to \theta$, type-level functions nat→data and nats→nat, and lifting from an array of int via the **liftNats** primitive:

$\texttt{liftNats} : (n : nat) \to \{T : \mathrm{data}\} \to n_{\bullet}\mathrm{int} \to ((m : \mathrm{nats}) \to T) \to T$

The type of **csr2dense** can now be expressed. The $ns@(i+1) - ns@i$ formula in the inner array models the *csr* format: the length of each sparse row is given by subtracting the start and end offset, as provided in the metadata array.

$\texttt{csr2dense} : (n : \mathrm{nat}) \to (m : \mathrm{nat}) \to (ns : \mathrm{nats}) \to n_{\bullet\bullet}(i \mapsto (ns@(i+1) - ns@i)_{\bullet}(\mathrm{f32} \times \mathrm{idx}[m])) \to n_{\bullet}m_{\bullet}\mathrm{f32}$

Let us now try to write the type of the inverse transformation, **dense2csr**. The goal is thus to produce a sparse matrix by attempting to return a *pair* containing both the sparse matrix's row offsets and non zero elements:

$\texttt{dense2csr} : (n : \mathrm{nat}) \to (m : \mathrm{nat}) \to n_{\bullet}m_{\bullet}\mathrm{f32} \to (\mathrm{nats}, n_{\bullet\bullet}(i \mapsto (?@(i+1) - ?@i)_{\bullet}(\mathrm{f32} \times \mathrm{idx}[m])))$

Once again, we run into type system limitations. Intuitively **?** should refer to the nats stored in the first element of the *pair*, but we have no way to express this dependence. The previous example uses a *dependent function* to express the dependency within components. But this is only possible when taking inputs, not when producing outputs.

$$\frac{\Delta, n : \kappa \vdash T : \text{data} \quad \kappa \in \{\text{nat}, \text{nats}\}}{\Delta \vdash (n : \kappa ** T) : \text{data}}$$

Figure 6. Dependent Pair Type

We can address the issue by extending the language with *dependent functions*'s dual construct: the *dependent pair*

3.3 Dependent Pair Type

A dependent pair is a pair wherein the second element's type may depend on the first element's value. Figure 6 shows the relevant type formation rules. The language only supports nat and nats dependent pairs. This limitation guarantees the efficient implementation of dependent pairs.

Dependent pairs are used to express data structures tagged with their size at runtime. For example, the dependent pair $(n : \text{nat} ** n_\bullet \text{f32})$ models an array together with its length, potentially dynamically computed at runtime.

We can now express the type of `dense2csr` by using the dependent pair to refer to the matrix's row offsets ns:

$$\texttt{dense2csr} : (n : \text{nat}) \to (m : \text{nat}) \to n_\bullet m_\bullet \text{f32} \to$$
$$(ns : \text{nats} ** \ n_{\bullet\bullet}(i \mapsto (ns@(i+1) - ns@i)_\bullet(\text{f32} \times \text{idx}[m])))$$

3.4 Primitives Operating with Dependent Pairs

Adding *dependent pairs* also requires the introduction of new primitives for their construction, destruction, and transformation. We give a brief description of each addition.

makeDepPair. This primitive serves as the constructor for dependent pairs. It, like most of the newly introduced primitives, comes in two version: `makeNatDepPair` for building nat-dependent pairs, and a `makeNatsDepPair` for building nats-dependent pairs.

$$\texttt{makeNatDepPair} : \ (f : \text{nat}{\to}\text{data}) \to (n : \text{nat}) \to$$
$$f(n) \to (k : \text{nat} ** \ f(k))$$
$$\texttt{makeNatsDepPair} : \ (f : \text{nats}{\to}\text{data}) \to (ns : \text{nats}) \to$$
$$f(ns) \to (ks : \text{nats} ** \ f(ks))$$

As the two versions are identical save for the nat/nats distinction, we refer to the first one only for brevity in the following discussion.

`makeNatDepPair`'s first argument $f : \text{nat}{\to}\text{data}$ is a type-level function that, given any nat, produces a data type, potentially dependent on the given nat. f is applied to the second argument $n : \text{nat}$, yielding the type of third argument $f(n)$: an instance of the data type generated by f. The return type is the constructed *dependent pair* $(k : \text{nat} ** f(k))$. The newly bound k has the same value as n, and the second element $f(k)$ has the same value as $f(n)$. Crucially, this rebinding ensures that the returned value no longer mentions an external n: the dependence is entirely hidden and contained within the dependent *dependent pair*.

matchDepPair. `matchNatDepPair` serves as a projection for dependent pairs, granting access to the elements. It also comes in both a nat and a nats version. Just as before, for conciseness, we only examine the first version.

$$\texttt{matchNatDepPair} : \ (f : \text{nat}{\to}\text{data}) \to (T : \text{data}) \to$$
$$(n : \text{nat} ** \ f(n)) \to ((k : \text{nat}) \to f(k) \to T) \to T$$
$$\texttt{matchNatsDepPair} : \ (f : \text{nats}{\to}\text{data}) \to (T : \text{data}) \to$$
$$(ns : \text{nats} ** \ f(ns)) \to ((ks : \text{nats}) \to f(ks) \to T) \to T$$

`matchNatDepPair`'s first argument is again the type level function $f : \text{nat}{\to}\text{data}$, followed by the data type T. The third parameter, $((k : \text{nat}) \to f(k) \to T)$, is a function computing a value of type T from the individual components of the *dependent pair*. This function therefore implements the body of the match. The behavior of `matchNatDepPair` is to deconstruct the pair, pass the elements individually to f, and return its result.

There is no risk of the matching function leaking the dependent element: since the type T is defined outside k's scope, T is guaranteed to not depend on k. The only way to propagate a result dependent on k is therefore to construct a new dependent pair within the matching function.

reduceToNat. Using `makeNatDepPair` and `matchNatDepPair` , it is possible to express arbitrary transformations of a *dependent pair*'s second element. However, it is not clear how one can transform the first element. Replacing the first element with a different value requires a corresponding change in the second element's type to preserve the dependence.

Deriving a general solution to this challenge is beyond the scope of this paper. We are however interested in providing some support for such transformations. In particular, there are cases in which the dependence on a nats can actually be reduced into dependence to a nat.

$$\texttt{reduceToNat} : \ \{f : \text{nats}{\to}\text{nat}\} \to \{T : \text{data}\} \to$$
$$(ns : \text{nats} ** \ f(ns)_\bullet T) \to (k : \text{nat} ** \ k_\bullet T)$$

The primitive evaluates the function $f(ns)$ and binds it to k, the first element of a new dependent pair. In the second element, $f(ns)$ is then substituted with k. The intuition here is that the dependence is "too broad", and only a single nat, the value of $f(ns)$, is sufficient as the second array's size.

This transformation is desirable if possible, because the resulting nat-dependent pair is potentially much smaller than the nats-dependent one. Note that the second element changes only in type but not in value. We will further discuss the utility of this primitive in sections 4.3 and 6.3.

4 Case Studies

This section presents three case studies to better motivate and illustrate the contributions of this work. Each case illustrates a potential use of *dependent typing* for expressing programs operating with *irregular data structures*. The three case studies analyzed are: *dense matrix to CSR matrix conversion*, *sparse matrix vector multiplication* and *breadth first search's parallel frontier expansion.*

4.1 Dense to CSR Matrix Conversion

4.1.1 Overview. Dense matrices are commonly used in the context of linear algebra and graph applications. They are traditionally implemented as a contiguous array of values, interpreted by the applications as a logical multi-dimensional grid. In many applications, however, a large proportion of the entries of a matrix are zero-valued. In these cases, the use of a *sparse matrix format* drastically reduces the storage needs, the number of operations performed, and thus leading to more efficient and better-performing implementations.

Sparse matrices are commonly implemented by using both a flat array storing the non-zero elements and additional metadata — often arrays of indices, offsets, lengths, etc — encoding the logical structure of the matrix. Most linear algebra packages include conversions between *dense* and *sparse* matrix formats.

4.1.2 Parallel Implementation. In Section 3.2, we have discussed a possible type signature for a **dense2csr**: a function converting a dense matrix into a sparse one. The generated sparse matrix has the type:

type csr[n,m] =
$(\mathit{offs} : \mathsf{nats} ** \ n_\bullet(i \mapsto (\mathit{offs}@(i+1) - \mathit{offs}@i)_\bullet(\mathsf{f32} \times \mathsf{idx}[m])))$

Let us discuss a parallel GPU implementation of **dense2csr**. Internally, **dense2csr** is split in two GPU kernels. The first calculates the array of row offsets, and then the second performs the compression of the matrix into the *CSR* format.

computeOffsets : $(n : \mathsf{nat}) \to (m : \mathsf{nat}) \to$
$n_\bullet m_\bullet \mathsf{f32} \to (n+1)_\bullet \mathsf{f32}$

compressMatrix : $(n : \mathsf{nat}) \to (m : \mathsf{nat}) \to$
$n_\bullet m_\bullet \mathsf{f32} \to (n+1)_\bullet \mathsf{f32} \to \mathit{csr}[n, m]$

computeOffsets is implemented as a parallel scan over the number of non-zero elements in each row. While there are several well-known low-level implementations of parallel scan on a GPU[7], giving an efficient functional-style formulation is outside the scope and topic of this paper. We instead focus on the second step, **compressMatrix**, which makes use of the primitives and type extensions introduce here and constitutes the core of this first case study.

Listing 1 shows the relevant code. The implementation assumes the dense matrix to be in the column-major format, as this is the formulation used by reference state-of-the-art libraries like cuSparse. The program can be adapted to row-major by simply removing the **transpose** call at line 4 and then adjusting the input type accordingly.

The program begins by lifting the array *offsets* to the type level (line 3). Lines 6 compresses each row in the matrix: the number of non-zero values is calculated from the offs array

```
1  def compressMatrix (n:nat) => (m:nat) =>
2   (offsets:n+1.nat) => (dense:m.n.f32) =>
3   liftNats(offsets)((offs: nats) =>
4    dense |> transpose
5          |> asDepArray
6          |> map(i => r => compressRow(offs,i,r))
7    makeNatsDepPair(offs)(sparse)
8
9  def compressRow offs rowIdx row =
10     let length = offs@(rowIdx+1)-offs@rowIdx
11     row |> map(x => x != 0.0)
12         |> which(length)
13         |> map(idx => (row@idx,idx))
```

Listing 1. Implementation of the compression phase of **dense2csr**

at line 10, and the which primitive is used to find their indices at lines 11-12. Line 13 extracts the sparse row, consisting of the non-zero value and its column index.

The resulting array, now dependent on *offs*, and thus a dependent pair is used to combine the two values (line 7), encapsulating the dependence.

While this expression may look complex, it is implemented by generating efficient code, as demonstrated later in 5.4

4.2 CSR Sparse Matrix Vector Multiplication

The previous case study has shown how to express the generation of a sparse data structure. Conversely, this case study depicts the consumption of a sparse data structure, by implementing the classic linear algebra algorithm parallel *sparse matrix-vector multiplication* (SpMV).

Our choice of type representation for the *CSR* matrix is identical to that used in the **dense2csr** case study. The whole *sparse matrix vector* operation therefore has type:

csrSpMV1 :
$(n : \mathsf{nat}) \to (m : \mathsf{nat}) \to \mathit{csr}[n, m] \to m_\bullet \mathsf{f32} \to n_\bullet \mathsf{f32}$

There is, however an alternative formulation, which does not use dependent pairs (as a reminder, *csr* is a type alias for a dependent pair). In this second version, the sparse matrix is expressed via two separate parameters, one for the row offsets, the other for the array of non-zero values and column indices. This representation is similar to that used in prior work [15] and in the **csr2dense** example covered in 3.1.

csrSpMV2 : $(n : \mathsf{nat}) \to (m : \mathsf{nat}) \to (\mathit{offs} : \mathsf{nats}) \to$
$(n_\bullet i \mapsto (\mathit{offs}@(i+1) - \mathit{offs}@i)_\bullet(\mathsf{f32} \times \mathsf{idx}[m])) \to$
$m_\bullet \mathsf{f32} \to n_\bullet \mathsf{f32}$

Implementation. The two implementations are remarkably similar – *SpMV1* can be straightforwardly implemented in terms of *SpMV2*

The source code for the core of the algorithm is given in listing 2, and is similar to common implementations of *dense matrix vector multiplication*: the matrix's rows are processed

```
1  def csrSpMV1 (n:nat) => (m:nat) =>
2   (matrix: csr[n,m]) => (vector: m.f32) =
3   matchNatsDepPair(matrix)(offs => data =>
4    csrSpMV2(n,m,offs,data,vector))
5
6  def csrSpMV2 (n:nat) => (m:nat) =>
7  (offs:nats) =>
8  (nnz:n..i->(offs@(i+1))-offs@i.(f32×idx[m]))=>
9  (vector:m.f32) =
10  nnz |> map(row => dotProduct(row,vector))
11
12 def dotProduct row vector =
13     row |> map (i => x => x*vector@i)
14         |> fold(0, +)
```

Listing 2. Implementations of sparse matrix vector multiplication (SpMV). Version 1 models the matrix with a dependent pair while version 2 uses multiple arguments

in parallel using the **map** primitive (line 10). For each row then, the dot product is implemented by a **map** and then a **fold** over pairs of matrix and vector elements (lines 13-14).

The advantage of using dependent pairs is the ability to express the whole sparse matrix as a single value, rather then splitting it across a number of parameters. There are also low-level differences concerning data representation that may have a measurable performance impact. A *dependent pair* is stored in memory as a single, contiguous buffer, while the two separate parameters make no specification on their relative position. These differences and the conditions under which a *dependent pair* need to be stored in memory are discussed in detail in section 5.1. The magnitude of these effects is presented in section 6.2

4.3 Breadth-First Search Frontier Expansion

Breadth-first search is a fundamental graph algorithm, with several well known parallel implementations in the context of General Purpose GPU programming. As a graph algorithm, *BFS* operates over inherently irregular data structures, while also being both relatively simple and well known.

Our implementation is a version of *parallel breadth-first search* [12]. It makes use of a *frontier*: an array containing the nodes to be explored next. At each iteration, the *frontier expansion* is computed, replacing each node with its children nodes. A second data structure tracks which nodes have been already seen. These are discarded from the next frontier.

Implementations may include other application-specific processing, such as recording the path traversed to a node, or updating a distance metric, etc. For simplicity, we omit these additional steps.

Figure 7 illustrates the graph representation used throughout the case study. A graph is modeled similarly to a sparse matrix: an array of offsets specifies for each node a slice in a second array, containing the edges of that node. The edges are then represented by storing the index of the child

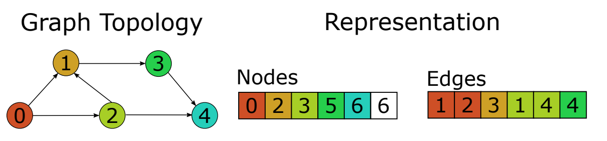

Figure 7. Graph representation used for the BFS algorithm implementation

node. The two-dimensional structure of the edges array is expressed with a *position dependent array*, stored together with the nodes within a *dependent pair*:

$$\texttt{type graph[n] =}$$
$$(\text{nodes} : \text{nats} \ast\ast\ n_{\bullet\bullet}(i \mapsto (\text{nodes}@(i+1) - \text{nodes}@i)_{\bullet}\,idx[n]))$$

4.3.1 Parallel Implementation.

We consider a parallel implementation of the *BFS* algorithm for GPUs, focusing solely on the computation of the next frontier, as it is a small use case sufficient for our purposes.

This is a GPU-oriented implementation subject to the considerations mentioned in 4.1.2. At its core, the frontier expansion phase copies a subset of each node's children from the input frontier to the output. Since the number of children extracted from each node is not statically known, issues arise on how to perform all these writes in parallel into the same buffer without causing data races.

A pessimistic naive solution is to write, for a graph of known max degree d, each ith node children starting at offset $d * i$, leaving gaps in the output buffer. This solution is highly inefficient, as the max degree d can be very high. This is particularly true on the very commonly encountered power-law graphs, where a few nodes have an exceedingly high degree, but most nodes a very low one. The large memory consumption renders the approach completely impractical for even moderately sized graphs.

A better solution is to split the calculation in two steps. The first step, **count**, computes for each node in the previous iteration frontier, the number of children not yet seen. Rather than just returning these lengths, **count** produces their prefix sum, to compute each frontier's *writeOffsets*.

$$\texttt{count}: \quad (n : \text{nat}) \to \text{graph}[n] \to (n.\text{bool}) \to$$
$$(m : \text{nat}) \to m_{\bullet}idx[n] \to m + 1_{\bullet}\text{int}$$

The second step, **computeFrontier**, iterates again over the input frontier, extracts the unseen children, and then writes them out to the output frontier, using the *writeOffsets* to determine each node's output slice in parallel. The slices are contiguous in the output buffers, while avoiding data races.

$$\texttt{computeFrontier}: \quad (n : \text{nat}) \to \text{graph}[n] \to (n_{\bullet}\text{bool}) \to$$
$$(m : \text{nat}) \to m_{\bullet}idx[n] \to$$
$$(m + 1)_{\bullet}\text{int} \to (k : \text{nat} \ast\ast\ k_{\bullet}idx[n])$$

Just as before, we omit the parallel scan implementation and focus on the implementation of **computeFrontier**, whose

```
1  def computeFrontier (n:nat)=>(gs: graph[n])=>
2   (frontier: m:nat ** m.idx[n])=>
3   (writeOffsets:(m+1).int) => (seen:n.bool) =
4   liftNats(writeOffsets)(writeOffsets =>
5   matchNatsDepPair(gs)(
6   nodes => edges =>
7   matchNatDepPair frontier (
8    fLen => fElems =>
9    let nextFrontier =
10    fElems
11     |> asDepArray
12     |> map(i => nodeId =>
13      let expected = (writeOffsets@(i+1)-
             writeOffsets@i)
14      let children = edges @ nodeId
15      filterSeen(seen,expected,children)
16     |> join
17   makeNatsDepPair(writeOffsets)(nextFrontier)
18     |> reduceToNat))))
19
20 def filterSeen seen expected nodes =
21  nodes |> map(node => !(seen@node))
22        |> which (expected)
23        |> map(i => nodes @ i)
```

Listing 3. Implementation of the next frontier computation step of parallel breadth first search

code is presented in listing 3. Lines 4 to 7 access the graph and frontier, and lifting the *writeOffsets* to the type level.

We then map over each frontier node (line 12), and extracting the unseen children (line 15). At this point, the intermediate next frontier result as type:

$$\mathit{fLen}_\bullet(i \mapsto (\mathit{writeOffsets}(i+1) - \mathit{writeOffsets}@i)_\bullet \mathrm{idx}[\mathrm{n}])$$

We now flatten this two dimensional structure by applying the **join** primitive (line 16), resulting in a single-dimensional array of type:

$$(\textstyle\sum_{i=0}^{\mathit{fLen}-1}(\mathit{writeOffsets}@(i+1) - \mathit{writeOffsets}@i))_\bullet \mathrm{idx}[n]$$

Using the **reduceToNat** at line 18, the summation is evaluated and the result bound to the fresh variable k, therefore discarding the cumbersome dependence on writeOffsets.

$$\texttt{before}:\ (\mathit{writeOffsets} : \mathrm{nats} \ast\ast \\ (\textstyle\sum_{i=0}^{\mathit{fLen}-1}(\mathit{writeOffsets}@(i+1) - \mathit{writeOffsets}@i))_\bullet \mathrm{idx}[n])$$

$$\texttt{after}:\ (k : \mathrm{nat} \ast\ast\ k_\bullet \mathrm{idx}[n])$$

This transformation may seem potentially expensive, as it involves computing the value of the summation. In practice, the compiler can optimized it away by inferring that the value of k is simply given by *writeOffsets @ (fLen + 1)* — ultimately implemented as an array lookup. The optimization relies on the symbolic algebra system included in the compiler. Further details of this optimization process are given in the *reduceToNat* paragraph of section 5.2

5 Compiler Implementation

In this section, we look at some implementation details and techniques used in the compiler to provide support to the language extensions proposed in section 3. We will then illustrate an example of code generation by stepping through the compilation of a simple example program.

5.1 Dependent Pair Representation

In the generated low-level code, *dependent pairs* are represented in a number of ways. There are two main representations: a *materialized* and a *non-materialized* representation. The *materialized* representation is used when a dependent pair crosses a GPU kernel boundary. It consists of storing the dependent pair components contiguously in a memory buffer, which can then be referred via a single handle. Being self-contained, it optimizes for simplicity of transport.

The *non-materialized* representation is used for dependent pairs that are defined within a GPU kernel and are not part of the output. It is the preferred representation unless materialization is needed. It consists of having two independent buffers for each element of the dependent pair. This allows the lazy construction and access of intermediate dependent pairs, eliminates unnecessary memory allocations and copies, and allows for the fusion of other lazy operations accessing the dependent pair contents. The tracking of separate buffers is entirely static, and therefore this is a zero-cost representation, optimizing for execution performance.

5.2 Primitives Implementation

As detailed in section 3, we extend the base language with several primitives. The previous sections have covered these primitive's type signature and semantics. We now cover the new primitive's low-level implementation.

mkDPair. This primitive constructs a dependent pair dependent from individual components.

If the generated pair is part of the program's output, it will have to be *materialised.* In this case, *mkDPair* requests for a sufficiently large buffer to be allocated (see section 5.3 for a discussion of memory allocation on GPGPU targets), copying the components in parallel.

Alternatively, if the resulting pair does not appear in the program's output, then the lazy *non-materialised* representation is used. In this case, no code is generated for the primitive.

dmatch. This primitive deconstructs a dependent pair, giving access to each element individually. If the argument is *materialized*, the compiler generates the two element pointers from the backing dependent pair buffer, and passes them to the matching function. If the pair is *not-materialized*, then the call is optimized away, the individual element pointers are already available for use.

reduceToNat. This primitive allows shrinking a *nats dependent pair* to a *nat dependent pair*, in cases where the second component depends only on some reduction of the first. If possible, the generated dependent pair is *non-materialized*.

The uniqueness of this primitive lies in the fact that it does not express a value-level computation, but rather a type-level one — namely the evaluation of a f : nats→data function into a single nat. This is performed by simplifying the function f using the compiler symbolic algebra engine and then generating the target code implementing any remaining computation.

To show the full power of the system, we walk through an example from section 4.3. A dependent pair with type:

$(ns : \mathsf{nat} ** \ \sum_{i=0}^{n-1}(ns@(i+1) - ns@i)_{\bullet}node)$

is passed to **reduceToNat**. The dependent pair is not materialized, and is stored in two separate buffers, of type (using C notation) int ∗ fst and node ∗ snd.

The symbolic algebra engine simplifies the summation in the second element's type into an efficient closed form, by using the algebraic rule:

$$\sum_{i=0}^{N}(f(i+c) - f(i)) = f(N+c) - f(0) \qquad (1)$$

Using the rule, the array size is simplified to $ns@n - ns@0$. The compiler generates the equivalent C Code fst[n] − fst[0]. A second optimization, whose details are omitted here, determines that fst[0] = 0, eliminating the subtraction.

Ultimately, the call *reduceToNat* is entire a type level operation and is thus optimized away: as the resulting dependent pair is *non-materialized*, accesses to the first element are redirected to fst[n], and accesses to the second element to node ∗ snd: the original second element buffer.

5.3 Memory Allocation

GPGPU programming platforms offer limited support for dynamic memory allocation. The programming model requires that memory allocation is performed ahead of kernel launch. This preserves a modicum of dynamicity: the host-side driver application, responsible for orchestrating the computation, has access to the kernel inputs and can inspect them to compute appropriate memory buffer sizes. However, no allocation can be performed during kernel execution.

The lack of dynamic memory allocation constitutes one of the main implementation challenges: dependent pairs are used to model data structures with dynamic size, but dynamically allocating memory for the structure is not possible within the execution of program.

The problem is addressed by using the language's type system, which tracks the length of arrays, the only type with variable size, with nat symbolic expressions. The compiler uses the symbolic simplification engine to calculate the upper bound of each expression, determining a minimal conservative size for each memory buffer.

```
1  def example (n:nat) => (numChunks:nat) =>
2  (data:n.int) => (offsets:(n+1).int) =
3   liftNats(offsets)(offs =>
4    let filtered_data = data
5     |> split(n/numChunks)
6     |> asDepArray
7     |> map(i => chunk =>
8      chunk
9       |> map(x => x != 0)
10      |> which(offs @ (i+1) - offs @ i)
11      |> map (idx => chunk @ idx))
12
13   makeNatsDepPair(offs, filtered_data)
14 ) |> matchNatsDepPair(offs => data =>
15       makeNatsDepPair(offs, join data))
16   |> reduceToNat
```

Listing 4. High level source for listing 5

```
1  void example(int* output, int n, int numChunks,
2               int* data, int* offs) {
3   parallel for (int chunkId = 0;
4                 chunkId < num_chunks;
5                 chunkId++) {
6   int indices[offs[i+1] - offs[i]];
7   int idx_write = 0;
8   for(int i 0; i<offs[i+1]-offs[i]; i++) {
9    if (data[chunkId*(n/numChunks)+i] != 0) {
10    indices[idx_write] = i;
11    idx_write++; }}
12  int* out = output + (1+offs[i+1]-offs[i]);
13  for (int i = 0; i < idx_write; i++) {
14    out[i] = data[indices[i]];
15  }
16  output[0] = offs[n];
17  } }
```

Listing 5. Idealised parallel C code generated by compiling listing 4

5.4 Low Level Code Generation

We now walk through an example showing how all the features described so far are combined together into an efficient implementation. For simplicity, the target language will be an idealized C with parallel loops.

The program is a simplified version of **csr2dense**, covered in section 4.1.2. Given an array of integers *data*, it returns an array of dynamic size containing the non-zero elements. It employs a parallelization strategy similar to that used in the case study: the input array is split in a number of chunks processed in parallel, and we expect to be given an array of the expected output offsets for each chunk, which is assumed to be computed in some other kernel.

The source program is found in listing 4, the generated code in listing 5. Listing 4:1-2 defines the kernel's signature. We can see in listing 5 that the parameter arrays are represented as flat buffers.

Listing 4:3 uses **liftNats**: in the scope of the supplied function, the array *offset* is visible at the type level as *off* : nats. This is a type-level operation and generates no C code.

In listing 4:5-6, the *data* array is the split in chunks and transformed into a *position dependent array*. As all these operations are lazy, no C code is generated.

In lines 7-11, the array is transformed by a *position-aware map*. This operation corresponds to the parallel for loop defined in listing 5:3-5. The iteration bound is derived from the outer array dimension, which is stored in *numChunks*.

Listing 4:9-10 filter away the zeros from the chunk. This generates listing 5:6-11. The computation's overall shape is derived from the **which** pattern, but the iteration bounds, data accesses, and filtering predicate are derived from the lazy **split** and **map** preceding it.

Listing 4:11 generates listing 5:12-15. This is the copy of the filtered non zero values to the output buffer. Iteration bounds and ranges are derived from type information.

Finally, listing 4:13-16 first construct a dependent pair packing together the offsets and the two dimensional filtered away. The pair is then matched apart, and flattened via **join**, before being finally passed to **reduceToNat**. The operations are heavily optimized, being compiled to a single scalar assignment in listing 5:16.

Listing 4 uses a somewhat verbose style to showcase the range of optimizations supported. The intermediate dependent pair crated in in listing 4:13 is optimized away. The **split** on line 15 'cancels out' the effects of the **split** at line 5, and **reduceToNat** is optimized away into a simple lookup as shown in section 5.2.

6 Evaluation

This section presents the results of several experiments, each dedicated to evaluating one aspect of this paper's case studies. To show the competitiveness of our approach, we compare our dense matrix to *CSR* conversion kernel with an equivalent implementation provided by cuSparse, a highly tuned library supplied by Nvidia.

We then compare the runtime of two *CSR* sparse matrix vector product implementations, one using dependent pairs to model its sparse matrix input, the other using the destructured representation previously used in [15].

Finally, we provide measurements for two implementations of BFS's *next frontier* step: a naive parallel implementation, compared to a more optimized implementation, illustrating the process of deriving the former into the latter.

Experimental Setup. All experiments are performed using the compiler's OpenCL CUDA backend, targeting version 1.2 of the standard, driver version 10.2.185. All experiments are run on an NVidia GeForce GTX 1070. For comparison purposes, we run kernels from the cuSparse library provided by the CUDA 11 suite. Each run-time number shown is computed from the median of 100 consecutive runs to measure

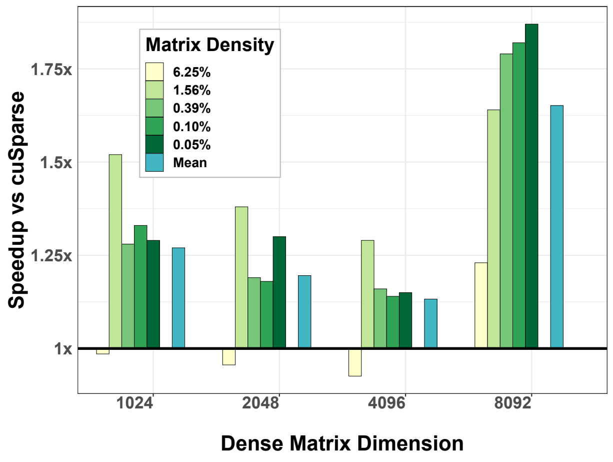

Figure 8. Relative performance of our implementation of *dense to sparse CSR matrix conversion* vs cusparseSdense2csr. Higher is better.

the GPU's steady-state performance. All execution times refer to GPU computation time only.

6.1 Dense to Sparse Conversion

The first experiment we present compares our parallel version of *dense to CSR* matrix conversion with an equivalent implementation provided by the cuSparse library, in the form of the cusparseSdense2csr library call.

cusparseSdense2csr is internally implemented with a number of kernels, as reported by the CUDA profiler *nvprof*. A single call to cusparseSdense2csr includes both the row offset computation step via parallel scan and the matrix construction step yielding a full *CSR* matrix. All the numbers presented include both the time of executing a prefix sum generating the offsets for both our implementation and the cusparseSdense2csr reference. Note that the row offset computation step constitutes a very small fraction of the overall application runtime — well under 1%.

Figure 8 shows the relative performance of our implementation taking cusparseSdense2csr as a baseline, across a number of matrix sizes, and densities ranging from 6% to 0.03%. The matrix density refers to the proportion of nonzero elements present in the dense matrix. The results show that our implementation outperforms the reference for most combinations of matrix density and size, with an average of 1.2× speedup across matrix sizes.

As we do not have access to cusparsSdense2csr's implementation, it is not possible to further analyze the differences between implementations. Previous work has shown that the optimizations enabled by data-parallel functional GPU languages can be used to produce comparable or outperforming implementations of state-of-the-art library versions [6, 15], substantiating our claim the extending expressiveness of

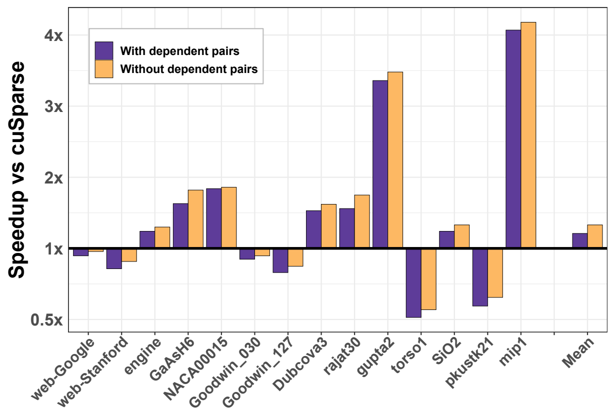

Figure 9. Performance of *spmv* exrpessed with dependent pairs and without dependent pairs *CSR* matrix relative to *cusparseSpMV*. Higher is better.

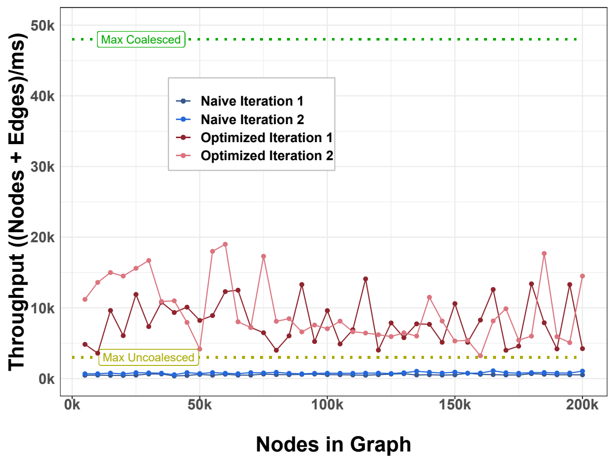

Figure 10. Throughput comparison for iteration 1 and 2 of naive and optimized versions of the BFS kernel. Included for reference are the GPU theoretical throughput limits for coalesced and non-coalesced memory accesses.

such languages, while carefully avoiding the introduction of unnecessary overhead, proves to be a fruitful endeavor.

6.2 Sparse Matrix Vector Multiplication

The second experiment measures the performance of our implementation of sparse matrix-vector multiplication, as compared with cuSparse's csrSpmV library implementation. The code for our implementation is largely based on the case study presented in section 4.2.

Moreover, the experiment is also used to measure the runtime overhead of using dependent pairs for representing sparse data structures by comparing the performance differences of the two versions of *sparse matrix-vector multiplication* presented in section 4.2. The first version accepts a dependent pair as input and therefore uses the *materialized* representation, while the second version, taking two separate arguments, one for each component, is equivalent to employing a *non-materialized representation.*

Figure 9 shows the relative performance of the two versions, in relation to the cuSparse baseline, using as input a selection of real-world sparse matrices commonly used in the literature. The matrices are sourced from the Suite Sparse Matrix Collection[4]. The cuSparse baseline is offered primarily to highlight the general efficacy of our approach, and to contextualize the impact of the performance loss incurred by using dependent pairs to express entire data structures.

The result shows that dependent pair materialization introduces a small overhead in most cases, but does not have a significant effect on the overall performance of the SpMV implementation, which remains competitive with cuSparse's, outperforming it by 1.2× on average vs 1.3× for the non-materialized case.

6.3 Breadth First Search Frontier Expansion

The last experiment concerns the third case study, the frontier expansion step of the Breadth-First Search. For this experiment, we generate random power-law graphs using the widely used Albert-Barabasi[1] model. Although the graphs are random, the model ensures that starting from one of the base-set's nodes, it is overwhelmingly likely that three iterations are sufficient to complete a full Breath First Search. We only report results for the first two iterations: the third iteration does very little work, as the graph is already almost entirely explored at this point. The experiment compares two implementations, termed *naive* and *optimized*, computing the next iteration frontier. Both versions roughly follow the description presented in section 4.3.

Naive Frontier Expansion. The *naive* adheres strictly to the case study version and is not concerned with implementing key GPU-specific optimizations. In particular, this implementation fails to take advantage of *memory coalescing*, an optimization that is fundamental to achieve high performance on the target GPU. Memory coalescing requires structuring the application data access patterns so that consecutive threads in a GPU *warp* access consecutive memory addresses. As a result, the *naive* version does not make efficient use of the GPU's available memory bandwidth.

Optimized Frontier Expansion. The BFS expansion algorithm can be modified to benefit from both *memory coalescing and increased parallelism levels.* The first step of the algorithm reads, for each node in the input frontier, all of its children. The *naive* version assigns individual threads to each node in the input frontier. This implies that all the children of any given node are read sequentially. Moreover, since consecutive threads read the children of different nodes, their memory accesses are scattered throughout the graph and cannot be reliably coalesced.

We improve on the *naive* implementation by writing a new version, taking advantage of well-known GPU programming techniques. We restructure the parallelism and memory access patterns of the program, splitting it into two separate executions: a *read* kernel and a *filter* kernel.

The *read* kernel's copies all the children of the frontier's node in a new buffer. Delaying the filtering allows for a straightforward access pattern. This simpler access pattern assigns an entire *warp* (a block of 16 consecutive threads) to each node, ensuring that reads are coalesced when possible

The *filter* kernel then performs the filtering of the already-seen nodes. As filtering a node's children is an inherently sequential process, this second step assigns a thread to each node, yields the next iteration's frontier.

Figure 10 compares the two versions, showing the throughput for the first and second iteration over several graphs. We also include the GPU's maximum theoretical throughput when performing coalesced and non-coalesced reads to contextualize the results. As we can see, the *naive* is consistently lower than the GPU's max non-coalesced read bandwidth, while the optimized version is always above the threshold.

7 Related Work

7.1 Data Parallel Functional Languages

Languages such as *Futhark* [8], *Rise* [5],*Lift* [18, 19], *Accelerate* [11] and *Single Assignment C* [17] provide a high level functional interface to GPU programming, greatly simplifying the task while generating correct and high performance code. They have extensive facilities for expressing programs that operate over dense and regular data structures, primarily modeled through multi dimensional arrays. These languages, however, do not provide a holistic solution to the problem of expressing irregular or sparse data structures.

7.2 Streaming Irregular Arrays

Streaming Irregular Arrays [3] are an extension to *Accelerate* which enable reasoning and programming with nested irregular arrays ergonomically and efficiently. They differ from our solution in several ways. Firstly, *streaming irregular arrays* uses data streams to model irregular data structures, while our work emphasizes the use of random-access arrays. Secondly, it provides language-level support to concepts specific to irregular data structures, such as *shapes* and *sequence computations*. Conversely, our work attempts to build such support from more general and orthogonal constructs such as dependent functions and pairs.

7.3 Dependently Typed Languages

Dependently typed languages, such as *Idris* [2] or *Agda* [13] or the earlier Epigram [10] possess powerful type systems, allowing the user to express complex invariants and are fully capable of modeling irregular data structures. This expressiveness is also an obstacle for efficient compilation. They are therefore unsuitable for high performance and numeric computing, and particularly GPGPU programming.

7.4 Domain Specific Compilers

An alternative trend to *data parallel functional languages* has been the development of domain-specific compilers, targeting popular application areas such as Halide [16] for image processing and TACO [9] for tensor algebra. These systems often support sparse and irregular data structures as required by each domain of interest. Their domain specific nature is the main difference from our approach, which offers a flexible solution valid in many different domains.

8 Conclusion

This paper has shown that by extending a data-parallel functional language with dependent typing facilities, it is possible to model irregular and sparse data structures, including *sparse matrices* and *graphs*. We explored these additions via three case studies implementing relevant parallel applications: *dense to sparse matrix conversion*, *sparse matrix vector multiplication* and *breadth first search frontier expansion*.

We detailed some of the most relevant implementation techniques necessary to implement the proposed extensions efficiently in the context of a highly parallel hardware platform, emphasizing memory allocation, the elimination of intermediate copies and symbolic simplification.

We have evaluated the approach through three experiments. We have demonstrated the general viability of the technique by comparing our implementations of *dense to sparse matrix conversion* and *sparse matrix vector multiplication* with the equivalent functionality provided by the highly optimized, state-of-the-art NVIDIA cuSparse library. We have demonstrated that the approach is competitive, and often outperforms cuSparse.

We have then analyzed the overhead of using some of the proposed additions, highlighting the importance of many of the optimizations internally provided by our compiler. Finally, we have shown the optimization process for parallel breadth first search, highlighting the importance of GPU specific techniques.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

References

[1] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Reviews of modern physics* 74, 1 (2002), 47. https://doi.org/10.1103/RevModPhys.74.47

[2] Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

[3] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. 2017. Streaming irregular arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, Iavor S. Diatchki (Ed.). ACM, 174–185. https://doi.org/10.1145/3122955.3122971

[4] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25. https://doi.org/10.1145/2049662.2049663

[5] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. https://doi.org/10.1145/3408974

[6] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization.* 100–112. https://doi.org/10.1145/3168824

[7] Mark Harris, Shubhabrata Sengupta, and John D Owens. 2007. Parallel prefix sum (scan) with CUDA. *GPU gems* 3, 39 (2007), 851–876.

[8] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 556–571. https://doi.org/10.1145/3062341.3062354

[9] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29. https://doi.org/10.1145/3133901

[10] Conor McBride. 2004. Epigram: Practical Programming with Dependent Types. In *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14-21, 2004, Revised Lectures (Lecture Notes in Computer Science, Vol. 3622)*, Varmo Vene and Tarmo Uustalu (Eds.). Springer, 130–170. https://doi.org/10.1007/11546382_3

[11] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 49–60. https://doi.org/10.1145/2500365.2500595

[12] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2 (2015), 14:1–14:30. https://doi.org/10.1145/2717511

[13] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures (Lecture Notes in Computer Science, Vol. 5832)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.). Springer, 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

[14] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. 2019. Position-Dependent Arrays and Their Application for High Performance Code Generation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing* (Berlin, Germany) *(FHPNC 2019)*. Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3331553.3342614

[15] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. 2020. Generating fast sparse matrix vector multiplication from a high level generic functional IR. In *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, Louis-Noël Pouchet and Alexandra Jimborean (Eds.). ACM, 85–95. https://doi.org/10.1145/3377555.3377896

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. https://doi.org/10.1145/2491956.2462176

[17] Sven-Bodo Scholz. 2003. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.* 13, 6 (2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[18] 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, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 205–217. https://doi.org/10.1145/2784731.2784754

[19] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 74–85. http://dl.acm.org/citation.cfm?id=3049841