Extending C0 to support data parallel operations

Rokhini Prabhu* and Harry Gifford†

Carnegie Mellon University

Abstract

There is a trend of ever increasing data parallelism on modern hardware[5]. To date compilers for C and its derivatives have been poor in leveraging this potential parallelism, partially due to the 'wild west' style design of C.

In this paper we propose and evaluate an array like C0 extension that is inherently parallel. By limiting what operations the programmer can express we can make assumptions about the programmers intent which allows for program transformations to efficiently map programs into vectorized x86.

Our implementation of parallel vectors exploits SIMD data parallel instructions in order to exploit modern CPU architectures. In future we could extend these to the GPU and also multiple cores.

1 Introduction

Consider the following C0 function:

```
int[] A = // array of size 1000
int[] B = // array of size 1000
int[] result = // array of size 1000

for (int i = 0; i < 1000; i++)
    result[i] = A[i] + B[i];
```

This is a verbose way of expressing a `map`-like operation. It is clear that there is ample opportunity for parallelism in this operation, and a good compiler may well be able to vectorize this code. However, it is not clear at what point a compiler would no longer be able to vectorize a loop. This means that programmers are largely at the mercy of the compiler as to whether or not their code will run efficiently on modern hardware.

This uncertainty is not desirable for either the programmer or the compiler writer. We would therefore like to be able to express parallelism in the source language. This will make our lives easier as compiler writers, while at the same time documenting parallelism in the source language.

In order to accomplish this we extend C0 by adding a vectorized array-like type.

1.1 Design goals

There has been a large amount of work on creating programming languages for expressing parallelism and two orthogonal design decisions seem to arise:

1. **Freedom**. In this design framework we allow the programmer as much freedom in expression as possible. With this design decision the compiler is explicitly aware that it is free to parallelize a section of code, whether or not such a transformation is valid. Thus the burden of correctness lies with the programmer. Many current languages for expressing parallelism follow this philosophy (such as CUDA, OpenCL, OpenMP) [2].
2. **Safety**. We restrict the domain of the operations that the programmer can express. As the language designer we can specifically choose to only allow valid parallel transformations to take place. This design decision can be seen in many modern GPU oriented languages (such as Matlab, GLSL, Brook, GraphLab) [1] [3] [4].

Freedom has the advantage that it is easy to modify existing code to allow for parallelism, and it also has the possibility of allowing for more expressiveness than option 2.

Safety has the advantage that any parallel code you write is guaranteed to output the same result regardless of what order it is evaluated in.

In keeping with the standard C0 design philosophy we opted to go with option 2, since one of the core design philosophies of C0 is safety. In our eyes, the expressiveness of option 1. does not outweigh the lack of guarantees. As an example, consider the following CUDA snippet (CUDA is a C like language that largely followed design 1.) for performing a prefix sum:

```
// CUDA has an implicit for loop,
// where blockidx and threadidx act
// as the indexing.

int* A = // data

int idx = blockidx.x*blockdim.x +
          threadidx.x

if (idx > A_len || idx <= 0) return;

A[idx] = A[idx-1] + A[idx];
```

Since this code is executed in parallel in lockstep, it is not at all obvious what the output of this code would be. How would memory be updated?

Instead, we can supply some built parallel operators for expressing data parallel operations. For example, consider the following:

*rokhinip@gmail.com
†hgifford@cmu.edu

```
int add(int a, int b)
{
    return a + b;
}

...

    int[] A = // data
    A = scan(add, A);
```

This is a far more concise expression of the same code. Since the code is high level, it is more restrictive in terms of the operations that can be performed, but on the other hand it gives much more information about the programmer's intent to the compiler, which means it is much easier to generate efficient code.

2 Language Design

We will describe the language. The grammar is given in Figure 1. Essentially, we allow compile time constant arrays, which can be indexed in the usual way, but with added support for array splicing, so that multiple contiguous elements can be accessed in parallel.

Vectors can be thought of as 'small types', and so they can be passed to and from functions (although this can be slow for real use cases). Vectors support many types of primitive element-wise operations, such as addition, multiplication etc.

For example, below is a set of simple map like operations that can be performed with vectors:

```
// vecs can be passed to
// and from functions.
int square(vec3 a)
{
    // parallel reductions
    return reduce('+', a*a);
}

int main()
{
    // vector constructors.
    vec100 A = vec100(10);
    vec3 b = sqaure(vec3(3, 5, 1));

    // vector splice, must be consts.
    A[5:8] = b;
    // element-wise multiply.
    A = B*A;
    int a = A[0];
    return a;
}
```

We decided to only support compile time constant vectors since this makes it possible to type check vectors against one another. Another reasonable design decision would have been to allow variable length vectors, but we ultimately decided that the added type security was more useful to us. We can easily envisage that parallel arrays could be added to the language too, which could ultimately replace standard arrays, at least for small types.

⟨*Nat*⟩ ::= 1 | 2 | 3 | ...

⟨*Type*⟩ ::= ...
| **vec**⟨nat⟩

⟨*Expr*⟩ ::= ...
| **vec**⟨nat⟩(⟨*arg-list*⟩)
| ⟨*Expr*⟩[⟨nat⟩:⟨nat⟩]

⟨*L-Value*⟩ ::= ...
| ⟨*L-Value*⟩[⟨nat⟩:⟨nat⟩]

Figure 1: Extension of C0 grammar to support parallel vectors.

We also added support for reduce style operations, which take a primitive operator (any operator that works on integer types) and applies it pairwise in some undefined order and outputs the pairwise application of all the elements.

If you are familiar with functional languages, then `reduce('op', A)` is roughly equivalent to `reduce op A (Array.elem A 0)`, where `(Array.elem A 0)` is the first element of the array. This can in fact be implemented very efficiently on x86 using instructions such as `hadd`.

Unfortunately, we did not get time to implement an efficient version of this reduction in our code, but our language extension is built with the assumption that we can get fast `map` and `reduce` style operations.

2.1 Static Semantics

These were the following additional static semantic rules which were added as a result of the language addition.

In the following rules, we explicitly require that n, a and b are compile-time constants.

$$\frac{n : int,\ \Gamma(x) = vec\langle n\rangle}{\Gamma \vdash x : vec\langle n\rangle}$$

.

$$\frac{n : int,\ \forall x \in arglist.\ x : int}{\Gamma \vdash vec(n)\langle arglist\rangle : vec(n)}$$

$$\frac{\Gamma \vdash e : vec(N),\ a : int,\ b : int,\ 0 \leq b - a \leq N,\ b - a = c}{\Gamma \vdash e[a : b] : vec(c)}$$

$$\frac{\Gamma \vdash e_1 : vec(n), e_2 : vec(n),\ op :\ int * int \longrightarrow int}{\Gamma \vdash e_1\ op\ e_2 : vec(n)}$$

$$\frac{\Gamma \vdash e : vec(n)\ op :\ int \longrightarrow int}{\Gamma \vdash op\ e : vec(n)}$$

$$\frac{\Gamma \vdash e_1 : vec(n), e_2 : int}{\Gamma \vdash e_1[e_2] : int}$$

$$\frac{\Gamma \vdash e_1 : vec(n), e_2 : vec(n)}{\Gamma \vdash e_1 == e_2 : bool}$$

$$\frac{\Gamma \vdash e_1 : vec(1), e_2 : int}{\Gamma \vdash e_1 == e_2 : bool}$$

2.2 Dynamic Semantics

The following were the additional dynamic semantic rules which were added as a result of the language addition. Observe that since our language is an extension of L3, none of these rules modify the heap at any point since `vectors` are statically allocatd.

$$\frac{\Gamma\eta \vdash e_1 \downarrow v_1, e_2 \downarrow v_2, v_1 : \texttt{vecn}, \forall\, i < n.\ v_1[i]\ op\ v_2[i] \downarrow v[i]}{\Gamma, \eta \vdash e_1\ op\ e_2 \downarrow v}$$

$$\frac{\Gamma\eta \vdash e_1 \downarrow v_1, e_2 \downarrow v_2, v_1 : \texttt{vecn}, \exists i < n.\ v_1[i]\ op\ v_2[i] \uparrow exn}{\Gamma, \eta \vdash e_1\ op\ e_2 \uparrow exn}$$

$$\frac{\Gamma\eta \vdash e_1 \downarrow v_1,\ e_2 \downarrow v_2,\ e_1 : \texttt{vecn}, 0 \leq e_2 < n}{\Gamma, \eta \vdash e_1[e_2] \downarrow v}$$

$$\frac{\Gamma\eta \vdash e_1 \downarrow v_1,\ e_2 \downarrow v_2,\ e_1 : \texttt{vecn}, (e_2 < 0)||(e_2 \geq n)}{\Gamma, \eta \vdash e_1[e_2] \uparrow exn}$$

Two of the most important things to discuss with respect to the dynamic specification of our language are out of bounds exceptions and division/mod exceptions. In safe mode, a vector operation that contains an out of bounds exception is guaranteed to raise an exception. In unsafe mode no bounds check is performed.

Division/mod exceptions are also guaranteed to be raised if the exception occurs for any of the vector elements, but again, only in safe mode.

One thing to note is that the exact exception that will be raised is not specified. If a vector has numerical exceptions as well as access violations the language does not specify which exception will be raised.

Out of memory accesses are not specified by our language. We assume that there is infinite memory and that arbitrary sizes of vectors can be created.

3 Implementation Details

To evaluate the usefulness of our language design we implemented our extension into our L5 compiler.

We decided to target LLVM in order to ease the generation of vector intrinsics. That said, in discussing performance we will focus exclusively on the x86 architecture.

3.1 Structure

Our Lab 5 compiler had the following structure:

1. Parser
2. Elaboration - This module elaborated certain constructs of the source language to an alternative equivalent form in our AST.
3. Typechecker - This module verified the static semantics of the L4 language.
4. Preprocess - This module elaborated array accesses and struct accesses into a common memory access construct in terms of a base pointer, an offset (in terms of bytes) and the size to read. Therefore, after this point in the pipeline, the AST is blind to whether the memory access came from a struct access or an array access.
5. CodeGen - The set of modules under CodeGen translate the AST to Abstract Assembly (in 3-operand) after performing optimizations like tail-call optimization.
6. Register Allocation
7. X86AssemGen - This module converted our Abstract Assembly into x86 assembly.

Our Lab 6 compiler, had far more complex structure. Since we performed the language extension on top of L3, in order to maintain backward compatability, we required the user to specify a `-V` flag if they are compiling codes which use `vectors`. The structure of our Lab 6 compiler is as follows:

1. Parser
2. Elaboration
3. Typechecker
4. Preprocess
5. CodeGen
   (a) AST to IR tree - We perform AST level constant folding, eliminate `if (false) ...` and `while (false) ...` statements, and perform a few other optimizations.
   (b) IR tree to SSA - In this module, we generate 3-operand IR in SSA form with parametrized labels
   (c) Optimize - We perform our 3 main optimizations here: Iterated constant and copy propagation and folding, dead code elimination, and tail call optimization.
   (d) deSSA
   (e) SSA to SSA with Phi Nodes - In this module, we translate parametrized labels to phi nodes.

6. LLVM AssemGen - If the `-V` flag is turned on, the code is not deSSAed after optimizations, but instead passed directly to the LLVM AssemGen module. This module then converts our 3 operand IR form to LLVM IR form and writes it to a `.ll` file

7. Register Allocation - If the `-V` flag is of, after optimizations, deSSA-ed code passes through this module.

8. X86AssemGen

3.2 Implementation of Vectors

In this section we will discuss how vectors are implemented.

We implemented vectors by using the built in SIMD intrinsics offered by LLVM. These allowed for a natural conversion from our intrinsics to x86 intrinsics. LLVM and x86 offer almost all primitive operations for parallel lock-step vectorized operations, which helped us greatly.

3.2.1 LLVM to x86 Conversion

LLVM converts its vector types into AVX SIMD intrinsics on the computer we tested on (Mobile Core i7 dual core). So a single vector would get allocated as an aligned stack array and then when operations were performed on the chunks of the vector were pulled into memory and the operation performed.

We did not experiment with multiple cores, but it is certainly possible to make use of many core machines using our language implementation. The fact the array sizes are known at compile times aids this, as it means we know roughly how much work will be executed and whether or not it is worth passing off to multiple cores.

3.3 LLVM Conversion

3.3.1 if-then-else in LLVM

In our 3 operand IR, we represent ifs as follows: `If op (src1, src2) label` where `label` is a branch to the `then` case if `src1 op src2` $\cong$ `True`. The branch to the `else` case is implicit since we simply fall through and continue execution on the next line. This was designed as such to be in line with the x86-64 architecture.

However, in LLVM, the `branch` instruction requires explicit labelling of the branches to jump to, for both the `then` and the `else` case. Therefore, we code-gen an if-then-else statement into the following: `If op (src1, src2) l, GoTo l'`. This requires the LLVMAssemGen to traverse each function 2 instructions at a time so that we can have the following conversion:

`toLLVMInstr <If op (src1, src2) l, GoTo l'>` $\cong$ `t = icmp op src1 src2, br t l l'`

3.3.2 Single exits from basic blocks

Since our if-then-else statements did not have explicit edges from the true and false clauses to an end block, we had to insert a `br label end_condition` jump. However, this could sometimes lead to instructions of the following form:

```
br label l
br label l'
```

LLVM however rejects such code as being malformed input since the second jump is unreachable. Similarly, if there is a `return` statement in the if-block, then we might generate code as follows:

```
l0:
br i1 cond label % l1 label % l2

l1:
ret i32 42
br label % l3

l2:
br label % l3

l3:
p = phi [%x, l1] [%x, l2]
```

Similarly, the above code is malformed since `l3` cannot be reached from `l1` for the unconditional jump cannot be reached after the return statement. Therefore, in order to ensure that blocks were single-exit only, we performed dead-code elimination on each basic block by eliminating all code after the first `branch` or a `ret` since it is unreachable anyway.

3.3.3 Simple assignment operations

There is no clear map, from a move instruction in our IR, to one in LLVM IR. While almost cases of moves were handled by constant and copy propagation, if we still had any move instructions left (for whatever reason), it generates the following LLVM assembly code: `toLLVMInstr (d` $\Leftarrow$ `s)` $\cong \tau$ `d = select i1 true,` τ `s,` τ `0`. This case however, rarely came up in our LLVM code.

4 Evaluation

In order to evaluate our results we implemented and tested our code on a variety of small, but descriptive tests, in order to show how our language extension can be used, where it excels and where it is weak.

Since we have a new language feature, we have tested the correctness of our compiler in compiling this new language feature by writing test cases using `vectors`. We therefore wrote a suite of test cases to verify the correctness of our compiler in compiling vectors.

In addition, we have also used some of those tests as benchmarks to show the performance instructions obtained

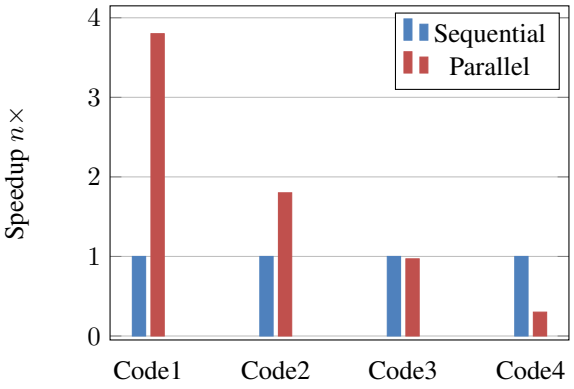

Figure 2: Speedup on AVX Intel Core i7

by using vectors for they are compiled to use SIMD instructions. By comparing the time taken to perform operations done in those tests to equivalent operations done with memory allocation on the heap, we were able to compare differences in performance from using vectors.

The tests can be seen in the appendix of this paper.

4.1 Tests

All of the tests ran the same piece of parallel code in a tight loop. This was to remove the effect of allocating memory etc. and focus on the SIMD instructions themselves. We had to specify specific compilation flags in order to make sure that LLVM did not attempt to optimize away operations.

For the Sequential test, we simply converted all vectors into heap allocated arrays, and convert the vectorized operations into simple loops.

As can be seen in the above figure, we get very good speedup on Code1, which was simply a multiply of two very large vectors, component-wise. This is close to optimal, in fact, since the SIMD width of Intel AVX instructions is 4 wide, so the speedup we see is as expected.

Code2 gets a reasonable speedup, but is hindered by the fact that our reduction is not parallelized and so we do not get good speedup.

Code3 is simply a reduction and so the generated code is near identical, leading to very similar speedups.

Code4 is a pathological case where we must continually unpack and pack the vectors, and so we get a very significant slow down as compared to the regular sequential operations.

This shows that it is still necessary for the programmer to know when they want parallelism and when they do not, as there is a significant cost associated with using vectors for sequential operations.

5 Future work

Whilst we get good results from our language extension there is plenty more we could do.

For example, it would be good to allow nested data parallelism, so that recursive functions could be parallelized into SIMD intrinsics.

We also do not have good support for gather type operations, which are well supported on modern hardware. Briefly, we would like to collect a set of non-contiguous vector elements and make them contiguous. Gather can be used to efficiently implement the functional `filter` on data parallel vectors. One example might be to collect all the odd elements of a vector into a new vector.

It would also be interesting to see how well our code would perform over multiple cores and perhaps even on GPUs.

6 Appendix

The code used for evaluation is below:

Code1:

```
// highly parallel map test.
int main()
{
    vec10000 A = vec10000(1.0);
    vec10000 B = vec10000(2.0);
    for (int i = 0; i < 100; i++)
        A *= B;
    return A[0];
}
```

Code2:

```
// map and reduce test.
int main()
{
    vec10000 A = vec10000(1.0);
    vec10000 B = vec10000(2.0);
    vec10000 C = vec10000(1.0);
    for (int i = 0; i < 100; i++)
    {
        A = B*C;
        reduce('+', A);
    }
    return A[0];
}
```

Code3:

```
// map and reduce test.
int main()
{
    vec10000 A = vec10000(1.0);
    vec10000 B = vec10000(2.0);
    vec10000 C = vec10000(1.0);
    int res = 0;
    for (int i = 0; i < 100; i++)
    {
        res = reduce('+', A);
    }
    return res;
}
```

Code4:

```
// sequential access test.
int main()
{
    vec10000 A = vec10000(1.0);
    vec10000 B = vec10000(2.0);
    vec10000 C = vec10000(1.0);
    int res = 0;
    for (int j = 0; j < 100; j++)
        for (int i = 0; i < 10000; i++)
        {
            A[i] = B[i]*C[i];
        }
    return A[i];
}
```

References

[1] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 777–786.

[2] LEE, J., KIM, J., SEO, S., KIM, S., PARK, J., KIM, H., DAO, T. T., CHO, Y., SEO, S. J., LEE, S. H., CHO, S. M., SONG, H. J., SUH, S.-B., AND CHOI, J.-D. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 193–204.

[3] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[4] MARROQUIM, R., AND MAXIMO, A. Introduction to gpu programming with glsl. In *Proceedings of the 2009 Tutorials of the XXII Brazilian Symposium on Computer Graphics and Image Processing* (Washington, DC, USA, 2009), SIBGRAPI-TUTORIALS '09, IEEE Computer Society, pp. 3–16.

[5] TARDITI, D., PURI, S., AND OGLESBY, J. Accelerator: Using data parallelism to program gpus for general-purpose uses. *SIGOPS Oper. Syst. Rev. 40*, 5 (Oct. 2006), 325–335.