Chapter 1

Extracting Data Parallelism

1.1 Introduction

SIMD instructions are ubiquitously available on most of the modern computer architectures. Extracting data parallelism using them requires close attention to the design of value types and algorithms that operate on them.

Value Type: A value type in a programming language is a datum and a set of context aware semantics associated with it. Most of the programming languages provide primitive value types such as `int`, `float`, `double` etc. and a way of creating compound value types that are composed of the primitive value types and/or other compound value types. In C++ a `struct` is used to create a compound value type. For example, value type for storing information of a book can be written in C++ as shown in listing 1.1. Memory layout of an instance of `book` is as shown in figure 1.1. Note the padding of two bytes at the end of the layout. Most compilers use padding to ensure self-alignment of instances of a `struct`. An array of a value type is a collection of instances of the value type in contiguous memory locations. Note that the array iteself can be regarded as a new value type. In C++, `std::vector<T>` is used to create such a type.

```
struct book {
  int id;
  float base_price;
  float price;
  char category[2];
};
```

Listing 1.1: Value type for storing information of a book

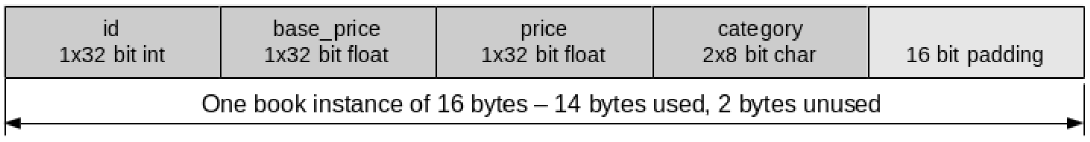


Figure 1.1: Memory layout of an instance of `book`

Data Parallel Algorithm: We will call an algorithm that performs data independent iterations over elements of an array or corresponding elements in two or more arrays as a data parallel algorithm. Following subsections describe how vectorization potential of such an algorithm is impacted by the design of a value type and its array.

1.1.1 Array Types for a Value Type

Array of a Struct (`AoS<T>`) Type

If instances of a value type `T` are stored in an array, it results in array of a struct type - `AoS<T>`. Consider the code in listing 1.2 that creates an array of 1000 instances of the book value type and updates price of each book instance in the array using the data parallel function `update_price(book_aos&,float)`. While this method of designing a value type and its array is natural to most of the programmers, it has following disadvantages when used in the context of a data parallel algorithm.

1. No vectorization: The loop in the `update_price(book_aos&,float)` function results into a scalar code since `price` and `base_price` fields of consecutive book instances are not stored in contiguous memory location. Note that some instruction sets (e.g. KNC and AVX512) provide SIMD instructions for loading and storing to memory locations with fixed strid, which can alleviate this issue. However, they still suffer from following two disadvantages.

2. Wastage of memory bandwidth: Since accesses to the `price` and `base_price` attributes result in loading of the entire cache line containing the attributes, the other values in the cache line, if any, are unused, resulting in wastage of memory bandwidth. In general, the amount of wasted bandwidth depends on the cache line size and memory layout of the struct that is operated on.

3. Wastage of space: Each book instance occupies 16 bytes in memory, of which 2 bytes are unused because of padding. Therefore, the amount of padded space grows linearly with the number of elements in the array.

```
typedef std::vector<book> book_aos;

void update_price(book_aos &books, float tax_rate) {
  const size_t count = books.size();
  float tax_mul = 1.0f+tax_rate/100.0f;

  // Data parallel loop that cannot be SIMD vectorized
  for(size_t i = 0; i < count; ++i) { // set price of each book to 10
    b[i].price = b[i].base_price*tax_mul;
  }
}
...
book_aos books(1000); // Creates AoS of 1000 books
// initialize objects in books

// Invoke data parallel function
update_price(books, 1000, 7.25f);
```

Listing 1.2: `AoS<book>` type

Structure of Attribute Arrays (`SoAA<T>`) Type

To achieve SIMD vectorization of the loop in the `update_price(book_aos&,float)` function, an attribute's values of consecutive book instances in the `AoS<book>` type should be stored in contiguous memory locations. Structure of attribute arrays (`SoAA<T>`) type can be used to achieve this. A `SoAA<T>` type can be derived from an `AoS<T>` type by converting each attribute of the type `T` to an array in the `SoAA<T>` type. This layout is refered as structure of arrays (SoA) in literature. Type `SoAA<book>` is shown in listing 1.3. It requires careful allocation and deallocation of the attribute arrays as the type is created, destroyed or resized. Note that the loop inside the `update_price(book_soaa&, float)` function can be SIMD vectorized.

```
struct book_soaa {
  int *id;
  float *base_price;
  float *price;
  char *category;

  book_asvt(size_t size); // allocate memory for the attribute streams
  ~book_asvt(); // release memory of the attribute streams
  void resize(size_t n); // changes the number of book objects in this container
  size_t size() const; // returns number of book objects in this container
};

void update_price(book_soaa &books, float tax_rate) {
  float tax_mul = 1.0f+tax_rate/100.0f;
  // SIMD vectorizable loop
  for(size_t i = 0; i < books.size(); ++i) {
    books.price[i] = books.base_price[i]*tax_mul;
  }
}

book_soaa books(1000); // Create SoAA of 1000 books
update_price(books, 7.25f);
```

Listing 1.3: `SoAA<book>` type

Advantages:

1. Saving in space: Each attribute array in the `SoAA<T>` type is self-aligned and does not require padding. Therefore it results in saving of space compared to the `AoS<T>` type.
2. Higher memory bandwidth: The data access pattern is contiguous, which favors hardware prefetchers.
3. Saving in memory bandwidth: Each cache line loaded due to an attribute access is completely utilized.

Disadvantages:

1. Prefetcher stream exhaustion: Typically, most of the processors support one prefetcher stream per page and only a few streams in total. A data parallel algorithm that operates on a `SoAA<T>` type needs one stream per attribute accessed in the loop. In such case, hardware data prefetcher may become ineffective if entire content of an instance of `SoAA<T>` type is contained in a single page or if the number of attributes accessed in the loop exceeds the maximum number of hardware prefetcher streams.
2. Conflict misses: Associativity of a cache is limited. If data of each attribute array starts at same offset from page boundary, cache conflicts may occur if the number of attributes accessed in an iteration exceeds associativity of the cache. This issue can be alleviated if the attribute streams start at different offsets from a page boundary.
3. Branching of code: On a machine with SIMD vector width of four float, suppose that the number of iterations of the loop are not a multiple of 4. To handle such a situation, compiler auto-vectorization typically generates a SIMD vectorized loop which operates on four consecutive float in one iteration and a scalar loop which operates on one float at a time. The vectorized version is executed as long as four consecutive values are available and scalar version is executed for the remaining iterations. Sometimes, the auto-vectorization also performs loop peeling by executing scalar version for a few of the starting iterations until data alignment so that in the subsequent iterations, the vectorized version can use aligned load and store. This is necessary since aligned data accesses is faster than unaligned data access on some architectures. A mix of the scalar and vector version, however, creates branches in the code. Cost of the branching can be amortized only for sufficiently large number of iterations.
4. This layout requires a change in syntax for accessing attributes of each `book`.

Array of Vectorized Struct (AoVS<T>) Type

For a type T, its vectorized variant can be derived by replacing each of its attribute with the attribute's vector type. Each attribute's vector type must hold the same number of instances of the attribute's type. Optimal number depends on the instruction set architecture in use. For example, listing 1.4 shows vectorized book type - book_v. For SSE 4.2 instruction set, SIMD vector register width is 4 int or 4 float or 16 char. If NW=4, then the book_vs<4> type stores data of four books. Each of the id, base_price and price attributes can be accomodated in one SSE 4.2 register. However, the category attribute falls short of 8 bytes to fill a SSE 4.2 register. This can be alleviated by either of the following means.

1. Increase NW to 8 so that each book_v<8> instance holds data of eight books.
2. Keep NW to 4 but specify the category attribute as char category[NW*2][2] - this creates space for 16 char (that fill one full SSE 4.2 register) of which first 8 char form used space and remaining 8 char form padded space
3. Keep NW to 4 but specify the category attribute as char category[NW][4] - this creates space for 16 char (that fill one full SSE 4.2 register) which is interleaving of 2 used and 2 padded char.
4. Keep NW to 4 but use mask while loading, storing or operating on the category attribute value to hide 8 most significant bytes.

The first approach saves space because it does not need padding, but it may increase chance of register spill due to larger number of registers needed to process the data in one iteration. Register spill reduces performance of an algorithm because of frequent loading and storing of data to L1 cache. The other two approaches require padding and thus cause wastage of space, but they reduce chance of registers spill and result into higher instruction throughput. The fourth approach saves space and also reduces chance of register spill, but not all instruction sets support masked operations.

```
template<size_t NW>
struct book_v {
  int id[NW]; // 1 int per book, hence NW int per book_v
  float base_price[NW]; // 1 float per book, hence NW float per book_v
  float price[NW]; // 1 float per book, hence NW float per book_v
  char category[NW][2]; // 2 char per book, hence NW*2 char per book_v
};

typedef std::vector<book_v<4> > book_aovs4;

void update_price(book_aovs4 &books, float tax_rate) {
  const size_t count = books.size();
  float tax_mul = 1.0f+tax_rate/100.0f;
  for(size_t i = 0; i < count; ++i) {
    // Following block is SIMD vectorizable for SSE 4.2 instruction set
    books[i].price[0] = books[i].base_price[0]*tax_rate;
    books[i].price[1] = books[i].base_price[1]*tax_rate;
    books[i].price[2] = books[i].base_price[2]*tax_rate;
    books[i].price[3] = books[i].base_price[3]*tax_rate;
  }
}

book_aovs4 books(250); // Create AoVS of 1000 books
update_price(books, 7.25f);
```

Listing 1.4: AoVS<T> type

Advantages:

1. SIMD vectorization: A data parallel algorithm can be SIMD vectorized.
2. Higher memory bandwidth: The strided data access pattern favors hardware prefetchers.
3. Unlike the SoAA<T> type, the AoVS<T> type requires only a single stream per AoVS<T> object which eliminates problems due to limited cache associativity and limited hardware prefetcher streams.

4. Unchanged syntax: Syntax of code that accesses attributes of the objects in AoVS layout is similar to the `AoS<T>` type except the additional array index for each attribute. This can be avoided by using SIMD instructions to operate on `NW` values all at once.

Disadvantages:

1. Wastage of memory bandwidth: An algorithm may not utilize cache contents fully if a SIMD register width is less than a cache line size.
2. Wastage of space: In a situation where an attribute's vectorized type does not fill full width of a SIMD register and padding is used, the `AoVS<T>` type causes wastage of space that increases linearly with the number of objects stored in the array.
3. Number of instances of type `T` for which a `AoVS<T>` is created is always a multiple of the number of ways (`NW`). This may result into wastage of space when a large number of `AoVS<T>` objects are used, each storing only a few values that are not multiple of `NW`.

1.2 Intra-Value and Inter-Value Data Parallel Algorithm

The data parallelism opportunities that `SoAA<T>` and `AoVS<T>` types provide is due to data independence between different instances of the value type in a data parallel algorithm. We will call such an algorithm an inter-value data parallel algorithm. However, some algorithms may be able to extract data parallelization opportunity from within a value type. Such an algorithm will be termed as intra-value data parallel algorithm in this text. For example, consider a case of vector dot product operation. In this case, the vector value type and dot product operation may be written as shown in listing 1.5. Note that the expression for calculating dot product can be vectorized because it performs four independent multiplications on consecutively stored values. The `vector4d_aos` is a `AoS<vector4d>` type. Thus a `AoS<T>` type may also be able to extract data parallelism.

```cpp
struct vector4d {
  float x, y, z, w;
};

typedef std::vector<vector4d> vector4d_aos;

float dot(std::vector<float> &dot_prod, const vector4d_aos &lhs, const vector4d_aos &rhs) {
  const size_t count = dot_prod.size();
  for(size_t i = 0; i < count; ++i) {
    // The multiplications in following expression can be vectorized using SSE 4.2
    dot_prod[i] = lhs[i].x*rhs[i].x + lhs[i].y*rhs[i].y + lhs[i].z*rhs[i].z + lhs[i].w*rhs[i].w;
  }
}

vector4d lhs(100), rhs(100]);
std::vector<float> dot_prod(100);
dot(dot_prod, lhs[i], rhs[i]);
```

Listing 1.5: Intra-value data parallel operation

1.3 SIMD Vectorization Technology

The purpose of following vetcorization technologies is to allow programmers to write architecture neutral SIMD vectorized code.

Assembly or Intrinsics/Built-ins While it is possible to write SIMD instruction by embedding assembly code in C/C++ programs, it may require a large amount of time to code and may result in suboptimal register allocation. Compiler intrinsics/built-ins can help achieve optimal register allocation with minimal programmer involvement. It also provides a compiler with all necessary information to perform code optimization through techniques such as dead code elimination, inlining, loop unrolling etc. While this method is guaranteed to generates vectorized code, it hides the original algorithm behind a verbose intrinsic code. Also code size increases because a different version of each vectorized algorithm is required for each supported architecture.

Auto-vectorization Auto-vectorization capable compilers try to detect code fragments that can be vectorized without violating code semantics. Compilers often enforce a data dependence analysis, sometimes with the help of programmer provided hints (using #pragma annotations), to prove that the instructions can be vectorized. Once such pattern is detected, the compiler generates SIMD instructions for the code. However, SIMD vectorization of code by this method depends largely on compiler capabilities and is limited to simple loops or fixed size blocks. Also, a compiler can generate both scalar and vector version for a loop when the number of iterations are not known at compile time. Which version executes at runtime depends on the number of iterations. Therefore, the generated code contains branches and is not guaranteed to have 100% vectorized execution. For aligned memory access, compilers implement techniques such as loop peeling which also generates both scalar and vector version for the loop. To assist a compiler in generating vectorized code, value type transformation such as array of struct (AoS) to struct of array (SoA), in addition to appropriate use of auto-vectorization pragmas and restrict keyword is generally used.

OpenMP/Cilk Plus Specifications such as OpenMP and Cilk Plus support vectorization using pragma, SIMD-enabled functions and array notations. This approach provides more context for the compiler for optimization of a code than C++ wrappers. Unlike compiler auto-vectorization, it relies on explicit hints provided by programmer for vectorization. It can produce vectorized code more reliably. However, a compiler supporting requred version of the specification is needed to adapt this approach to SIMD vectorization.

SIMD Wrapper Libraries Libraries such as Boost.SIMD, Generic SIMD library, Vc etc. provide yet another way of achieving SIMD vectorization. It is an abstraction layer on top of the SIMD intrinsics. Often such a library provides a set of C++ classes with overloaded arithmetic operators and mathematical functions. Implementation of the classes hides architecture specific instructions. Therefore, algorithms written using the wrapper classes allow architecturally portable algorithmic expression. However, not all libraries provide similar level of architectural neutrality. Following is a summary of key features of various SIMD libraries

Boost.SIMD: This library provides a type called `boost::simd::pack<T, N>` that abstracts a SIMD register. It also provides a type `boost::simd::logical<T, N>` that results from a comparison between two `boost::simd::pack<T, N>` types. Various operations such as C++ operators, constant generators, arithmetic functions and reduction functions etc. are provided for the `boost::simd::pack<T, N>` type. It provides STL components such as aligned allocators, iterator adapters for iterating over STL container elements, STL algorithms, sliding window iterator for stencil computations, interleaved iterators for interleaved read or write of data. The library uses expression templates to analyze AST of a large expression involving the `boost::simd::pack<T, N>` type and generates optimized code.

Generic SIMD Library: This is a header only library that is designed with the goal of providing programmers scalar like syntax, portability and fixed lane data types for mixed mode arithmetic . It provides vector type called short vector (svec) that wraps SIMD register/s and provides overloaded operators for them. Short vector are of "fixed lane" meaning that it can hold a fixed number of scalar elements irrespective of type of the scalar elements. This allows mixed mode arithmetic on short vectors. For example, one short vector of 4 64-bit doubles and another short vector of 4 32-bit integers can be used as operands of a mixed mode arithmetic expression because of the one to one correspondance of the elements of the two vectors.

Vc - SIMD Vector Classes for C++: This library provides `Vc::Vector<T>` and corresponding operators that provide abstraction for the underlying SIMD hardware registers. It also provides capability to compare two vector types that results into `Vc::Mask<T>` type. The mask can be used to perform write-masking, gather and scatter operations. It also provides a data types called `Vc::SimdArray<T, N>` and `Vc::SimdMaskArray<T, N>` that can be used to perform fixed lane vector operations similar to the Generic SIMD Library. The library also provides `Vc::simdize<T>` expressions that can be used to transform a `struct` into its vector variant. The simdized `struct` can be used for creating an array of vectorized structure (AoVS).

Even though the OpenMP, Cilk Plus and autovectorization technologies allow easy migration of exiting programs to use SIMD capability in a portable way, they require monitoring of optimization logs or generated assembly code to make sure that the SIMD vectorization is taking place as expected whenever a piece of code is changed. To ensure successful vectorization without the need for the continuous monitoring, a wrapper library will be used for this research. Due to certain limitations of existing SIMD libraries a SIMD wrapper library is being developed. It is described in next section.

1.4 SIMD Wrapper for a Short Array

Both Boost.SIMD and Vc have a rich set of functionality. However, they have following limitations.

1. Boost.SIMD does not support 512 bit instruction sets such as KNC and AVX512 as of this writing.

2. Vc's support for permutation operations is limited to one to one mapping to underlying ISA instructions and therefore it is not suitable for writing instruction set neutral code when permutations are involved.

3. Vc and Boost.SIMD do not support inplace `new` and `delete` operators, which are required when a different view of a memory area is required. This capability is required for an algorithm that deals with solving a tridiagonal linear system in later sections.

4. The SIMD wrapper libraries provide a vector type that wraps a pack of scalar values. For example, Boost.SIMD provides `boost::simd::pack<T, N>`, Vc provides `Vc::Vector<T>` and `Vc::SimdArray<T, N>`↩ , where `T` is a primitive type and `N` is the number of scalar values packed in the vector type. With the help of SIMD intrinsics and C++ operator overloading, the vector types get arithmetic operators semantics that work independently on each scalar value packed in them. Objects of these types can be used in an arithmetic expression just as if they are scalar values. These types provide one-to-one mapping of each packed scalar value to one lane in architecture dependent SIMD register. For example, in SSE 4.2 instruction set, a SIMD register can hold four `int` values. Therefore, `Vc::`↩ `Vector<int>` type maps a pack of four `int`s one-to-one to the four `int` lanes in SSE 4.2 SIMD register. However, even though the SIMD register is a pack of four `int` values in physical space, it can also be viewed as a pack of two values of type `int[2]` or a single value of type `int[2][2]` in logical space. If a wrapper library can provide a way to distinguish between vector types that provide different logical view of a SIMD register, it can be used for overload resolution or compile time detection of logical errors in a code. This capability is provided in the SIMD library that is being developed.

Though adding these capabilities to the libraries is possible by modifying their source code, it requires a good amount of understanding of their source code. Therefore I have rolled out my own SIMD wrapper library. The library is not as feature rich as Boost.SIMD or Vc and is not meant to replace any existing vectorization technology, but it is tailored specifically to achieve vectorization of algorithms dealt in this work. The library is described in following subsections.

1.4.1 File Structure of the Library

The library is header-only to allow full compiler optimization and uses C++ templates. It provides a `simd::pack<T, NW>` type that represents a SIMD vector type. One instance of `simd::pack<T, NW>` packs `NW` instances of type `T`. Note that the type `T` is restricted to a primitive type or an array of a primitive type for which extents of each dimension are known at compile time and `NW` is restricted to values that perfectly map the `NW` values of type `T` to one or more SIMD registers. As of now, the library defines simd::pack<T, NW> for int, long, float and double primitive types and their multi-timensional array types, and for AVX, AVX2, AVX512-F and KNC instruction sets. It supports compilation using Intel C++ 15.0 and GNU GCC 4.9.3 compiler.

Implementation of the library is spread into one interface definition header file and platform specific implementation header files. It is possible to perform this segregation between interface and implementation due to lazy instantiation of C++ templates. The file simd.h is the only file that programmers need to include in their code. The file simddef.h contains interface definition of the library. This interface is implemented in files such as simd256x86def.h for AVX and AVX2 instruction sets and simd512x86def.h for AVX512-F and KNC instruction sets. Using compiler predefined macros, the library detects target instruction set and includes appropriate implementation file. For programs that are designed to run on heterogeneous systems with different SIMD register width, such as a program running on Ivy Bridge processor offloading part its execution to Xeon Phi device, the vector type chooses its default scalar width equal to largest of the SIMD register width on all of the systems. This allows use of same data structure for all systems and avoids the need to transforms data layout to and from the data layout on the target system. The library defines a preprocessor macro called SIMD_MAX_WIDTH that is initialized to the byte width of the widest SIMD register.

1.4.2 Defaults

The simd::defaults<T> class provides a constant called simd::defaults<T>::nway that is initialized to a value equal to the number of values of type T that can be packed perfectly in SIMD_MAX_WIDTH bytes. The constant is used when simd::pack<T, NW> type is instantiated without specifying NW. Such a SIMD vector type maps perfectly to one or more target specific SIMD registers without the need to specify NW explicitely. If the constant cannot be defined for a given type T, a static assertion is raised. Definition of simd::defaults<T> is shown in listing 1.6. Note that T must conform to the restrictions mentioned earlier.

```
namespace simd {
  template<typename T>
  struct defaults {
    static_assert(
      simd::defaults<typename std::remove_all_extents<T>::type>::nway*
      sizeof(typename std::remove_all_extents<T>::type)/sizeof(T)*sizeof(T)
      ==
      simd::defaults<typename std::remove_all_extents<T>::type>::nway*
      sizeof(typename std::remove_all_extents<T>::type), "Invalid T"
    );

    static const int nway = simd::defaults<typename std::remove_all_extents<T>::type↩
        >::nway*sizeof(typename std::remove_all_extents<T>::type)/sizeof(T);
  };
}
```

Listing 1.6: simd::defaults<T> implementation

1.4.3 Type Traits

The struct simd::type_traits<T, NW> provides meta information about the simd::pack<T, NW> type. An instance of simd::pack<T, NW> wraps NW values of type T. The meta information includes number of SIMD registers per pack, number of values of type T per pack and number of values of type BT per pack. Here, BT is a base type obtained by removing all dimensions of type T. Other meta information includes number of values of type T per SIMD register, number of values of type BT per SIMD register and number of values of type BT per T. It also provides information about instruction specific SIMD register and mask types and base type of T. The struct is defined recursively when T is an array type as shown in listing 1.7. Base case definitions of the struct are implemented by instruction set specific specialization of the type_traits<T, NW> structure.

```
namespace simd {
  template<typename T, int NW = simd::defaults<T>::nway>
  struct type_traits {
  };

  template<typename T, size_t N, int NW>
  struct type_traits<T[N], NW> {
   typedef typename simd::type_traits<T, N*NW>::base_type base_type;
   typedef typename simd::type_traits<T, N*NW>::register_type register_type;
   typedef typename simd::type_traits<T, N*NW>::mask_register_type mask_register_type;

   static const int num_regs = simd::type_traits<T, N*NW>::num_regs;
   static const int num_vals = simd::type_traits<T, N*NW>::num_vals/N;
   static const int num_bvals = simd::type_traits<T, N*NW>::num_bvals;
   static const int num_vals_per_reg=simd::type_traits<T, N*NW>::num_vals_per_reg/N;
   static const int num_bvals_per_reg=simd::type_traits<T, N*NW>::num_bvals_per_reg;
   static const int num_bvals_per_val=simd::type_traits<T, N*NW>::num_bvals_per_val*N;

   static_assert(num_vals*N == simd::type_traits<T, N*NW>::num_vals, "Invalid num_vals↩
      ");
   static_assert(num_vals_per_reg*N == simd::type_traits<T, N*NW>::num_vals_per_reg, "↩
      Invalid num_vals_per_reg");
   static_assert(num_bvals_per_val/N == simd::type_traits<T, N*NW>::num_bvals_per_val,↩
       "Invalid num_bvals_per_val");
  };

  // AVX and AVX2 instruction set specific specialization of simd_traits<T, NW> class ↩
      for scalar float type.
  // Defined in simd256x86def.h
  template<int NW>
  struct type_traits<float, NW> {
    static_assert(NW%simd::defaults<float>::nway == 0, "Invalid NW");
    static_assert(NW/simd::defaults<float>::nway > 0, "Invalid NW");

    typedef float base_type;
    typedef __m256 register_type;
    typedef __m256i mask_register_type;

    static const int num_regs = NW/simd::defaults<float>::nway;
    static const int num_vals = NW;
    static const int num_bvals = NW;
    static const int num_vals_per_reg = NW/num_regs;
    static const int num_bvals_per_reg = NW/num_regs;
    static const int num_bvals_per_val = 1;
  };
}
```

Listing 1.7: `simd::type_traits<T,NW>` implementation

1.4.4 SIMD Vector Type

The library provides `simd::pack<T, NW>` as its SIMD vector type. Its template definition is as shown in listing 1.8. It maps a pack of `NW` instances of type `T` to one or more instruction set specific SIMD registers. If `NW` instances of type `T` do not map perfectly to one or more SIMD registers a static assertion is raised. The template parameter `NW` defaults to platform specific value provided by `simd::defaults<T>::nway`. There are three more template parameters which default to values provided by `simd::type_traits<T, NW>`. They are, `RT` - platform specific SIMD register type, `BT` - scalar type of type `T`, and `W` - number of values of type `BT` that are contained in one instance of type `T`. These three parameters are not meant to be specified by programmer. They exist solely for the purpose of function or operator overload resolution. The `simd::pack<T, NW>` provides following features.

Typedefs for value, register and base type, mask type and unaligned and aligned pointer types.

Constructors of following types.

1. default constructor

2. copy constructor
3. constructor that takes `simd::pack<U, NW1>` with `U` != `T` and `NW1` != `NW`
4. constructor that takes `register_type` - all registers in `simd::pack<T, NW>` are initialized to this value
5. constructor that allows initialization from initializer list - if the initializer list is not long enough to initialize all values in `simd::pack<T, NW>` remaining values in it are undefined
6. constructor that loads values from a memory pointer - this will be discussed in more details in later section

Assignment operators that follow same pattern as the constructors.

User defined conversion operators for conversion to `register_type`, `value_type` pointer and `simd::pack<U, NW1>` types.

Indexing operators that allow register or value type access by its index. Following operators are provided. They don't perform any bounds check. These operators can be defined where lvalue is expected.

1. operator `()` that takes an index and returns SIMD register at that index.
2. operator `[ ]` that takes an index and returns value of type `T` at that index.

Overloaded `new` and `delete` operators that perform aligned allocation and deallocation. The operators are implemented for scalar and array allocation/deallocation with semantics of throw, nothrow, inplace or a combination of these. These overloaded operators use `alignof()` to determine alignment of `simd::pack<T, NW>`. Therefore the type is marked with `alignas(SIMD_MAX_WIDTH)`.

Compound assignment operators are implemented using non-member overloaded operators defined outside the `simd::pack<T, NW>` type.

Overloaded `operator<<(std::ostream&,...)` for outputting the contents of `simd::pack<T, NW>` to output stream.

```
namespace alignas(SIMD_MAX_WIDTH) simd {
  template<
    typename T,
    int NW = simd::defaults<T>::nway,
    typename RT = typename simd::type_traits<T, NW>::register_type,
    typename BT = typename simd::type_traits<T, NW>::base_type,
    int W = simd::type_traits<T, NW>::num_bvals_per_val
  >
  class pack {
    friend friend std::ostream& operator<< <T, NW, RT, BT, W>(std::ostream &strm, ↩
        const pack<T, NW, RT, BT, W> &op);

  public:
    typedef pack<T, NW, RT, BT, W> self_type;
    typedef T value_type;
    typedef RT register_type;
    typedef BT base_type;
    typedef mask<T, NW> mask_type;
    typedef pack<T, NW, RT, BT, W>* pointer;
    typedef WIN_PTR_ALIGN pack<T, NW, RT, BT, W>* LINUX_PTR_ALIGN aligned_pointer;

  public:
    // Constructors
    pack();
    pack(const self_type &rhs);
    pack(const register_type &r);
    pack(std::initializer_list<base_type> init);
    template<int hint> pack(const_mptr<base_type, hint> ptr);
    template<typename fT, int fNW, typename fRT, typename fBT, int fW> pack(const pack↩
        <fT, fNW, fRT, fBT, fW> &rhs);

    // Assignment operators
    self_type & operator=(const self_type &rhs);
    self_type & operator=(const register_type &r);
    self_type& operator=(std::initializer_list<base_type> init);
    template<int hint> self_type& operator=(const_mptr<base_type, hint> ptr);
    template<typename fT, int fNW, typename fRT, typename fBT, int fW> operator=(const↩
         pack<fT, fNW, fRT, fBT, fW> &rhs);

    // User defined conversion operators
    operator register_type & ();
    operator const register_type & () const;
    operator value_type * ();
    operator const value_type * () const;
    template<typename fT, int fNW, typename fRT, typename fBT, int fW> operator pack<↩
        fT, fNW, fRT, fBT, fW> ();

    // Indexing operators
    register_type & operator()(int index = 0);
    const register_type & operator()(int index = 0) const;
    value_type & operator[](int index);
    const value_type & operator[](int index) const;

    // Overloaded new and delete operators
    static void* operator new(size_t count) throw(std::bad_alloc);
    static void operator delete(void *ptr) throw();
    static void* operator new[](size_t count) throw(std::bad_alloc);
    static void operator delete[](void *ptr) throw();
    static void* operator new(size_t count, void *where) throw(std::bad_alloc);
    static void operator delete(void *ptr, void *where) throw();
    static void* operator new[](size_t count, void *where) throw(std::bad_alloc);
    static void operator delete[](void *ptr, void *where) throw();
    static void* operator new(size_t count, const std::nothrow_t &nt) throw();
    static void operator delete(void *ptr, const std::nothrow_t &nt) throw();
    static void* operator new[](size_t count, const std::nothrow_t &nt) throw();
    static void operator delete[](void *ptr, const std::nothrow_t &nt) throw();
    static void* operator new(size_t count, const std::nothrow_t &nt, void *where) ↩
        throw();
    static void operator delete(void *ptr, const std::nothrow_t &nt, void *where) ↩
        throw();
    static void* operator new[](size_t count, const std::nothrow_t &nt, void *where) ↩
        throw();
    static void operator delete[](void *ptr, const std::nothrow_t &nt, void *where) ↩
        throw();

    // Compound assignment operators
    self_type& operator+=(const self_type &rhs);
    self_type& operator-=(const self_type &rhs);
    self_type& operator*=(const self_type &rhs);
    self_type& operator/=(const self_type &rhs);
    self_type& operator%=(const self_type &rhs);
    self_type& operator&=(const self_type &rhs);
    self_type& operator|=(const self_type &rhs);
    self_type& operator^=(const self_type &rhs);
    self_type& operator<<=(const self_type &rhs);
    self_type& operator>>=(const self_type &rhs);

  private:
    union {
      register_type reg[simd::type_traits<T, NW>::num_regs];
      value_type elm[NW];
      base_type belm[simd::type_traits<T, NW>::num_bvals];
    };
  };
}
```

Listing 1.8: `simd::pack<T, NW>` declaration

Overloaded Operators and Mathematical Functions

Default implementation of overloaded arithmetic operators and mathematical functions for `simd::pack<T, NW>` type is defined in header file `simddef.h`. They always throw static assertion. For specific instruction set, these operators and functions are overloaded and defined in implementation specific header files. For example, the binary arithmetic operator + is overloaded as shown in listing 1.9. The library currently defines following operators and functions for AVX, AVX2, KNC and AVX512-F instruction sets and for `int`, `long`, `float`, `double` and their array types.

1. binary arithmetic operators: +, −, ∗, /, %
2. comparison operators: ==, !=, <, >, <=, >=
3. trigonometric functions: `sin`, `cos`, `tan`

Result of a comparison operator is of type `simd::mask<T, NW>` that is described in next subsection. Support for other functions and operators will be added in future release.

```cpp
namespace simd {
  // Base definition of arithmetic operator +
  // Defined in simddef.h
  template<typename T, int NW, typename RT, typename BT, int W>
  pack<T, NW, RT, BT, W> operator+(const pack<T, NW, RT, BT, W> &lhs, const pack<T, NW, RT, BT, W> &rhs) {
    static_assert(false, "Not implemented");
  }

  // Overloaded definition of arithmetic operator + for AVX/AVX2 instruction set
  // Defined in simd256x86def.h
  template<typename T, int NW, int W>
  inline pack<T, NW, __m256, float, W> operator+(const pack<T, NW, __m256, float, W> &lhs, const pack<T, NW, __m256, float, W> &rhs) {
    pack<T, NW, __m256, float, W> temp;

    for(int i = 0; i < pack<T, NW, __m256, float, W>::num_regs; ++i) {
      temp(i) = _mm256_add_ps(lhs(i), rhs(i));
    }

    return temp;
  }

  // Overloaded definition of arithmetic operator + for KNC/AVX512-F instruction set
  // Defined in simd512x86def.h
  template<typename T, int NW, int W>
  inline pack<T, NW, __m512, float, W> operator+(const pack<T, NW, __m512, float, W> &lhs, const pack<T, NW, __m512, float, W> &rhs) {
    pack<T, NW, __m512, float, W> temp;

    for(int i = 0; i < pack<T, NW, __m512, float, W>::num_regs; ++i) {
      temp(i) = _mm512_add_ps(lhs(i), rhs(i));
    }

    return temp;
  }
}
```

Listing 1.9: Overloaded binary operator + for `simd::pack<T, NW>` when `T` is a `float` or `float` array

1.4.5 SIMD Mask Type

The library provides `simd::mask<T, NW>` type that is used to create a mask or is generated as a result of performing comparison between two operands of type `simd::pack<T, NW>` (see listing 1.10). Different instruction sets use different mask types. For example, a mask is represented by a SIMD register (e.g. `__m256` etc. which is 256 bit long) on AVX and AVX2 instruction set, while KNC and AVX512-F instruction sets use a special mask register (e.g. `__mmask16` which is 16 bit long). The `simd::mask<T, NW>` type uses meta information provided by `simd::type_traits<T, NW>` to get implementation specific mask register type.

The `simd::mask<T, NW>` also provides following functions and operators.

User defined conversion functions

1. Conversion to bool that returns true is all bits in the mask are set, false otherwise.
2. Conversion to `mask_register_type` that returns reference to first mask register.

Indexing operators

1. operator () that takes an index and returns reference to mask register at that index.
2. operator [ ] that takes an index and returns reference to `mask_bit` at that index.

Bit checking functions

1. bool `test_all_true()`const that checks if all bit of the mask are set.
2. bool `test_all_false()`const that checks if all bit of the mask are unset.

Implementation specific functionality of these functions is provided with the help of function overloading.

```
namespace simd {
  template<
    typename T,
    int NW
  >
  class mask {
  public:
    typedef mask<T, NW> self_type;
    typedef typename simd::type_traits<T, NW>::base_type base_type;
    typedef typename simd::type_traits<T, NW>::mask_register_type register_type;

  private:
    class mask_bit {
      friend class mask<T, NW>;
      // Implementation
    };

  public:
    // User defined conversion functions
    operator bool ();
    operator bool () const;
    operator register_type & ();
    operator const register_type & () const;

    // Indexing operators
    register_type & operator()(int index = 0);
    const register_type & operator()(int index = 0) const;
    mask_bit operator[](int index);
    mask_bit operator[](int index) const;

    // Bit checking functions
    bool test_all_true() const;
    bool test_all_false() const;
  };
}
```

Listing 1.10: `simd::mask<T, NW>` declaration

1.4.6 Memory Load and Store Operations

All instruction sets support various types of semantics for transfering data between a SIMD register and a memory address. These semantics are as follows.

1. temporal or non-temporal load and stores
2. aligned or unaligned load and stores
3. strided load and stores
4. masked load and stores

Out of these, the library supports combinations of temporal/non-temporal and aligned/unaligned semantics for the load and store. The library provides `load(pack<T, NW, RT, BT, W>&, const_mptr<BT, hint>)` and `store(const pack<T, NW, RT, BT, W>&, mptr<BT, hint>)` functions. The load and store functions take parameter of type `const_mptr<T, int>` or `mptr<T, int>` to allow programmers specify the semantics explicitely. Apart from this, when storage is allocated to a `simd::pack<T, NW>` type on stack or on heap, compiler can automatically determine appropriate method to load or store.

```
namespace simd {
  enum memhint {
    temporal = 0x0,
    unaligned = 0x1,
    aligned = 0x2,
    nontemporal = 0x4
  };

  template<typename T, int hint = memhint::temporal|memhint::unaligned>
  class const_mptr {
  public:
    const_mptr(const T *p) : ptr(p) {}
    const_mptr(const const_mptr<T, hint> &src) : ptr(src.ptr) {}
    operator const T*() const { return ptr; }

  private:
    const T *ptr;
  };

  template<typename T, int hint = memhint::temporal|memhint::unaligned>
  class mptr {
  public:
    mptr(T *p) : ptr(p) {}
    mptr(const mptr<T, hint> &src) : ptr(src.ptr) {}
    operator T*() { return ptr; }
    operator const T*() const { return ptr; }
    operator const_mptr<T, hint>() { return const_mptr<T, hint>(ptr); }

  private:
    T *ptr;
  };

  // For loading a pack from a memory address. Overloads of this function are
  // implemented for each supported instruction set and semantics.
  template<typename T, int NW, typename RT, typename BT, int W, int hint>
  void load(pack<T, NW, RT, BT, W> &op, const_mptr<BT, hint> ptr);

  // For storing a pack to a memory address. Overloads of this function are
  // implemented for each supported instruction set and semantics.
  template<typename T, int NW, typename RT, typename BT, int W, int hint>
  void store(const pack<T, NW, RT, BT, W> &op, mptr<BT, hint> ptr);
}
```

Listing 1.11: Classes for memory load and store operations

1.4.7 Permutation Operations

Most of the SIMD instruction sets provide facility to permute lanes of a SIMD register. To make this functionality available to users of the library, `permutation` class is provided (see listing 1.12). The `permute_pattern` enumeration provides constants for available permutation patters. As of now the library supports permutation for a type `simd::pack<T, NW, RT, BT, W>` that has `W = 4`, that is, the type `T` is such that its base type width is 4.

```
namespace simd {
  enum permute_pattern {
    aacc,
    abab,
    bbdd,
    cdcd,
    dcba,
    dbca,
    // ... many such patterns can be added to the library
  };

  template<permute_pattern pattern>
  class permutation {
  };

  // Specialization of permutation for AVX512-F and KNC instruction sets for ↩
      permutation pattern "aacc" and for int, long, float, double. Similar ↩
      specializations are implemented for other instruction sets and other permutation ↩
      patterns
  template<>
  class permutation<permute_pattern::aacc> {
  public:
    template<typename T, int NW>
    static pack<T, NW, __m512i, int, 4> permute(const pack<T, NW, __m512i, int, 4> &op↩
        );

    template<typename T, int NW>
    static pack<T, NW, __m512i, long, 4> permute(const pack<T, NW, __m512i, long, 4> &↩
        op);

    template<typename T, int NW>
    static pack<T, NW, __m512, float, 4> permute(const pack<T, NW, __m512, float, 4> &↩
        op);

    template<typename T, int NW>
    static pack<T, NW, __m512d, double, 4> permute(const pack<T, NW, __m512d, double, ↩
        4> &op);
  };
}
```

Listing 1.12: Permutation functionality

1.4.8 Vectorizing a Value Type using `simd::pack<T, NW>`

Now, let's look back at the `book` example. The `struct book` can be converted to its vectorized version as shown in listing 1.13. Note that the `update_price(book_aovs&, float)` function has same syntax as the version operating on the `AoS<book>` type, but still the loop gets vectorized. Here, each attribute `T` of a primitive type or its array type is replaced by `simd::pack<T, NW>`. If a compound type contains an attribute of another compound type, then the attribute should be replaced by its type's vectorized version, which is again obtained similarly by recursion. Though this process can be automated by using a meta-compiler or C++ template trics, current implementation of the library does not support this automation. It relies on programmers to correctly perform this vectorization. Also note that the default template parameter `NW` is set to target instruction set specific value. Therefore, if it is not specified explicitely, the same code can be used for 128 bit, 256 bit or 512 bit SIMD instruction sets.

```
template<int NW = simd::defaults<int>::nway>
struct book_v {
  simd::pack<int, NW> id;
  simd::pack<float, NW> base_price;
  simd::pack<float, NW> price;
  simd::pack<char[4], NW> category; // char[4] is used instead of char[2] to make this↩
      attribute of same width as int or float
};

typedef std::vector<book_v> book_aovs;

void update_price(book_aovs &books, float tax_rate) {
  const size_t count = books.size();
  simd::pack<float> tax_mul = 1.0f + tax_rate/100.0f;
  for(size_t i = 0; i < count; ++i) {
    // This line is vectorized by the wrapper library
    books[i].price = books[i].base_price*tax_rate;
  }
}

book_aovs books(1000/simd::defaults<T>::nway); // Create AoVS of 1000 books
update_price(books, 7.25f);
```

Listing 1.13: Vectorization of type `book` and of the algorithms that operate on it

1.5 Value Types for Blocked Linear Algebra

Discretization of a multi-component PDE gives a matrix system $Ax = b$ where matrix A is a sparse matrix with blocks of same fixed dimensions as its elements. The block elements of the matrix A are usually stored in an array using a sparse storage scheme such as compressed sparse row (CSR). Therefore, a matrix block is the fundamental value type for a library that provides blocked sparse linear algebra routines. In this section we will consider various ways of designing the value type.

1.5.1 A Plain Matrix Type

Suppose that we need a value type for storing a matrix block of dimensions $R \times C$. It can be defined using a C++ structure as shown in listing 1.14. However, this value type does not enforce the notion of ordering of elements in the matrix. Therefore, a function that operates on instances of `mat<R, C, T>` needs to assume certain ordering of elements. The `multiply_r(...)` function in the listing performs matrix-matrix multiplication assuming row-major ordering of elements of the `res`, `lhs`, and `rhs` matrices. If the matrix-matrix multiplication function for column-major ordered matrices is required, a different function name is required, such as `multiply_c(...)`. A client code, that uses the multiplication functionality, needs different code base when it uses row-ordered matrices vs column-ordered matrices.

```
template<int R, int C, typename T>
struct mat {
  static constexpr int rows = R;
  static constexpr int cols = C;
  T elm[R*C]; // order of elements not enforced
};

template<int R, int Q, int C, typename T>
void multiply_r(mat<R, C, T> &res, const mat<R, Q, T> &lhs, const mat<Q, C, T> &rhs) {
  for(int r = 0; r < mat<R, C, T>::rows; ++r) {
    for(int c = 0; c < mat<R, C, T>::cols; ++c) {
      res.elm[r*mat<R, C, T>::cols+c] = 0.0f;
      for(int q = 0; q < mat<R, Q, T>::cols; ++q) {
        res.elm[r*mat<R, C, T>::cols+c] += lhs.elm[r*mat<R, Q, T>::cols+q] * rhs.elm[q↩
            *mat<Q, C, T>::cols+c];
      }
    }
  }
}

template<int R, int Q, int C, typename T>
void multiply_c(mat<R, C, T> &res, const mat<R, Q, T> &lhs, const mat<Q, C, T> &rhs) {
  for(int r = 0; r < mat<R, C, T>::rows; ++r) {
    for(int c = 0; c < mat<R, C, T>::cols; ++c) {
      res.elm[c*mat<R, C, T>::rows+r] = 0.0f;
      for(int q = 0; q < mat<R, Q, T>::cols; ++q) {
        res.elm[c*mat<R, C, T>::rows+r] += lhs.elm[q*mat<R, Q, T>::rows+r] * rhs.elm[c↩
            *mat<Q, C, T>::rows+q];
      }
    }
  }
}

// Row ordered matrices
typedef std::vector<mat<4, 4, float>> mat4x4r_aos;
mat4x4r_aos lhsr(1000), rhsr(1000), resr(1000);
// Initialize lhsr and rhsr elements

// Column ordered matrices
typedef std::vector<mat<4, 4, float>> mat4x4c_aos;
mat4x4c_aos lhsc(1000), rhsc(1000), resc(1000);
// Initialize lhsc and rhsc elements

// Client code for row-ordered matrices
for(size_t i = 0; i < 1000; ++i) {
  multiply_r(resr[i], lhsr[i], rhsr[i]);
}

// Client code for column-ordered matrices
for(size_t i = 0; i < 1000; ++i) {
  multiply_c(resc[i], lhsc[i], rhsc[i]);
}
```

Listing 1.14: Value type for a matrix block of dimensions $R \times C$

1.5.2 An Ordered Matrix Type

The situation shown in last subsection can be alleviated by adding the ordering semantics into the value type. An approach to achieve this is to define separate types for row-ordered and column-ordered matrices as shown in listing 1.15. Function overloading can be used to define the matrix-matrix multiplication function with the same name for both the row-ordered and column-ordered matrices. This makes the client code invariant to the ordering semantics enforced by the matrix data layout. However, this approach requires creating a separate implementation of `multiply()` function for each matrix layout type. Note that the definition of both the functions is identical, yet separate implementations are required for dealing with the two types of matrices. Further, to enable mixing of row ordered and column ordered matrices, separate implementation is required for each combination of the matrix types.

```
template<int R, int C, typename T>
class matr {
public:
  static constexpr int rows = R;
  static constexpr int cols = C;

  T& operator(int r, int c); // Access element at row index r and column index c
  const T& operator(int r, int c) const; // Access element at row index r and column ↩
      index c

private:
  T elm[R][C]; // row-ordered elements
};

template<int R, int C, typename T>
class matc {
public:
  static constexpr int rows = R;
  static constexpr int cols = C;

  T& operator(int r, int c); // Access element at row index r and column index c
  const T& operator(int r, int c) const; // Access element at row index r and column ↩
      index c

private:
  T elm[C][R]; // column-ordered elements
};

template<int R, int Q, int C, typename T, typename U, typename V>
void multiply(matr<R, C, T> &res, const matr<R, Q, U> &lhs, const matr<R, Q, V> &rhs) ↩
    {
  for(int r = 0; r < matr<R, C, T>::rows; ++r) {
    for(int c = 0; c < matr<R, C, T>::cols; ++c) {
      res(r, c) = 0;
      for(int q = 0; q < matr<R, Q, T>::cols; ++q) {
        res(r, c) += lhs(r, q) * rhs(q, c);
      }
    }
  }
}

template<int R, int Q, int C, typename T, typename U, typename V>
void multiply(matc<R, C, T> &res, const matc<R, Q, U> &lhs, const matc<R, Q, V> &rhs) ↩
    {
  for(int r = 0; r < matc<R, C, T>::rows; ++r) {
    for(int c = 0; c < matc<R, C, T>::cols; ++c) {
      res(r, c) = 0;
      for(int q = 0; q < matc<R, Q, U>::cols; ++q) {
        res(r, c) += lhs(r, q) * rhs(q, c);
      }
    }
  }
}

// Row ordered matrices
typedef std::vector<matr<4, 4, float>> mat4x4r_aos;
mat4x4r_aos resr(1000), lhsr(1000), rhsr(1000);
// Initialize lhs and rhs elements

// Column ordered matrices
typedef std::vector<matc<4, 4, float>> mat4x4c_aos;
mat4x4c_aos resc(1000), lhsc(1000), rhsc(1000);
// Initialize lhs and rhs elements

// Client code for row-ordered matrix
for(size_t i = 0; i < 1000; ++i) {
  multiply(resr[i], lhsr[i], rhsr[i]);
}

// Client code for column-ordered matrix
for(size_t i = 0; i < 1000; ++i) {
  multiply(resc[i], lhsc[i], rhsc[i]);
}
```

Listing 1.15: Value types for a row-ordered and a column-ordered matrix block of dimensions $R \times C$

1.5.3 A Vectorized and Ordered Matrix Type

SIMD vectorization of the matrix types adds yet another layer of semantics in the design of a matrix type and requires overloaded definition of related functions for the vectorized matrix types as well. SIMD vectorization enabled matrix types are `matr_v` and `matc_v` as shown in the listing 1.16. If a SIMD register width is ω, one instance of `matr_v` or `matc_v` stores data for ω matrix blocks in row ordered and column ordered layout respectively. The listing also shows code for the overloaded `multiply(...)` functions for the vector types. Accessing (i, j)th element of the instance gives access to (i, j)th element of all the ω matrix blocks stored in the instance. Therefore, one invocation of the `multiply(matr_v<R, C, T>&, const matr_v<R, Q, T>&, const matr_v<Q, C, T>&)` or the `multiply(matc_v<R, C, T>&, const matc_v<R, Q, T>&, const matc_v<Q, C, T>&)` function performs ω matrix multiplications simultaneously. Note that the arithmetic operators are overloaded for `simd::pack<T, NW>` type, which allows using the arithmetic operators for the elements of the matrices. Creating functions for operating on mixed matrix layouts requires overloading the function for all the combinations of the layouts even if their code structure is same.

```
template<int R, int C, typename T>
class matr_v {
public:
  static constexpr int rows = R;
  static constexpr int cols = C;

  simd::pack<T>& operator(int r, int c); // Access element pack at row index r and column index c
  const simd::pack<T>& operator(int r, int c) const; // Access element pack at row index r and column index c

private:
  simd::pack<T> elm[R][C]; // row-ordered elements
};

template<int R, int C, typename T>
class matc_v {
public:
  static constexpr int rows = R;
  static constexpr int cols = C;

  simd::pack<T>& operator(int r, int c); // Access element pack at row index r and column index c
  const simd::pack<T>& operator(int r, int c) const; // Access element pack at row index r and column index c

private:
  simd::pack<T> elm[C][R]; // column-ordered elements
};

template<int R, int Q, int C, typename T>
void multiply(matr_v<R, C, T> &res, const matr_v<R, Q, T> &lhs, const matr_v<Q, C, T> &rhs) {
  for(int r = 0; r < matr_v<R, C, T>::rows; ++r) {
    for(int c = 0; c < matr_v<R, C, T>::cols; ++c) {
      res(r, c) = 0.0f;
      for(int q = 0; qt < matr_v<R, Q, T>::cols; ++qt) {
        res(r, c) += lhs(r, q) * rhs(q, c);
      }
    }
  }
}

template<int R, int Q, int C, typename T>
void multiply(matc_v<R, C, T> &res, const matc_v<R, Q, T> &lhs, const matc_v<Q, C, T> &rhs) {
  for(int r = 0; r < matc_v<R, C, T>::rows; ++r) {
    for(int c = 0; c < matc_v<R, C, T>::cols; ++c) {
      res(r, c) = 0.0f;
      for(int q = 0; qt < matc_v<R, Q, T>::cols; ++qt) {
        res(r, c) += lhs(r, q) * rhs(q, c);
      }
    }
  }
}
```

Listing 1.16: Vectorized `mat4x4` that stores ω matrices of dimension 4×4

1.5.4 A Tiled, Vectorized and Ordered Matrix Type

One drawback of the SIMD vectorized matrix types is that, it needs ω independent sets of matrices for each invocation of the function. If there are less than ω independent sets, padding is required for each operand to fill remaining lanes of a SIMD register. The padding results into wastage of space and computations. The amount of wastage increases as ω increases. In such a situation, a value type that can reduce data parallelism is required. Yet, algorithms using the type should be be able to use entire width of a SIMD register. Such a value type can be designed for a $R \times C$ matrix block as shown in listing 1.17. This type assumes that each matrix is composed of tiles of dimensions 2×2 arranged in row-major order. Therefore the `elm` attribute is of type `simd::pack<float[2][2]>`. This means that one SIMD register of width ω contains $\omega/4$ values of type `float [2][2]` stored consecutively. Therefore, `elm[i][j]` contains (i, j)th tile of first matrix followed by (i, j)th tile of second matrix and so on up to (i, j)th tile of $\omega/4$th matrix. In other words, `elm[i][j]` refers to (i, j)th tile of all of the $\omega/4$ matrices. If we further assume that within a tile elements are stored in row major order (in-tile order), then the `multiply(...)` function can be overloaded for the tiled and vectorized matrix type as shown in the listing 1.17. Note the use of permutations to achieve multiplication of two tiles. It is the only difference from earlier versions of the `multiply(...)` function. This function will work on any architecture that has SIMD register width of at least 4 or a multiple of 4 `float`s. Otherwise, a compiler error will be generated (due to design of the `simd::pack<T, NW>` type).

```
template<int R, int C, typename T>
class matr_t2x2r_v {
public:
  static constexpr int rows = R;
  static constexpr int cols = C;
  static constexpr int blocked_rows = R/2+(R%2?1:0);
  static constexpr int blocked_cols = C/2+(C%2?1:0)
  typedef simd::pack<T[2][2]> value_type;

  value_type& operator(int r, int c); // Access element pack at tile index r and ↩
      column index c
  const value_type& operator(int r, int c) const; // Access element pack at tile index↩
       r and column index c

private:
  simd::pack<T[2][2]> elm[blocked_rows][blocked_cols];
};

template<int R, int Q, int C, typename T>
void multiply(matr_t2x2r_v<R, C, T> &res, const matr_t2x2r_v<R, Q, T> &lhs, const ↩
    matr_t2x2r_v<Q, R, T> &rhs) {
  typedef typename matr_t2x2r_v<Q, R, T>::value_type lvt;
  typedef typename matr_t2x2r_v<R, Q, T>::value_type rvt;

  for(int rt = 0; rt < matr_t2x2r_v<R, C, T>::blocked_rows; ++rt) {
    for(int ct = 0; ct < matr_t2x2r_v<R, C, T>::blocked_cols; ++ct) {
      res(rt, ct) = 0.0f;
      for(int qt = 0; qt < matr_t2x2r_v<R, Q, T>::cols; ++qt) {
        lvt lhs_aacc = simd::permutation<simd::permute_pattern::aacc>::permute(lhs(rt,↩
             qt));
        lvt lhs_bbdd = simd::permutation<simd::permute_pattern::bbdd>::permute(lhs(rt,↩
             qt));
        rvt rhs_abab = simd::permutation<simd::permute_pattern::abab>::permute(rhs(qt,↩
             ct));
        rvt rhs_cdcd = simd::permutation<simd::permute_pattern::cdcd>::permute(rhs(qt,↩
             ct));
        res(rt, ct) += lhs_aacc*rhs_abab + lhs_bbdd*rhs_cdcd;
      }
    }
  }
}
```
Listing 1.17: Vectorized `mat4x4` that stores $\omega/4$ matrices of dimension 4×4

1.5.5 A Unified Matrix Type

The ordering, SIMD vectorization and tiling semantics result into a number of ways of designing a matrix value type. This also requires writing a plethora of overloaded functions for a matrix operation even if their code structure is same. To avoid this inconvenience, we need a single value type that can create data layout for any combination of the semantics. If close attention is paid to the matrix types mentioned above, it can be observed that each layout semantics affects only a particular aspect of the `elm` attribute. For example, ordering semantics just changes the order of the 2D array extents, vectorization semantics changes just the type of `elm` from float to `simd::pack<float>` etc. Therefore, to inject the orthogonal aspects of the layout semantics into a matrix type, we need policy classes that can influence the way the matrix type lays out its data, while exposing same interface to its client code.

Policy classes for element ordering

For injecting ordering semantics into the matrix type, `column_ordered<T>` and `row_ordered<T>` policy classes are designed as shown in listing 1.18 and 1.19. These classes contain `block_generate<R, C>` as a templated type generator. The generator provides a `block_generator<R, C>::block_type` type that can be used for storing elements of a matrix block in the desired order. It also provides static methods that the matrix type can use to extract (i, j)th element from the data type.

```
template<typename T>
struct column_ordered {
  using matrix_type = typename mat_traits<T>::matrix_type;

  static constexpr int rows = mat_traits<T>::rows;
  static constexpr int cols = mat_traits<T>::cols;

  template<int R, int C>
  struct block_generator {
    static constexpr int blocked_rows = R/rows + (R%rows?1:0);
    static constexpr int blocked_cols = C/cols + (C%cols?1:0);
    using block_value_type = matrix_type;
    using block_type = matrix_type[blocked_cols][blocked_rows];

    inline static block_value_type& get_block_value(block_type &block, int index) {
      return *(&block[0][0]+index);
    }

    inline static const block_value_type& get_block_value(const block_type &block, int↩
        index) {
      return *(&block[0][0]+index);
    }

    inline static block_value_type& get_block_value(block_type &block, int br, int bc)↩
        {
      return block[bc][br];
    }

    inline static const block_value_type& get_block_value(const block_type &block, int↩
        br, int bc) {
      return block[bc][br];
    }
  };
};
```
Listing 1.18: Policy class for injecting column ordering semantics into matrix type

```
template<typename T>
```

```
struct row_ordered {
  using matrix_type = typename mat_traits<T>::matrix_type;

  static constexpr int rows = mat_traits<T>::rows;
  static constexpr int cols = mat_traits<T>::cols;

  template<int R, int C>
  struct block_generator {
    static constexpr int blocked_rows = R/rows + (R%rows?1:0);
    static constexpr int blocked_cols = C/cols + (C%cols?1:0);
    using block_value_type = matrix_type;
    using block_type = matrix_type[blocked_rows][blocked_cols];

    inline static block_value_type& get_block_value(block_type &block, int index) {
      return *(&block[0][0]+index);
    }

    inline static const block_value_type& get_block_value(const block_type &block, int↩
        index) {
      return *(&block[0][0]+index);
    }

    inline static block_value_type& get_block_value(block_type &block, int br, int bc)↩
        {
      return block[br][bc];
    }

    inline static const block_value_type& get_block_value(const block_type &block, int↩
        br, int bc) {
      return block[br][bc];
    }
  };
};
```

Listing 1.19: Policy class for injecting row ordering semantics into matrix type

Policy class for SIMD vectorization

The `vectorized<T>` policy class injects vectorization semantics into the matrix type. This class has a type generator that provides `block_generator<R, C>::block_type` type that is a SIMD vector of underlying matrix type of `T`.

```
template<typename T>
struct vectorized {
  using matrix_type = simd::pack<typename mat_traits<T>::matrix_type>;

  static constexpr int rows = mat_traits<T>::rows;
  static constexpr int cols = mat_traits<T>::cols;

  template<int R, int C>
  struct block_generator {
    static constexpr int blocked_rows = 1;
    static constexpr int blocked_cols = 1;
    using block_value_type = simd::pack<typename mat_traits<T>::block_generator<R, C↩
        >::block_type>;
    using block_type = block_value_type;

    inline static block_value_type& get_block_value(block_type &block, int index) {
      return block;
    }

    inline static const block_value_type& get_block_value(const block_type &block, int↩
        index) {
      return block;
    }

    inline static block_value_type& get_block_value(block_type &block, int br, int bc)↩
        {
      return block;
    }

    inline static const block_value_type& get_block_value(const block_type &block, int↩
        br, int bc) {
      return block;
    }
  };
};
```

Listing 1.20: Policy class for enorcing SIMD vectorization

The Matrix Class

The class `mat<R, C, T>` is the type that can be used to store elements of a matrix of dimension $R \times C$. The type `T` must be one of the policy class as shown earlier. Various policies can be chained together to achieve desired type of data layout. For injecting tiling semantics into the matrix type, no separate policy classes are designed because tiling can be achieved by using nested matrices.

```
template<int R, int C, typename T>
class mat {
private:
  using block_generator = typename mat_traits<T>::block_generator<R, C>;

public:
  using matrix_type = mat<R, C, T>;
  static constexpr int rows = R;
  static constexpr int cols = C;

  static constexpr int blocked_rows = block_generator::blocked_rows;
  static constexpr int blocked_cols = block_generator::blocked_cols;
  using block_value_type = typename block_generator::block_value_type;
  using block_type = typename block_generator::block_type;

public:
  inline block_value_type& operator[](int index) {
    return block_generator::get_block_value(m_block, index);
  }

  inline const block_value_type& operator[](int index) const {
    return block_generator::get_block_value(m_block, index);
  }

  inline block_value_type& operator()(int br, int bc) {
    return block_generator::get_block_value(m_block, br, bc);
  }

  inline const block_value_type& operator()(int br, int bc) const {
    return block_generator::get_block_value(m_block, br, bc);
  }

private:
  block_type m_block;
};
```

Listing 1.21: Class for matrix type

Traits Classes

Traits class and its specialization are used to bind the policy classes with the matrix class.

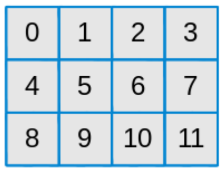
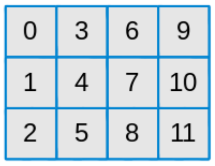
```
template<typename T>
```

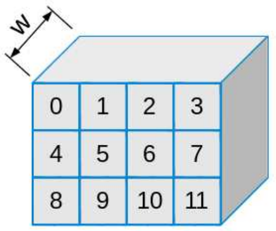
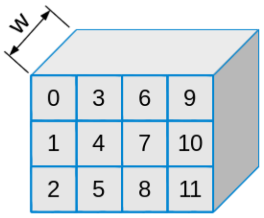
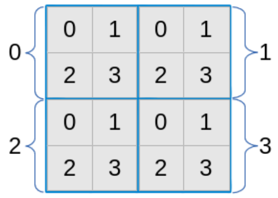
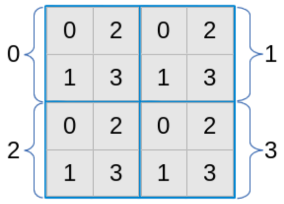
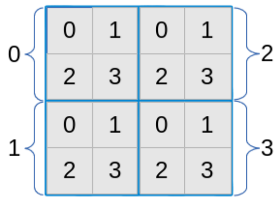
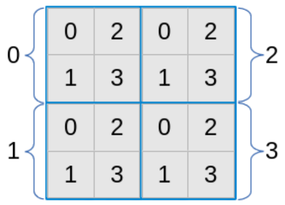
```
struct mat_traits {
  using matrix_type = typename T::matrix_type;

  static constexpr int rows = T::rows;
  static constexpr int cols = T::cols;

  template<int R, int C>
  using block_generator = typename T::block_generator<R, C>;
};

template<int R, int C, typename T>
struct mat_traits<mat<R, C, T> > {
  using matrix_type = typename mat<R, C, T>::matrix_type;

  static constexpr int rows = mat<R, C, T>::rows;
  static constexpr int cols = mat<R, C, T>::cols;
};

template<>
struct mat_traits<float> {
  using matrix_type = float;

  static constexpr int rows = 1;
  static constexpr int cols = 1;
};

template<>
struct mat_traits<double> {
  using matrix_type = double;

  static constexpr int rows = 1;
  static constexpr int cols = 1;
};
```

Listing 1.22: Traits classes for binding policy classes with the class `mat<R, C, T>`

1.5.6 Creating Instances of The Unified Matrix Type

Even though the framework described in this section for creating various data layouts has complicated internal structure, it is much easier to use in practice. Some examples of concrete matrix types along with their graphical representation are shown in table 1.1. Set of elements that have blue border around them represents one element of the outermost matrix.

No.	Type and Description	Graphical Representation
1	`mat<3, 4, row_ordered<float>>`: 3 × 4 matrix composed of float elemments arranged in row-major order	
2	`mat<3, 4, column_ordered<float>>`: 3 × 4 matrix composed of float elements arranged in column-major order	

3	`mat<3, 4, row_ordered<vectorized<float>>>`: 3×4 matrix composed of vectorized float elements arranged in row-major order	
4	`mat<3, 4, column_ordered<vectorized<float>>>`: 3×4 matrix composed of vectorized float elements arranged in column-major order	
5	`mat<4, 4, row_ordered<mat<2, 2, row_ordered<float>>>>`: 4×4 matrix composed of `mat<2, 2, row_ordered<float>>` elements arranged in row-major order. Each element is again a 2×2 matrix composed of float elements arranged in row-major order	
6	`mat<4, 4, row_ordered<mat<2, 2, column_ordered<float>>>>`: 4×4 matrix composed of `mat<2, 2, row_ordered<float>>` elements arranged in row-major order. Each element is again a 2×2 matrix composed of float elements arranged in column-major order	
7	`mat<4, 4, column_ordered<mat<2, 2, row_ordered<float>>>>`: 4×4 matrix composed of `mat<2, 2, row_ordered<float>>` elements arranged in column-major order. Each element is again a 2×2 matrix composed of float elements arranged in row-major order	
8	`mat<4, 4, column_ordered<mat<2, 2, column_ordered<float>>>>`: 4×4 matrix composed of `mat<2, 2, row_ordered<float>>` elements arranged in column-major order. Each element is again a 2×2 matrix composed of float elements arranged in column-major order	

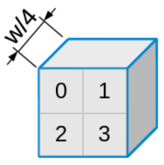
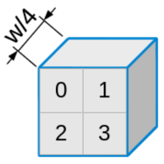
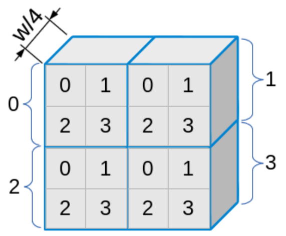
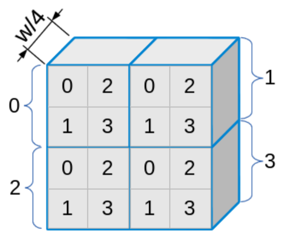
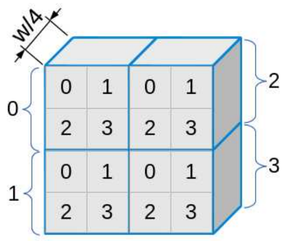
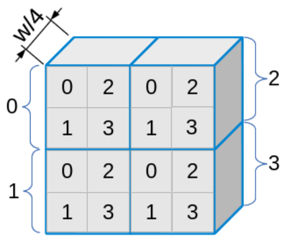
9	mat<2, 2, vectorized<row_ordered<float>>>: 2 × 2 matrix composed of vectorized float [2][2] row-major ordered elements	
10	mat<2, 2, vectorized<column_ordered<float>>>: 2×2 matrix composed of vectorized float [2][2] column-major ordered elements	
11	mat<4, 4, row_ordered<mat<2, 2, vectorized<row_ordered<float↩ >>>>>: 4 × 4 matrix composed of mat<2, 2, vectorized<↩ row_ordered<float>>> elements arranged in row-major order. Each element is again a 2 × 2 matrix composed of vectorized float [2][2] row-major ordered elements.	
12	mat<4, 4, row_ordered<mat<2, 2, vectorized<column_ordered<float↩ >>>>>: 4 × 4 matrix composed of mat<2, 2, vectorized<↩ column_ordered<float>>> elements arranged in row-major order. Each element is again a 2 × 2 matrix composed of vectorized float [2][2] column-major ordered elements.	
13	mat<4, 4, column_ordered<mat<2, 2, vectorized<row_ordered<↩ float>>>>>: 4 × 4 matrix composed of mat<2, 2, vectorized↩ <row_ordered<float>>> elements arranged in column-major order. Each element is again a 2 × 2 matrix composed of vectorized float [2][2] row-major ordered elements.	
14	mat<4, 4, column_ordered<mat<2, 2, vectorized<column_ordered<↩ float>>>>>: 4 × 4 matrix composed of mat<2, 2, vectorized↩ <column_ordered<float>>> elements arranged in column-major order. Each element is again a 2 × 2 matrix composed of vectorized float [2][2] column-major ordered elements.	

Table 1.1: Examples of using mat<R, C, T> and policy classes for obtaining various data layouts

1.5.7 Creating Algorithms for The Unified Matrix Type

Let's take an example of multiply-add operation on matrices. For matrices A, B and C of dimensions $R \times C$, $R \times Q$, $Q \times C$ respectively, this operation can be defined as $A = A + B * C$. The operation can be implemented for the unified matrix type as shown in listing 1.23. The `multiply_add(...)` function captures basic code structure common to the multiply-add operation of all types of matrices. To make this function work, we need base case implementations as shown in listing 1.24. With these functions in place, the `multiply_add(...)` function can be used with any of the matrix layouts shown in table 1.1. This example demonstrates that the unified matrix type can significantly reduce the amount of code required to operate on a variety of data layouts of a matrix block.

```
// Driver function for fused multiply add of matrices (a += b*c)
template<int R, int Q, int C, typename T>
inline void multiply_add(mat<R, C, T> &res, mat<R, Q, T> &lhs, mat<Q, C, T> &rhs) {
  for(int rti = 0; rti < mat<R, C, T>::blocked_rows; ++rti) {
    for(int cti = 0; cti < mat<R, C, T>::blocked_cols; ++cti) {
      for(int qti = 0; qti < mat<R, Q, T>::blocked_cols; ++qti) {
        multiply_add(res(rti, cti), lhs(rti, qti), rhs(qti, cti));
      }
    }
  }
}
```

Listing 1.23: Multiply-add functions for the unified matrix type

```
template<typename T, typename U>
using is_primitive_type = std::enable_if<
  std::is_same<float, T>::value ||simd::pack<T>
  std::is_same<double, T>::value ||
  std::is_same<int, T>::value ||
  std::is_same<long, T>::value,
  U
>;

// Base case when matrix is of type float | double | int | long
template<typename T>
inline void multiply_add(typename is_primitive_type<T, T>::type &res, const T &lhs, ↩
    const T &rhs) {
  res += lhs * rhs;
}

// Base case when matrix is of type simd::pack<float | double | int | long>
template<typename T>
inline void multiply_add(typename is_primitive_type<T, simd::pack<T>>::type &res, ↩
    const simd::pack<T> &lhs, const simd::pack<T> &rhs) {
  res += lhs * rhs;
}

// Base case when matrix is of type mat<2, 2, vectorized<row_ordered<float | double | ↩
    int | long>>>
template<typename T>
inline void multiply_add(
    typename is_primitive_type<T, mat<2, 2, vectorized<row_ordered<T>>>>::type &res,
    const mat<2, 2, vectorized<row_ordered<T>>> &lhs,
    const mat<2, 2, vectorized<row_ordered<T>>> &rhs) {
  typedef typename mat<2, 2, vectorized<row_ordered<T>>>::block_value_type type_t;

  type_t lhs_aacc=simd::permutation<simd::permute_pattern::aacc>::permute(lhs(0, 0));
  type_t lhs_bbdd=simd::permutation<simd::permute_pattern::bbdd>::permute(lhs(0, 0));
  type_t rhs_abab=simd::permutation<simd::permute_pattern::abab>::permute(rhs(0, 0));
  type_t rhs_cdcd=simd::permutation<simd::permute_pattern::cdcd>::permute(rhs(0, 0));

  res(0, 0) += lhs_aacc*rhs_abab + lhs_bbdd*rhs_cdcd;
}

```

```
// Base case when matrix is of type mat<2, 2, vectorized<column_ordered<float | double↩
    | int | long>>>
template<typename T>
inline void multiply_add(
   typename is_primitive_type<T, mat<2, 2, vectorized<column_ordered<T>>>>::type &res,
   const mat<2, 2, vectorized<column_ordered<T>>> &lhs,
   const mat<2, 2, vectorized<column_ordered<T>>> &rhs) {
  typedef typename mat<2, 2, vectorized<row_ordered<T>>>::block_value_type type_t;

  type_t rhs_aacc=simd::permutation<simd::permute_pattern::aacc>::permute(rhs(0, 0));
  type_t rhs_bbdd=simd::permutation<simd::permute_pattern::bbdd>::permute(rhs(0, 0));
  type_t lhs_abab=simd::permutation<simd::permute_pattern::abab>::permute(lhs(0, 0));
  type_t lhs_cdcd=simd::permutation<simd::permute_pattern::cdcd>::permute(lhs(0, 0));

  res(0, 0) += rhs_aacc*lhs_abab + rhs_bbdd*lhs_cdcd;
}
```

Listing 1.24: Multiply-add functions for base case matrices