ispc: A SPMD Compiler for High-Performance CPU Programming

Matt Pharr
Intel Corporation
matt.pharr@intel.com

William R. Mark
Intel Corporation
william.r.mark@intel.com

ABSTRACT

SIMD parallelism has become an increasingly important mechanism for delivering performance in modern CPUs, due its power efficiency and relatively low cost in die area compared to other forms of parallelism. Unfortunately, languages and compilers for CPUs have not kept up with the hardware's capabilities. Existing CPU parallel programming models focus primarily on multi-core parallelism, neglecting the substantial computational capabilities that are available in CPU SIMD vector units. GPU-oriented languages like OpenCL support SIMD but lack capabilities needed to achieve maximum efficiency on CPUs and suffer from GPU-driven constraints that impair ease of use on CPUs.

We have developed a compiler, the Intel® SPMD Program Compiler (`ispc`), that delivers very high performance on CPUs thanks to effective use of both multiple processor cores and SIMD vector units. `ispc` draws from GPU programming languages, which have shown that for many applications the easiest way to program SIMD units is to use a single-program, multiple-data (SPMD) model, with each instance of the program mapped to one SIMD lane. We discuss language features that make `ispc` easy to adopt and use productively with existing software systems and show that `ispc` delivers up to 35x speedups on a 4-core system and up to 240x speedups on a 40-core system for complex workloads (compared to serial C++ code).

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Compilers*; D.1.3 [**Concurrent programming**]: Parallel programming

Keywords

SPMD, parallel programming, SIMD, CPUs

1. INTRODUCTION

Recent work has shown that CPUs are capable of delivering high performance on a variety of highly parallel workloads by using both SIMD and multi-core parallelism [22]. Coupled with their ability to also efficiently execute code with moderate to small amounts of parallelism, this makes CPUs an attractive target for a range of computationally-intensive applications, particularly those that exhibit varying amounts of parallelism over the course of their execution.

However, achieving this performance is difficult in practice; although techniques for parallelizing across CPU cores are well-known and reasonably easily adopted, parallelizing across SIMD vector lanes remains difficult, often requiring laboriously writing intrinsics code to generate desired instruction sequences by hand. The most common parallel programming languages and libraries designed for CPUs—including OpenMP, MPI, Thread Building Blocks, UPC, and Cilk—focus on multi-core parallelism and do not provide any assistance for targeting SIMD parallelism within a core. There has been some hope that CPU implementations of GPU-oriented languages that support SIMD hardware (such as OpenCL) might address this gap [10], but OpenCL lacks capabilities needed to achieve maximum efficiency on CPUs and imposes productivity penalties caused by needing to accommodate GPU limitations such as a separate memory system. This situation led us to ask what would be possible if one were to design a language specifically for achieving high performance and productivity for using SIMD vector units on modern CPUs.

We have implemented a language and compiler, the Intel® SPMD Program Compiler (`ispc`), that extends a C-based language with "single program, multiple data" (SPMD) constructs for high-performance SIMD programming.[1] `ispc`'s "SPMD-on-SIMD" execution model provides the key feature of being able to execute programs that have divergent control flow across the SIMD lanes. `ispc`'s main focus is effective use of CPU SIMD units, though it supports multi-core parallelism as well.

The language and underlying programming model are designed to fully expose the capabilities of modern CPU hardware, while providing ease of use and high programmer productivity. Programs written in `ispc` generally see their performance scale with the product of both the number of processing cores and their SIMD width; this is a standard characteristic of GPU programming models but one that is much less common on the CPU.

The most important features of `ispc` for performance are:

- Explicit language support for both scalar and SIMD operations.
- Support for structure-of-arrays data structures, including for converting previously-declared data types into structure of arrays layout.
- Access to the full flexibility of the underlying CPU hardware, including the ability to launch asynchronous tasks and to perform fast cross-lane SIMD operations.

[1] `ispc` is available for download in both source and binary form from `http://ispc.github.com`.

The most important features of `ispc` for usability are:

- Support for tight coupling between C++ and `ispc`, including the ability to directly call `ispc` routines from C++ and to also call C++ routines from `ispc`.[2]
- Coherent shared memory between C++ and `ispc`.
- Familiar syntax and language features due to its basis in C.

Most `ispc` language features are inspired by earlier SIMD languages such as C* [33], Pixar's FLAP C [24], the RenderMan Shading Language [11] and the MasPar programming language [29], and in some cases by more modern GPU-oriented languages such as CUDA [31] and OpenCL [19]. The primary contribution of this paper is to design and implement a language and compiler targeted at modern CPU architectures, and to evaluate the performance impact of key language features on these architectures.

2. DESIGN GOALS

In order to motivate some of the design differences between `ispc` and other parallel languages, we will discuss the specific goals of the system and the key characteristics of the hardware that it targets.

2.1 Goals

Performance on today's CPU hardware: The target users for `ispc` are performance-focused programmers. Therefore, a key design goal is that the system should provide performance transparency: just as with C, it should be straightforward for the user to understand how code written in the language will be compiled to the hardware and roughly how the code will perform. The target hardware is modern CPU hardware, with SIMD units from four to sixteen elements wide, and in particular x86 CPUs with SSE or AVX instructions. A skilled user should be able to achieve performance similar (85% or better) to that achievable by programming in C with SSE and AVX intrinsic functions. Modern x86 workstations have up to forty cores and the Intel MIC architecture will have over fifty cores on a single chip, so `ispc` programs should be able to scale to these core counts and beyond.

Programmer productivity: Programmer productivity should be substantially higher than that achievable by programming with intrinsic functions, and should be comparable to that of writing high-performance serial C code or OpenCL kernels. Productivity is measured not just in terms of writing code, but also by the ease of reading and modifying code. Of course, it is expected that it will take longer and require more skill to write highly-tuned code than it does to write less efficient code, just as when programming in other languages. It should be possible (though not mandatory) to write code that is portable across architectures with differing SIMD widths.

Ease of adoption and interoperability: It should be easy for programmers to adopt the language, both for new code and for incremental enhancements to existing systems. The language should be as familiar as possible so that it is easy to learn. Ideally the language should be so similar to C that porting code or sharing data structure definitions between it and C/C++ is easy. To support incremental use of the system, it should be easy to call back and forth between C/C++ and `ispc` code, and it should be easy to share complex pointer-based data structures. Code generated by the compiler should interoperate with existing memory allocators and task schedulers, rather than imposing its own. Finally, the system should easily work with existing build systems, debuggers and tools like memory verifiers (e.g. *valgrind*). (Lua's "embeddability" goals are similar [12].)

[2]Here and throughout this paper, we use "C++ code" or "application code" to indicate the rest of the software system that `ispc` is being used with. This could include, for example, Fortran or Python code that called `ispc` code.

2.2 Non-goals

It is useful to specifically list several non-goals of `ispc`.

No support for GPUs: CPU and GPU architectures are sufficiently different that a single performance-focused programming model for both is unlikely to be an ideal fit for either. Thus, we focus exclusively on CPU architectures. For example, we assume a single cache-coherent address space, which would not be possible if the language had to support today's discrete GPUs.

Don't try to provide "safe" parallelism: we do not attempt to protect the programmer by making races or deadlock difficult or impossible. Doing so would place too many levels of abstraction between the programmer and the underlying hardware, and we choose to focus on programmers who are willing give up some safety in return for achieving peak machine performance, just as they might do when they choose C or C++ over Java.

2.3 Target Hardware

Since one of the primary goals of the `ispc` language is to provide high efficiency on modern CPU hardware, it is helpful to review some of the characteristics of this hardware that impact the language and compiler design.

Multi-core and SIMD parallelism: A modern CPU consists of several cores, each of which has a scalar unit and a SIMD unit. The instructions for accessing the SIMD unit have different names on different architectures: SSE for 128-bit wide SIMD on x86 processors, AVX for 256-bit wide SIMD on Intel processors, AltiVec on PowerPC processors, and Neon on ARM processors. The `ispc` compiler currently supports SSE and AVX.

Simultaneous execution of scalar and SIMD instructions: Modern CPU architectures can issue multiple instructions per cycle when appropriate execution units are available for those instructions. There is often a performance advantage from replacing a SIMD instruction with a scalar instruction due to better occupancy of execution units. The architects of the Pixar FLAP observed long ago that even SIMD-heavy code has a large number of addressing and control computations that can be executed on a scalar unit [24].

One program counter per core: The scalar unit and all lanes of the associated SIMD unit share a single hardware program counter.

Single coherent memory: All cores share a single cache-coherent address space and memory system for both scalar and SIMD operations. This capability greatly simplifies the sharing of data structures between serial and parallel code. This capability is lacking on today's GPUs.

Cross-lane SIMD operations: SSE and AVX efficiently support various cross-lane SIMD operations such as swizzles via a single instruction. GPUs generally provide

weaker support for these operations, although they can be mimicked at lower performance via memory.

Tightly defined execution order and memory model: Modern CPUs have relatively strict rules on the order with which instructions are completed and the rules for when memory stores become visible to memory loads. GPUs have more relaxed rules, which provides greater freedom for hardware scheduling but makes it more difficult to provide ordering guarantees at the language level.

3. PARALLELISM MODEL: SPMD ON SIMD

Any language for parallel programming requires a conceptual model for expressing parallelism in the language and for mapping this language-level parallelism to the underlying hardware. For the following discussion of `ispc`'s approach, we rely on Flynn's taxonomy of programming models into SIMD, MIMD, etc. [8], with Darema's enhancement to include SPMD (Single Program Multiple Data) [7].

3.1 Why SPMD?

Recall that our goal is to design a language and compiler for today's SIMD CPU hardware. One option would be to use a purely sequential language, such as unmodified C, and rely on the compiler to find parallelism and map it to the SIMD hardware. This approach is commonly referred to as *auto-vectorization* [37]. Although auto-vectorization can work well for regular code that lacks conditional operations, a number of issues limit the applicability of the technique in practice. All optimizations performed by an auto-vectorizer must honor the original sequential semantics of the program; the auto-vectorizer thus must have visibility into the entire loop body, which precludes vectorizing loops that call out to externally-defined functions, for example. Complex control flow and deeply nested function calls also often inhibit auto-vectorization in practice, in part due to heuristics that auto-vectorizers must apply to decide when to try to vectorize. As a result, auto-vectorization fails to provide good performance transparency—it is difficult to know whether a particular fragment of code will be successfully vectorized by a given compiler and how it will perform.

To achieve `ispc`'s goals of efficiency and performance transparency it is clear that the language must have parallel semantics. This leads to the question: how should parallelism be expressed? The most obvious option is to explicitly express SIMD operations as explicit vector computations. This approach works acceptably in many cases when the SIMD width is four or less, since explicit operations on 3-vectors and 4-vectors are common in many algorithms. For SIMD widths greater than four, this option is still effective for algorithms without data-dependent control flow, and can be implemented in C++ using operator overloading layered over intrinsics. However, this option becomes less viable once complex control flow is required.

Given complex control flow, what the programmer ideally wants is a programming model that is as close as possible to MIMD, but that can be efficiently compiled to the available SIMD hardware. SPMD provides just such a model: with SPMD, there are multiple instances of a single program executing concurrently and operating on different data. SPMD programs largely look like scalar programs (unlike explicit SIMD), which leads to a productivity advantage for programmers working with SPMD programs. Furthermore, the SPMD approach aids with performance transparency: vectorization of a SPMD program is guaranteed by the underlying model, so a programmer can write SPMD code with a clear mental model of how it will be compiled. Over the past ten years the SPMD model has become widely used on GPUs, first for programmable shading [28] and then for more general-purpose computation via CUDA and OpenCL.

`ispc` implements SPMD execution on the SIMD vector units of CPUs; we refer to this model as "SPMD-on-SIMD". Each instance of the program corresponds to a different SIMD lane; conditionals and control flow that are different between the program instances are allowed. As long as each program instance operates only on its own data, it produces the same results that would be obtained if it was running on a dedicated MIMD processor. Figure 1 illustrates how SPMD execution is implemented on CPU SIMD hardware.

3.2 Basic Execution Model

Upon entry to a `ispc` function called from C/C++ code, the execution model switches from the application's serial model to `ispc`'s SPMD model. Conceptually, a number of *program instances* start running concurrently. The group of running program instances is a called a *gang* (harkening to "gang scheduling", since `ispc` provides certain guarantees about when program instances running in a gang run concurrently with other program instances in the gang, detailed below.)[3] The gang of program instances starts executing in the same hardware thread and context as the application code that called the `ispc` function; no thread creation or implicit context switching is done by `ispc`.

The number of program instances in a gang is relatively small; in practice, it's no more than twice the SIMD width of the hardware that it is executing on.[4] Thus, there are four or eight program instances in a gang on a CPU using the 4-wide SSE instruction set, and eight or sixteen on a CPU using 8-wide AVX. The gang size is set at compile time.

SPMD parallelization across the SIMD lanes of a single core is complementary to multi-core parallelism. For example, if an application has already been parallelized across cores, then threads in the application can independently call functions written in `ispc` to use the SIMD unit on the core where they are running. Alternatively, `ispc` has capabilities for launching asynchronous tasks for multi-core parallelism; they will be introduced in Section 5.4.

3.3 Mapping SPMD To Hardware: Control

One of the challenges in SPMD execution is handling divergent control flow. Consider a `while` loop with a termination test `n > 0`; when different program instances have different values for `n`, they will need to execute the loop body different numbers of times.

`ispc`'s SPMD-on-SIMD model provides the illusion of separate control flow for each SIMD lane, but the burden of

[3]Program instances thus correspond to threads in CUDA and work items in OpenCL. A gang roughly corresponds to a CUDA warp.

[4]Running gangs wider than the SIMD width can give performance benefits from amortizing shared computation (such as scalar control flow overhead) over more program instances, better cache reuse across the program instances, and from more instruction-level parallelism being available. The costs are greater register pressure and potentially more control flow divergence across the program instances.

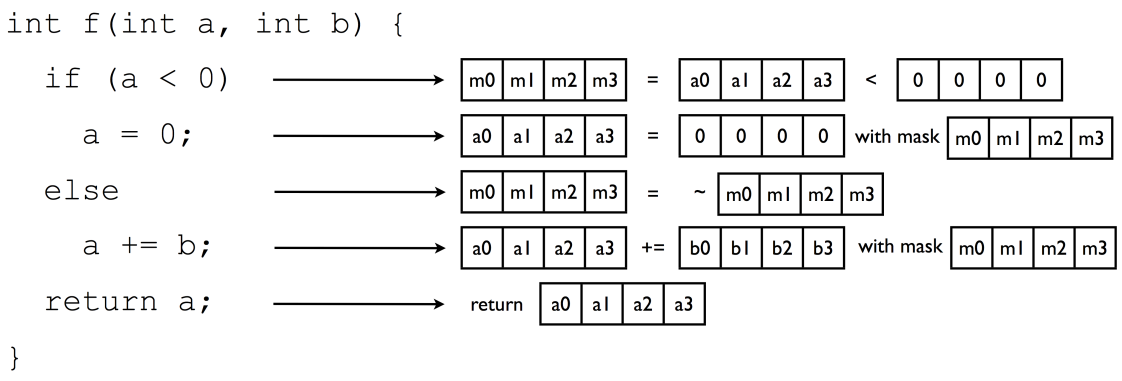

Figure 1: Execution of a 4-wide SPMD program on 4-wide SIMD vector hardware. On the left we have a short program with simple control flow; the right illustrates how this program is compiled to run on SIMD vector hardware. Here, the `if` statement has been converted into partially predicated instructions, so the instructions for both the "true" and "false" cases are always executed. A mask is used to prevent side effects for program instances that should not themselves be executing instructions in a particular control flow path.

supporting this illusion falls on the compiler. Control flow constructs are compiled following the approach described by Allen et al. [2] and recently generalized by Karrenberg et al. [17], where control flow is transformed to data flow.

A simple example of this transformation is shown in Figure 1, where assignments to a variable are controlled by an `if` statement. The SIMD code generated by this example maintains a mask that indicates which program instances are currently active during program execution. Operations with side-effects are masked so that they don't have any effect for program instances with an "off" mask value. This approach is also applied to loops (including `break` and `continue` statements) and to multiple `return` statements within one function.

Implementation of this transformation is complex on SSE hardware due to limited support for SIMD write-masks, in contrast to AVX, MIC and most GPUs. Instead, the compiler must use separate blend/select instructions designed for this purpose. Fortunately, masking isn't required for all operations; it is unnecessary for most temporaries computed when evaluating an expression, for example. Because the SPMD programming model is used pervasively on GPUs, most GPUs have some hardware/ISA support for SPMD control flow, thereby reducing the burden on the compiler [25, 27].

3.4 SPMD and Synchronization

`ispc` provides stricter guarantees of execution convergence than GPUs running SPMD programs do; these guarantees in turn provide ease-of-use benefits to the programmer. `ispc` specifically provides an important guarantee about the behavior of the program counter and execution mask: the execution of program instances within a gang is *maximally converged*. Maximal convergence means that if two program instances follow the same control path, they are guaranteed to execute each program statement concurrently. If two program instances follow diverging control paths, it is guaranteed that they will re-converge at the earliest point in the program where they could re-converge.[5]

In contrast, CUDA and OpenCL have much looser guarantees on execution order, requiring explicit barrier synchronization among program instances with `__syncthreads()` or `barrier()`, respectively, when there is communication between program instances via memory. Implementing these barriers efficiently for OpenCL on CPUs is challenging [10].

Maximally converged execution provides several advantages compared to the looser model on GPUs; it is particularly helpful for efficient communication of values between program instances without needing to explicitly synchronize among them. However, this property also can introduce a dependency on SIMD width; by definition, ordering changes if the gang size changes. The programmer generally only needs to consider this issue when doing cross-program-instance communication.

The concept of lockstep execution must be precisely defined at the language level in order to write well-formed programs where program instances depend on values that are written to memory by other program instances within their gang. With `ispc`, any side effect from one program instance is visible to other program instances in the gang after the next sequence point in the program, where sequence points are defined as in C. Generally, sequence points include the end of a full expression, before a function is entered in a function call, at function return, and at the end of initializer expressions. The fact that there is no sequence point between the increment of `i` and the assignment to `i` in `i=i++` is why that expression yields undefined behavior in C, for example. Similarly, if multiple program instances write to the same location without an intervening sequence point, undefined behavior results. (The `ispc` User's Guide has further details about these convergence guarantees and resulting implications for language semantics [14].)

3.5 Mapping SPMD To Hardware: Memory

The loads and stores generated by SPMD execution can present a performance challenge. Consider a simple array indexing operation like `a[index]`: when executed in SPMD, each of the program instances will in general have a different value for `index` and thus access a different memory location. Loads and stores of this type, corresponding to loads and stores with SIMD vectors of pointers, are typically called "gathers" and "scatters" respectively. It is frequently the case at runtime that these accesses are to the same location

[5]This guarantee is *not* provided across gangs in different threads; in that case, explicit synchronization must be used.

or to sequential locations in memory; we refer to this as a coherent gather or scatter. For coherent gather/scatter, modern DRAM typically delivers better performance from a single memory transaction than a series of discrete memory transactions.

Modern GPUs have memory controllers that coalesce coherent gathers and scatters into more efficient vector loads and stores [25]. The range of cases that this hardware handles has generally expanded over successive hardware generations. Current CPU hardware lacks "gather" and "scatter" instructions; SSE and AVX only provide vector load and store instructions for contiguous data. Therefore, when gathers and scatters are required, they must be implemented via a less-efficient series of scalar instructions.[6] `ispc`'s techniques for minimizing unnecessary gathers and scatters are described in Section 6.4.

4. LANGUAGE OVERVIEW

To give a flavor of the syntax and how the language is used, here is a simple example of using `ispc`. For more extensive examples and language documentation, see the `ispc` online documentation [13].

First, we have some setup code in C++ that dynamically allocates and initializes two arrays. It then calls an `update()` function.

```
float *values = new float[1024];
int *iterations = new int[1024];
// ... initialize values[], iterations[] ...
update(values, iterations, 1024);
```

The call to `update()` is a regular function call; in this case it happens that `update()` is implemented in `ispc`. The function squares each element in the `values` array the number of times indicated by the corresponding entry in the `iterations` array.

```
export void update(uniform float values[],
                   uniform int iterations[],
                   uniform int num_values) {
    for (int i = programIndex; i < num_values;
         i += programCount) {
        int iters = iterations[i];
        while (iters-- > 0)
            values[i] *= values[i];
    }
}
```

The syntax and basic capabilities of `ispc` are based on C (C89 [3], specifically), though it adopts a number of constructs from C99 and C++. (Examples include the ability to declare variables anywhere in a function, a built-in `bool` type, references, and function overloading.) Matching C's syntax as closely as possible is an important aid to the adoptability of the language.

The `update()` function has an `export` qualifier, which indicates that it should be made callable from C++; the `uniform` variable qualifier specifies scalar storage and computation and will be described in Section 5.1.

`ispc` supports arbitrary structured control flow within functions, including `if` statements, `switch` statements, `for`, `while`, and `do` loops, as well as `break`, `continue`, and `return` statements in all of the places where they are allowed in C.[7] Different program instances can follow different control paths; in the example above, the `while` loop may execute a different number of times for different elements of the array.

`ispc` provides a standard library of useful functions, including hardware atomic operations, transcendentals, functions for communication between program instances, and various data-parallel primitives such as reductions and scans across the program instances.

4.1 Mapping Computation to Data

Given a number of instances of the program running in SPMD (i.e. one gang), it's necessary for the instances to iterate over the input data (which is typically larger than a gang). The example above does this using a `for` loop and the built-in variables `programIndex` and `programCount`. `programCount` gives the total number of instances running (i.e. the gang size) and `programIndex` gives each program instance an index from zero to `programCount-1`. Thus, in the above, for each `for` loop iteration a `programCount`-sized number of contiguous elements of the input arrays are processed concurrently by the program instances.

`ispc`'s built-in `programIndex` variable is analogous to the `threadIdx` variable in CUDA and to the `get_global_id()` function in OpenCL, though a key difference is that in `ispc`, looping over more than a gang's worth of items to process is implemented by the programmer as an in-language `for` or `foreach` loop, while in those languages, the corresponding iteration is effectively done by the hardware and runtime thread scheduler outside of the user's kernel code. Performing this mapping in user code gives the programmer more control over the structure of the parallel computation.

4.2 Implementation

The `ispc` compiler uses *flex* and *bison* for tokenization and parsing. The compiler front-end performs type-checking and standard early optimizations such as constant folding before transforming the program to the vector intermediate representation of the LLVM toolkit [21]. LLVM then performs an additional set of traditional optimizations. Next our custom optimizations are applied, as discussed in Section 6. LLVM then generates final assembly code.

It is reasonably easy to add support for new target instruction sets: most of the compiler is implemented in a fashion that is target agnostic (e.g. "gathers" are issued generically and only late in the compilation process are they transformed to a target-specific operation).

5. DESIGN FEATURES

We'll now describe some key features of `ispc` and how they support the goals introduced in Section 2.

5.1 "Uniform" Datatypes

In a SPMD language like `ispc`, a declaration of a variable like `float x` represents a variable with a separate storage location (and thus, potentially different value) for each of the program instances. However, some variables and their

[6]This limitation will be removed in future hardware (The Haswell New Instructions provide gather [15] and MIC provides both gather and scatter [35]).

[7]Unstructured control flow (i.e. `goto` statements) is more difficult to support efficiently, though `ispc` does support `goto` in cases where it can be statically determined that all program instances will execute the `goto`.

associated computations do not need to be replicated across program instances. For example, address computations and loop iteration variables can often be shared.

Since CPU hardware provides separate scalar computation units, it is important to be able to express non-replicated storage and computation in the language. `ispc` provides a `uniform` storage class for this purpose, which corresponds to a single value in memory and thus, a value that is the same across all elements. In addition to the obvious direct benefits, the use of `uniform` variables facilitates additional optimizations as discussed in Section 6.1. It is a compile-time error to assign a non-uniform (i.e., "varying") value to a `uniform` variable.

In the absence of the `uniform` storage class, an optimizing compiler could convert varying variables into uniform variables when appropriate. (For example, in OpenCL or CUDA, all kernel parameters are effectively uniform and only variables that have values that are derived from the thread index are varying.) However, there are a number of reasons why having uniform as an explicit property of types in the language is important:

- **Interoperability with C/C++ data structures**: `uniform` is necessary to explicitly declare in-memory variables of just a single element, as is common in C/C++.
- **Performance transparency**: Treating `uniform` as an optimization rather than an explicit type property would make it difficult for the programmer to reason about performance. A small change to a program could inadvertently inhibit the optimization elsewhere resulting in significant and difficult-to-understand performance regressions.
- **Support for separate compilation**: Optimizations cannot cross separate-compilation boundaries, so at a minimum it must be possible to define a formal function parameter as `uniform`. But to guarantee that a call to such a function with a variable as an actual parameter is legal, `uniform` must be an explicit part of the type system. Otherwise, the legality of the function call would depend on the optimizer's behavior for the variable.

There is a downside to distinguishing between `uniform` and `varying` types in the type system: with separately compiled libraries of functions, to provide optimum performance it may be necessary to have multiple variants of functions that take different combinations of uniform and varying parameters.

The `uniform` and `varying` keywords were first used in the RenderMan shading language [11], but a similar distinction was made even earlier in general-purpose SIMD languages. To designate a SIMD variable, C* uses `poly`; Pixar's FLAP-C uses `parallel`; and MPL uses `plural`. CUDA and OpenCL do not provide this distinction; all variables are semantically varying in those languages.

5.2 Support For SOA Layout

It is well known that the standard C/C++ layout in memory for an "array of structures" (AOS) leads to sub-optimal performance for SIMD code. The top third of Figure 2 illustrates the issue using a simple structure, which corresponds to the `ispc` code below:

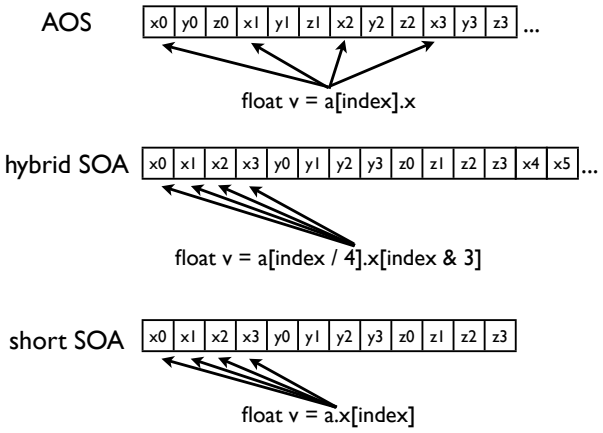

Figure 2: "Array of structures" layout (top), "hybrid structure of arrays" layout (middle), and "short structure of arrays" layout (bottom) of the example structure from Section 5.2. Reading data in an AOS layout generally leads to expensive gather instructions, while the SOA layouts lead to efficient vector load instructions.

```
struct Foo { float x, y, z; };
uniform Foo a[...] = { ... };
int index = ...;
float x = a[index].x;
```

Even if program instances access the elements of contiguous structures (i.e. the values of `index` are sequential over the program instances), the locations accessed are strided in memory and performance suffers from gathers (Section 3.5).

A better performing in-memory layout is "hybrid structure of arrays" (hybrid SOA layout), where the structure members are widened to be short arrays. On a system with a 4-wide vector unit, one might instead use the following `struct` declaration and access code:

```
struct Foo4 { float x[4], y[4], z[4]; };
uniform Foo4 a[...] = { ... };
int index = ...;
float x = a[index / 4].x[index & 3]
```

The corresponding memory layout is shown in the middle third of Figure 2. In many cases, accessing structure elements in hybrid SOA layout can be done with efficient vector load and store instructions.

The above syntax for declaring hybrid SOA layout and accessing hybrid SOA data is awkward and unnecessarily verbose; each element of `Foo4` has the same array width repeated in its declaration. If we want both SOA and AOS versions of the `struct`, we would have to declare two `struct`s with different types, which is undesirable. Furthermore, accessing elements of the structure is much more unwieldy to express than in the AOS case.

`ispc` addresses these problems and encourages more efficient hybrid SOA data layout by introducing a keyword `soa`, which modifies existing types to be laid out in SOA format. The `soa` qualifier converts primitive types (e.g. `float` or `int`) to fixed-sized arrays of that type, while for nested data structures or arrays, `soa` propagates downward through the data

structure until it reaches a primitive type. Traditional array indexing syntax is used for indexing into hybrid SOA data, while the code generated by the compiler actually implements the two-stage indexing calculation. Thus, use of the more efficient hybrid SOA layout can be expressed as follows in ispc:

```
struct Foo { float x, y, z; };
soa<4> struct Foo a[...] = { ... };
int index = ...;
float x = a[index].x;
```

Other than the soa<4> keyword, the code looks just like what one would write for an AOS layout, yet it delivers all of the performance benefits of hybrid SOA. As far as we know, these SOA capabilities have not been provided before in a general-purpose C-like language.

SOA layout also improves the performance of accesses to variables used by each program instance in a gang. We refer to this layout as a "short SOA layout" and illustrate it in the bottom of Figure 2. In the SPMD programming model, such variables should "look" scalar when they are used in expressions, so the indexing of such variables by programIndex should be implicit. Note that CUDA and OpenCL achieve similar behavior by storing such variables in a separate per-lane memory space. The keyword varying produces the desired behavior in ispc: it causes a structure to be widened to the gang size and to be implicitly indexed by programIndex. In the code below, after the expensive AOS structure loads have been performed by the indexing operation, the elements of fv are laid out contiguously in memory and so can be accessed efficiently.

```
uniform struct Foo a[...] = {...};
int index = ...;
varying Foo fv = a[index];
// now e.g. fv.x is contiguous in memory
fv.x = fv.y + fv.z; // looks scalar
```

varying structures of this form are also available in the VecImp and IVL languages designed concurrently to ispc [23].

The ability to conveniently but explicitly declare and access hybrid SOA and short SOA data structures is one of the major advantages of ispc over OpenCL when targeting CPU hardware. Note that languages that do not define memory layout as strictly as C/C++ (and hence, typically forbid pointers or restrict pointer arithmetic) may choose to optimize layout to SOA form even when the declaration appears to be AOS. For languages with strict layout rules, the compiler may still optimize layout to SOA form if it can guarantee that pointers are never used to access the data. However, these approaches provide less performance transparency than ispc's approach and cannot be used for zero-copy data structures that are shared with the C/C++ application.

5.3 Full C Pointer Model

ispc generalizes the full set of C pointer operations to SPMD, including both uniform and varying pointers, pointers to pointers, and function pointers. This feature is important for the expressability of algorithms that use complex pointer-based data structures in ispc and is also critical for allowing ispc programs to interoperate with existing application data structures. Often the code that builds these data structures is not performance-critical and can be left in C/C++, while the performance-critical portions of the application that read or update the data structure can be rewritten in ispc.

The distinction between uniform and varying data exists for both the pointer itself and for the data that is pointed to. (MasPar's C extensions make a similar distinction [29].) Thus, there are four kinds of pointers:

```
uniform float * uniform x;
varying float * uniform x;
uniform float * varying x;
varying float * varying x;
```

The first two declarations above are uniform pointers; the first is to uniform data and the second is to varying data. Both are thus represented as single scalar pointers. The second two declarations are varying pointers, representing a separate pointer for each program instance. Because all variables and dynamically allocated storage reside in a single coherent address space, any pointer can point to any data of the appropriate underlying type in memory.

In OpenCL and CUDA, all locally-declared pointers are in effect varying pointers to data, with additional limitations imposed by the fragmented memory architecture. CUDA supports function pointers and pointers to pointers, whereas OpenCL does not support function pointers and only supports certain cases of pointers to pointers.

5.4 Task Launch

In order to make it easy to fill multiple CPU cores with computation, ispc provides an asynchronous task launch mechanism, closely modeled on the "spawn" facility provided by Cilk [4]. ispc functions called in this manner are semantically asynchronous function calls that may run concurrently in different hardware threads than the function that launched them. This capability makes multi-core parallelization of ispc programs straightforward when independent computation is available; generally just a few lines of additional code are needed to use this construct.

Any complex multi-core C++ application typically has its own task system or thread pool, which may be custom designed or may be an existing one such as Microsoft's Concurrency Runtime, or Intel Thread Building Blocks. To interoperate with the application's task system, ispc allows the user to provide a callback to a task enqueue function, and then uses this callback to enqueue asynchronous tasks.

As in Cilk, all tasks launched from an ispc function must have returned before the function is allowed to return. This characteristic ensures parallel composability by freeing callers of functions from having to be aware of whether tasks are still executing (or yet to be executed) from functions they called. ispc also provides an explicit built-in sync construct that waits for tasks launched earlier in the function to complete.

Current GPU programming languages have no support for task launch from the GPU, although it is possible to implement a task system in "user space" in CUDA [1].

5.5 Cross-lane operations

One strength of SIMD capabilities on CPUs is the rich set of fast cross-lane operations. For example, there are instructions for broadcasting a value from one lane to all other lanes, and instructions for permuting values between lanes. ispc exposes these capabilities through built-in func-

tions that allow the program instances in a gang to exchange data. These operations are particularly lightweight thanks to the gang convergence guarantees described in Section 3.4.

5.6 Coherent Control Flow Hints

As described in Section 3.3, divergent control flow requires extra instructions on CPU hardware compared to regular control flow. In many uses of control flow, the common case is that all program instances follow the same control path. If the compiler had a way to know this, it could perform a number of optimizations, which are introduced in Section 6.5. `ispc` provides language constructs to express the programmer's expectation that control flow will typically be converged at a given point in the program. For each control flow construct, there is a corresponding "coherent" variant with the character "c" prepended to it. The following code shows `cif` in use:

```
float x = ...;
cif (x < 0) {
  // handle negative x
}
```

These coherent control flow variants do not affect program correctness or the final results computed, but can potentially lead to higher performance.

For similar reasons, `ispc` provides convenience `foreach` constructs that loop over arrays of one or more dimensions and automatically set the execution mask at boundaries. These constructs allow the `ispc` compiler to easily produce optimized code for the subset of iterations that completely fill a gang of program instances (see Section 6.6 for a description of these optimizations).

5.7 Native Object Files and Function Calls

The `ispc` compiler generates native object files that can be linked into the application binary in the same way that other object files are. `ispc` code can be split into multiple object files if desired, with function calls between them resolved at link time. Standard debugging information is optionally included. These capabilities allow standard debuggers and disassemblers to be used with `ispc` programs and make it easy to add `ispc` code to existing build systems.

`ispc`'s calling conventions are based on the platform's standard ABI, though functions not marked `export` are augmented with an additional parameter to provide the current execution mask. Functions that are marked `export` can be called with a regular function call from C or C++; calling a `ispc` function is thus a lightweight operation—it's the same as the overhead of calling to an externally-defined C or C++ function. In particular, no data copying or reformatting is performed, other than than possibly pushing parameters onto the stack if required by the platform ABI. While there are some circumstances where such reformatting could lead to improved performance, introducing such a layer is against our goals of performance transparency.

Lightweight function calls are a significant difference from OpenCL on the CPU, where an API call to a driver must be made in order to launch a kernel and where additional API calls are required to set each kernel parameter value.

6. EFFICIENT SPMD-ON-SIMD

There are a number of specialized optimizations that `ispc` applies to generate efficient code for SPMD on CPUs. We will show how the features introduced in Section 5 make a number of these optimizations possible. We focus on the optimizations that are unique to SPMD-on-SIMD; `ispc` also applies a standard set of traditional optimizations (constant folding, inlining, etc).

6.1 Benefits of "Uniform"

Having scalar `uniform` data types, as introduced in Section 5.1, provides a number of benefits compared to always having a separate per-program-instance storage location for each variable in the program:

- It reduces the total amount of in-memory storage used for data, which in turn can lead to better cache performance.
- Less bandwidth is consumed when reading and writing scalar values to memory.
- CPUs have separate register sets for scalar and vector values; storing values in scalar registers when possible reduces pressure on vector registers.
- CPUs can co-issue scalar and vector instructions, so that scalar and vector computations can happen concurrently.
- In the usual case of using 64-bit pointers, pointer arithmetic (e.g. for addressing calculations) is more efficient for scalar pointers than for vector pointers.
- Dereferencing a uniform pointer (or using a uniform value to index into an array) corresponds to a single scalar or vector memory access, rather than a general gather or scatter.
- Code for control flow based on `uniform` quantities can be more efficient than code for control flow based on non-uniform quantities (Section 6.2).

For the workloads we use for evaluation in Section 7, if all uses of the `uniform` qualifier were removed thus eliminating all of the above benefits, the workloads ran at geometric mean (geomean) 0.45x the speed of when `uniform` was present. The ray tracer was hardest hit, running at 0.05x of its previous performance, "aobench" ran at 0.36x its original performance without `uniform` and "stencil" at 0.21x.

There were multiple causes of these substantial performance reductions without `uniform`; the most significant were the higher overhead of non-uniform control flow and the much greater expense of varying pointer operations compared to uniform pointer operations. Increased pressure on the vector registers which in turn led to more register spills to memory also impacted performance without `uniform`.

6.2 Uniform Control Flow

When a control flow test is based on a `uniform` quantity, all program instances will follow the same path at that point in a function. Therefore, the compiler is able to generate regular jump instructions for control flow in this case, avoiding the costs of mask updates and overhead for handling control flow divergence.

Treating all uniform control flow statements as varying caused the example workloads to run with performance geomean 0.91x as fast as when this optimization was enabled. This optimization had roughly similar effectiveness on all of

the workloads, though the ray tracer was particularly hard-hit without it, running 0.65x as fast as it did without this optimization.

6.3 Benefits of SOA

We measured the benefits of SOA versus AOS layout with a workload based on a collision detection algorithm that computed collision points, when present, between two groups of spheres. We implemented this workload with AOS layout and then modified the implementation to also use SOA. By avoiding the gathers required with AOS layout, the SOA version was 1.25x faster than the AOS one.

6.4 Coherent Memory Access

After conventional compiler optimizations have been applied, it's often possible to detect additional cases where the program instances are actually accessing memory coherently [17]. The `ispc` compiler performs an additional optimization pass late in compilation that detects cases where all the instances, even if using "varying" indexing or pointers, are actually accessing the same location or consecutive locations.

When this optimization was disabled for the example workloads, performance was geomean 0.79x slower than when it is enabled. This optimization had a significant effect on the "stencil" workload, which ran 0.23x as fast when it was disabled.

6.5 Dynamic Control Flow Coherence

Recall from Section 3.3 that control flow is generally transformed by the compiler to data flow with masking, so that for example both the "if" and "else" clauses of an `if` statement are executed. In many such cases, the executing program instances will actually follow a converged control-flow path at runtime; for example, only the "else" clause might be actually needed. The code generated by the compiler can check for this case at points in the program where control flow could diverge. When it actually does not diverge, a more efficient code path can be followed. Performing this check can be especially helpful to performance for code paths that are rarely executed (corner case handling, etc.)

The `ispc` compiler uses the "coherent" control flow statements described in Section 5.6 to indicate when these additional tests should be performed. Performing this check for dynamic convergence at runtime gives two main advantages.

- It makes it possible to avoid executing instructions when the mask is "all off" and to jump over them.
- It gives an opportunity for dynamically reestablishing that the mask is "all on" and then taking a specialized code path for that case; the advantages of doing so are discussed in the following subsection.

Disabling the the coherent control flow statements caused the example workloads to run at geomean 0.85x their performance of when it is enabled. This optimization was particularly important for "aobench", which ran at 0.33x of regular performance without it. For the workloads that only have "uniform" control flow (e.g. Black-Scholes), disabling this optimization had no effect.

6.6 All On Mask

When it can be determined (statically or dynamically) that all of the program instances in a gang are executing at a point in the program, there are additional optimization opportunities. For example, scatters need to be "scalarized" on current CPUs; they are turned into a scalar store for each currently-executing program instance. In the general case, this scalarization requires a conditional test for each program instance before the corresponding store instruction. If all program instances are known to be executing, however, the per-lane mask check can be omitted.

There is furthermore some benefit to turning masked loads and stores to regular loads and stores even on systems that support masked memory instructions natively when the mask is known to be all on. Doing so can in turn allow those memory operations to be emitted as direct memory operands to instructions without needing to be first loaded into registers.

Disabling all of the optimizations that take advantage of statically determining that the execution mask is all on led to geomean 0.73x the performance of when it was enabled.

Sec.	Optimization	Perf. when disabled
6.1, 6.2	Uniform data & control flow	0.45x
6.2	Uniform control flow	0.91x
6.4	Gather/scatter improvements	0.79x
6.5	Coherent control flow	0.85x
6.6	"All on" mask improvements	0.73x

Table 1: Effect of individually disabling various optimizations (geometric mean over all of the example workloads)

7. RESULTS

We have measured performance of a variety of workloads written in `ispc`, comparing to serial, non-SIMD C++ implementations of the same algorithms. Both the C++ and `ispc` implementations received equivalent amounts of performance tuning. (In general, the `ispc` and C++ implementations are syntactically very similar.)

These workloads are all included in the open-source `ispc` distribution. They include two options pricing workloads, a third-order stencil computation, a ray tracer, a volume renderer, Mandelbrot set computation, and "aobench", a Monte Carlo rendering benchmark [9]. Most of these are not suitable for conventional auto-vectorization, due to complex data-dependent control flow and program complexity.

For the results reported here, we did a number of runs of each workload, reporting the minimum time. The results were within a few percent over each run. Other than the results on a 40-core system, results were measured on a 4-core Apple iMac with a 4-core 3.4GHz Intel® Core-i7 processor using the AVX instruction set. The basis for comparison is a reference C++ implementation compiled with a version of the clang compiler built using the same version of the LLVM libraries that are used by `ispc`.[8] Thus, the results should generally indicate the performance due to more effective use of the vector unit rather than differences in implementation of traditional compiler optimizations or code generation.

We have not performed direct comparisons between `ispc` and CPU OpenCL implementations in these evaluations; it

[8]We have also tested with various versions of gcc with essentially the same results.

Workload	1 core / 1 thread	4 cores / 8 threads
aobench	5.58x	26.26x
Binomial Options	4.39x	18.63x
Black-Scholes	7.43x	26.69x
Mandelbrot Set	5.85x	24.67x
Ray Tracer	6.85x	34.82x
Stencil	3.37x	12.03x
Volume Rendering	3.24x	15.92x

Table 2: Speedup of various workloads on a single core and on four cores of a system with 8-wide SIMD units, compared to a serial C++ implementation. The one core speedup shows the benefit from using the SIMD lanes of a single core efficiently, while the four core speedup shows the benefit from filling the entire processor with useful computation.

Workload	40 cores / 80 threads
aobench	182.36x
Binomial Options	63.85x
Black-Scholes	83.97x
Mandelbrot Set	76.48x
Ray Tracer	195.67x
Stencil	9.40x
Volume Rendering	243.18x

Table 3: Speedup versus serial C++ implementations of various workloads on a 40-core system with 4-wide SIMD units.

would be hard to interpret the results in that the effects of different compiler optimizations and code generators would be confounded with the effects of the impact of the language designs. Instead, we have focused on evaluating the performance benefit of various `ispc` features by disabling them individually, thus isolating the effect of the factor under evaluation.

Table 1 recaps the effects of the various compiler optimizations that were reported in Section 6.

7.1 Speedup Compared to Serial Code

Table 2 shows speedups due to `ispc`'s effective use of SIMD hardware and due to `ispc`'s use of task parallelism on a 4-core system. The table compares three cases for each workload: a serial non-SIMD C++ implementation; an `ispc` implementation running in a single hardware thread on a single core of the system; and an `ispc` implementation running eight threads on the four two-way hyper-threaded cores of the system. The four core performance shows the result of filling the entire processor with computation via both task-parallelism and SPMD. For the four-core results, the workloads were parallelized over cores using the tasking functionality described in Section 5.4.

7.2 Speedup Versus Intrinsics

The complexity of the example workloads (which are as much as 700 lines of `ispc` code) makes it impractical to also implement intrinsics-based versions of them for performance comparisons. However, a number of users of the system have implemented computations in `ispc` after previously implementing the same computation with intrinsics and seen good results—the examples we've seen are an image downsampling kernel (`ispc` performance 0.99x of intrinsics), a collision detection computation (`ispc` 1.05x faster), a particle system rasterizer (`ispc` 1.01x faster).

7.3 Speedup with wider vectors

We compared the performance of compiling the example workloads to use four-wide SSE vector instructions versus eight-wide AVX on a system that supported both instruction sets. No changes were made to the workloads' `ispc` source code. The geometric mean of the speedup for the workloads when going from SSE to AVX was 1.42x. Though this is not as good as the potential 2x speedup from the doubling of vector width, there are a number of microarchitectural details in the first generation of AVX systems that inhibit ideal speedups; they include the fact that the integer vector units are still only four-wide, as well as the fact that cache write bandwidth was not doubled to keep up with the widening of the vector registers.

7.4 Scalability on Larger Systems

Table 3 shows the result of running the example workloads with 80 threads on a 2-way hyper-threaded 40-core Intel® Xeon E7-8870 system at 2.40GHz, using the SSE4 instruction set and running Microsoft Windows Server 2008 Enterprise. For these tests, the serial C/C++ baseline code was compiled with MSVC 2010. No changes were made to the implementation of workloads after their initial parallelization, though the "aobench" and options pricing workloads were run with larger data sets than the four-core runs (2048x2048 image resolution versus 512x512, and 2M options versus 128k options, respectively).

The results fall into three categories: some (aobench, ray tracer, and volume rendering), saw substantial speedups versus the serial baseline, thanks to effective use of all of the system's computational resources, achieving speedups of more than the theoretically-ideal 160x (the product of number of cores and SIMD width on each core); again, the super-linear component of the speedups is mostly due to hyper-threading. Other workloads (both of the options pricing workloads and the Mandelbrot set workload), saw speedups around 2x the system's core count; for these, the MSVC compiler seems to have been somewhat effective at automatically vectorizing them, thus improving the serial baseline performance. Note, however, that these are the simplest of the workloads; for the more complex workloads the auto-vectorizer is much less effective.

The stencil computation saw a poor speedup versus the serial baseline (and indeed, a worse speedup than on a four-core system.) The main issue is that the computation is iterative, requiring that each set of asynchronous tasks complete before the set of tasks for the next iteration can be launched; the repeated ramp up and ramp down of parallel computation hurts scalability. Improvements to the implementation of the underlying task system could presumably reduce the impact of this issue.

7.5 Users

There have been over 1,500 downloads of the `ispc` binaries since the system was first released; we don't know how many additional users are building the system from source. Users

have reported roughly fifty bugs and made a number of suggestions for improvements to code generation and language syntax and capabilities.

Overall feedback from users has been positive, both from users with a background in SPMD programming from GPUs but also from users with extensive background in intrinsics programming. Their experience has generally been that `ispc`'s interoperability features and close relationship to C has made it easy to adopt the system; users can port existing code to `ispc` by starting with existing C/C++ code, updating it to remove any constructs that `ispc` doesn't support (like classes), and then modifying it to use `ispc`'s parallel constructs. It hasn't been unusual for a user with a bit of `ispc` experience to port an existing 500–1000 line program from C++ to `ispc` in a morning's work. From the other direction, many `ispc` programs can be compiled as C with the introduction of a few preprocessor definitions; being able to go back to serial C with the same source code has been useful for a number of users as well.

Applications that users have reported using `ispc` for include implementing a 2D Jacobi Poisson solver (achieving a 3.60x speedup compared to the previous implementation, both on a single core); implementing a variety of image processing operations for a production imaging system (achieving a 3.2x speedup, again both on single core); and implementing physical simulation of airflow for aircraft design (speedups not reported to us). Most of these users had not previously bothered to try to vectorize their workloads with intrinsics, but have been able to see substantial speedups using `ispc`; they have generally been quite happy with both performance transparency and absolute performance.

8. RELATED WORK

The challenge of providing language and compiler support for parallel programming has received considerable attention over many years. To keep the discussion of related work tractable, we focus on languages whose goal is high performance (or more precisely, high efficiency) programming of SIMD hardware. We further focus on general purpose languages (in contrast to domain-specific languages) with a particular emphasis on languages that are C-like. We do not discuss languages and libraries that are focused just on multi-core or distributed parallelism, such as OpenMP, TBB, Cilk, MPI, etc. even though some of these languages use an SPMD programming model.

8.1 Historical systems

In the late 1980s and early 1990s, there was a wave of interest in SIMD architectures and accompanying languages. In all of the cases we discuss, SIMD computations were supported with a true superset of C; that is, serial C code could always be compiled, but the SIMD hardware was accessible via language extensions. The Pixar FLAP computer had a scalar integer ALU and 4-wide SIMD floating-point ALU, with an accompanying extended-C language [24]. FLAP is also notable for providing hardware support for SIMD mask operations, like the MIC ISA and some modern GPUs. The Thinking Machines CM-1 and CM-2 and the MasPar MP-1 and MP-2 supercomputers used very wide SIMD (1000s of ALUs), programmed in the extended-C languages C* [33] and MPL [29] respectively.

All of these systems used a single language for both serial and parallel computations; had a single hardware program counter; and provided keywords similar to `ispc`'s `uniform` and `varying` to distinguish between scalar and SIMD variables. MPL provided vector control constructs with syntax similar to `ispc`, OpenCL, and CUDA; C* provided a more limited capability just for `if` statements. MPL provided generalized SIMD pointers similar to the ones in `ispc`, but each SIMD ALU and the scalar ALU had its own memory so these pointers could not be used to communicate data between units as they can in `ispc`. Both C* and MPL had sophisticated communication primitives for explicitly moving data between SIMD ALUs.

Clearspeed's C^n is a more recent example of this family of languages; the paper describing it has a good discussion of design trade-offs [26].

8.2 Contemporary systems

CUDA is a SPMD language for NVIDIA GPUs [31] and OpenCL is a similar language developed as an open standard, with some enhancements such as API-level task parallelism designed to make it usable for CPUs as well as GPUs [10, 19, 34]. At a high level, the most important differences between these languages and `ispc` are that `ispc`'s design was not restricted by GPU constraints such as a separate memory system, and that `ispc` includes numerous features designed specifically to provide efficient performance on CPUs. All three languages are C-like but do not support all features of C. `ispc` and CUDA have some C++ features as well.

The difference in hardware focus between CUDA/OpenCL and `ispc` drives many specific differences. OpenCL has several different address spaces, including a per-SIMD-lane memory address space (called "private"), and a per-work-group address space (called "local") whereas `ispc` has a single global coherent address space for all storage. OpenCL and CUDA also have complex APIs for moving data to and from a discrete graphics card that are unnecessary in `ispc`. `ispc` has language-level support for task parallelism, unlike OpenCL and CUDA. CUDA and OpenCL lack `ispc`'s support for "uniform" variables and convenient declaration of structure of arrays data types. Although these features are less important for performance on GPUs than on CPUs, we believe they would provide some benefit even on GPUs.

There are several implementations of CUDA and OpenCL for CPUs. Some do not attempt to vectorize across SIMD lanes in the presence of control flow [10, 36]. Intel's OpenCL compiler does perform SIMD vectorization [34], using an approach related to Karrenberg et al.'s [17] (who also applied their technique to OpenCL kernels.)

Parker et al.'s RTSL system provided SPMD-on-SIMD on current CPUs in a domain-specific language for implementing ray tracers [32].

Microsoft's C++ AMP [30] provides a set of extensions to C++ to support GPU programming. As with CUDA and OpenCL, its design was constrained by the goal of running on today's GPUs. It is syntactically very different from CUDA, OpenCL, and `ispc` because of its choice of mechanisms for extending C++.

The UPC language extends C to provide an SPMD programming model for multiple cores [5]. UPC includes mechanisms for scaling to very large systems that lack hardware memory coherence, but the language was not designed to target SIMD parallelism within a core and as far as we know it has never been used for this purpose.

8.3 Concurrently-developed systems

The IVL and VecImp languages described in a recent paper are similar to `ispc` in a number of ways [23]; they were developed concurrently with `ispc` with some cross-pollination of ideas. These three languages are the only C-like general-purpose languages that we are aware of that provide a mechanism for creating a structure-of-arrays variant of a previously-declared struct data type.

There are substantial differences in emphasis between the VecImp/IVL paper and this work. The VecImp/IVL paper focuses on describing the language and formally proving the soundness of the type system, whereas we focus on justifying and quantitatively evaluating language features such as uniform variables and structure-of-arrays support. IVL and its evaluation focus on the MIC architecture, whereas `ispc` focuses on the SSE and AVX architectures which have less dedicated ISA support for SPMD-style computation. This paper also introduces and analyzes the compiler optimizations required to reap the full benefit of language features such as uniform variables.

There are a variety of other detailed differences between `ispc`, IVL, and VecImp. For example, IVL supports function polymorphism, which is not currently supported in `ispc`, and `ispc`'s pointer model is more powerful than IVL's. `ispc` uses LLVM for code generation, but the IVL compiler generates C++ code with intrinsics. `ispc` is the only one of the three languages with an implementation available for public use.

The Intel C/C++ compiler provides an "elemental functions" extension of C++ that is intended to provide SPMD as an extension of a full C++ compiler [16]. Its language functionality for SPMD is more limited than `ispc`'s; for example its equivalent of `uniform` can only be applied to function parameters and there is no general facility for creating SOA types from AOS types. It has been demonstrated that its capabilities can be used to achieve good utilization of SIMD units [20].

9. CONCLUSION

We have presented `ispc`, a SPMD language for programming CPU vector units that is easy to adopt and productive to use. We have shown that a few key language features–uniform data types, native support for SOA structure layout, and in-language task launch–coupled with a series of custom optimization passes make it possible to efficiently execute SPMD programs on the SIMD hardware of modern CPUs. These programs can effectively target the full capabilities of CPUs, executing code with performance essentially the same as hand-written intrinsics. Support for `uniform` types is particularly important; our experiments showed that this capability provides over a 2x performance improvement.

In the future, we plan to further refine the `ispc` language, eliminating remaining differences with C and adding convenience features like polymorphic functions. We are already adding support for the MIC architecture, which is an attractive target due to its 16-wide SIMD and good ISA support for SPMD execution.

Experience with `ispc` suggests a number of avenues for improving future hardware architectures. For conventional CPUs, improved support for masking and scatter would be desirable, and extending vector units to operate on 64-bit integer values at the same performance as when operating on 32-bit integer values (for vector pointer addressing calculations) may be helpful.

The decision to include both scalar and SIMD computation as first-class operations in the language may be applicable to other architectures. For example, AMD's forthcoming GPU has a scalar unit alongside its vector unit [27] as does a research architecture from NVIDIA [18]. Such architectures could have a variety of efficiency advantages versus a traditional "brute force" SIMD-only GPU implementation [6]. More broadly, many of the available approaches for achieving high SIMD efficiency can be implemented in different ways: by the programmer/language, by the compiler, or by the hardware. In the power-constrained environment that limits all hardware architectures today, we expect continued exploration of the complex trade offs between these different approaches.

Acknowledgments

The parser from the (non-SPMD) C-based "nit" language written by Geoff Berry and Tim Foley at Neoptica provided the starting-point for the `ispc` implementation; ongoing feedback from Geoff and Tim about design and implementation issues in `ispc` has been extremely helpful. Tim suggested the "SPMD on SIMD" terminology and has extensively argued for the advantages of the SPMD model.

We'd like to specifically thank the LLVM development team; without LLVM, this work wouldn't have been possible. Bruno Cardoso Lopes's work on support for AVX in LLVM was particularly helpful for the results reported here.

We have had many fruitful discussions with Ingo Wald that have influenced the system's design; `ispc`'s approach to SPMD and Ingo's approach with IVL languages have had bidirectional and mutually-beneficial influence. More recently, discussions with Roland Leißa and Sebastian Hack about VecImp have been quite helpful.

We appreciate the support of Geoff Lowney and Jim Hurley for this work as well as Elliot Garbus's early enthusiasm and support for it. Thanks to Kavyvon Fatahalian, Solomon Boulos, and Jonathan Ragan-Kelley for discussions about SPMD parallel languages and SIMD hardware architectures. Discussions with Nadav Rotem about SIMD code generation and LLVM as well as discussions with Matt Walsh have also directly improved the system. Ali Adl-Tabatabai's feedback and detailed questions about the precise semantics of `ispc` have been extremely helpful as well.

Thanks to Tim Foley, Mark Lacey, Jacob Munkberg, Doug McNabb, Andrew Lauritzen, Misha Smelyanskiy, Stefanus Du Toit, Geoff Berry, Roland Leißa, Aaron Lefohn, Dillon Sharlet, and Jean-Luc Duprat for comments on this paper, and thanks to the early users of `ispc` inside Intel—Doug McNabb, Mike MacPherson, Ingo Wald, Nico Galoppo, Bret Stastny, Andrew Lauritzen, Jefferson Montgomery, Jacob Munkberg, Masamichi Sugihara, and Wooyoung Kim—in particular for helpful suggestions and bug reports as well as for their patience with early versions of the system.

10. REFERENCES

[1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[2] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. POPL '83*.

[3] American National Standards Institute. *American National Standard Programming Language C, ANSI X3.159-1989*, 1989.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, July 1995.

[5] W.-Y. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. In *Proc. of 17th Annual Intl. Conf. on Supercomputing*, pages 63–73, 2003.

[6] S. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Proc. of the 2009 Intl. Conf. on Parallel Processing*, Euro-Par'09.

[7] F. Darema, D. George, V. Norton, and G. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1), 1988.

[8] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept. 1972.

[9] S. Fujita. AOBench. `http://code.google.com/p/aobench`.

[10] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. PACT '10.

[11] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. *SIGGRAPH Comput. Graph.*, 24:289–298, September 1990.

[12] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Passing a language through the eye of a needle. *ACM Queue*, 9(5).

[13] Intel. Intel SPMD Program Compiler documentation. `http://ispc.github.com/documentation.html`.

[14] Intel. Intel SPMD Program Compiler User's Guide. `http://ispc.github.com/ispc.html`.

[15] Intel. Intel advanced vector extensions programming reference. June 2011.

[16] Intel. *Intel Cilk Plus Language Extension Specification Version 1.1*, 2011. Online document.

[17] R. Karrenberg and S. Hack. Whole Function Vectorization. In *CGO 2011*.

[18] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *IEEE Micro*, 31:7–17, Sept–Oct 2011.

[19] Khronos OpenCL Working Group. *The OpenCL Specification*, Sept. 2010.

[20] C. Kim, N. Satish, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Closing the ninja performance gap through traditional programming and compiler technology. Technical report, Intel Corporation, Dec 2011.

[21] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of CGO '04*, Mar 2004.

[22] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proc. ISCA 2010*.

[23] R. Leißa, S. Hack, and I. Wald. Extending a C-like language for portable SIMD programming. In *PPoPP*, Feb 2012.

[24] A. Levinthal, P. Hanrahan, M. Paquette, and J. Lawson. Parallel computers for graphics applications. *SIGPLAN Not.*, October 1987.

[25] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, Mar–April 2008.

[26] A. Lokhmotov, B. Gaster, A. Mycroft, N. Hickey, and D. Stuttard. Revisiting SIMD programming. In *Languages and Compilers for Parallel Computing*, pages 32–46. 2008.

[27] M. Mantor and M. Houston. AMD graphics core next: Low power high performance graphics and parallel compute. *Hot3D, High Performance Graphics Conf.*, 2011.

[28] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, July 2003.

[29] MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatible MPL) Reference Manual, Software Version 3.0*, July 1992.

[30] Microsoft Corporation. *MSDN Library: Overview of C++ Acceleration Massive Parallelism (C++ AMP)*, 2011. Online preview documention, visited Dec 11.

[31] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6:40–53, March 2008.

[32] S. G. Parker, S. Boulos, J. Bigler, and A. Robison. RTSL: a ray tracing shading language. In *Proc. of the 2007 IEEE Symp. on Interactive Ray Tracing*, 2007.

[33] J. Rose and J. G. Steele. C*: An extended C language for data parallel programming. In *Proc. of the Second Intl. Conf. on Supercomputing*, May 1987.

[34] N. Rotem. Intel OpenCL SDK vectorizer. In *LLVM Developer Conf. Presentation*, Nov. 2011.

[35] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, August 2008.

[36] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In *Proc. 21st Int'l Workshop on Languages and Compilers for Parallel Computing*, 2008.

[37] M. Wolfe, C. Shanklin, and L. Ortega. *High-Performance Compilers for Parallel Computing*. Addison Wesley, 1995.