NumbaSummarizer: A Python Library for Simplified Vectorization Reports

Neftali Watkinson*, Preston Tai†, Alexandru Nicolau‡ and Alexander Veidenbaum§
Department of Computer Science
University of California, Irvine
Irvine, United States of America
*watkinso@uci.edu, †prestoht@uci.edu, ‡nicolau@ics.uci.edu, §alexv@ics.uci.edu

***Abstract*—Python is a very popular programming language and has been adopted by many learning institutions as the first or only programming language taught to students. While its simple syntax and gentle learning curve makes it a very attractive option for new programmers, among its main drawbacks are that it obviates many key concepts of programming and computer architecture as well as rarely optimizing code to modern architectures.**

There are tools that provide optimizing capability for Python programmers. *Numba* is a JIT compiler for Python that among other things, optimizes Python and Numpy functions for better performance. *Numba* can exploit automatic parallelism and vectorization.

In this paper we present NumbaSummarizer, a wrapper for Numba's vectorization module that works as a vectorization reporting tool. Students can use it to identify functions that are exploiting SIMD instructions. The main purpose of this tool is to teach students about Parallel Computing, Vectorization, Data Dependence, and Loop Transformations.

We evaluated the use of NumbaSummarizer with 64 students during a course on intermediate programming. Most students had little to no background in Computer Architecture nor Parallel Computing. When asked to expose parallelism in a set of 8 loops, 95% percent of the students were able to transform and optimize at least 6 of these loops. When testing conceptual understanding, they scored an average of 95% for parallel computing concepts, and 62% for dependence analysis.

***Index Terms*—SIMD, Numba, Python, Vectorization**

I. Introduction

Python is a popular language thanks to its high abstraction and simple syntax. Many universities are adopting Python as their main programming language to introduce students to programming and computer science. For example, the University of California, Irvine (UCI), requires that students take three quarters of Python programming. While C, C++ and Java are also offered, these are electives and required for some upper-division courses. However, there are relevant drawbacks in using a programming language that obviates many of the intricacies of programming. For example, Alzahrani et al. [1] discovered that for many basic programming tasks, Python users struggled more when compared to students familiarized with C++.

Another issue is that Python is not typically a performance oriented language. CPython, Python's default interpreter, is locked to a single thread due to its Global Interpreter Lock, meant to keep programs being thread safe and make library integration relatively easy. Because of this, Python lacks capabilities to take advantage of modern architectural features.

The inclusion of multiple cores on a single chip, adoption of Single Instruction Multiple Data (SIMD) or Vector instructions that exploit low level parallelism, and the pervasiveness of Graphical Processing Units (GPU), categorically guarantee that any serious programmer will eventually interact with some level of parallel computing. This introduces a shift on how we learn and teach programming. As recognized by Prasad et al. [15] and Rague [16], instead of focusing on serial programming, parallel programming should be introduced early in Computer Science curricula. Additionally, many experts in parallel computing agree that dependence analysis for the purpose of exploiting vectorization is a useful way to introduce students to parallelism [20, 2, 14, 8].

Using Python to teach parallel computing allows for students to easily understand the tools and necessary coding skills for manual optimization of their code without the need to learn complex syntax [5, 21]. There are tools that allow programmers exploit high performance optimizations while still keeping the practicality of Python. The work of Marowka [11], reviews some of the most popular tools and workarounds to Python's drawbacks, noting that Numba [9] is a good option for exploiting optimizations. Numba is a Just-In-Time (JIT) compiler that uses LLVM [10] as a back end to compile entire functions. This allows for code optimizations and use of automatic parallelism. Thanks to Numba's simple interface that uses decorators, it can be easily adopted in the classroom. One of the major drawbacks that Numba has, is that it functions as a "black box". Programmers annotate the functions they want to optimize but there's little to no user friendly feedback on what's been optimized and how they can exploit more parallelism or vectorization.

In this paper, we present a Python library named NumbaSummarizer that streamlines Numba's vectorization analysis. It is designed as a pedagogical tool for teaching Dependence Analysis, Loop Transformations, and Vectorization as practical examples of parallel computing. We used this tool along with a prepared lesson plan to introduce concepts of parallelism and high performance computing to students from an intermediate programming class.

The rest of the paper is organized as follows: **Related Work** is a summary of other tools and teaching models that

are related and have inspired the work in this paper. **Design Overview** describes NumbaSummarizer, its components and functionality. **Experience** outlines our teaching experience using the tool and the materials used along with NumbaSummarizer. **Discussion** presents a brief analysis in retrospective of the advantages and disadvantages of using this tool, and proposed improvements for future experiences. **Conclusion** gives a brief overview of why we believe this tool is useful as well as propose different scenarios where it could be adopted.

II. Related Work

Introducing concepts of high performance and distributed computing outside specialized or high level courses is not new. As noted by Prasad et al. [15] and the ACM/IEEE-CS curricula 2013 [7], it is more effective to learn about parallelism "sprinkled" throughout the curricula. Other works have tried to put this idea into practice. The work of Garrity et al. [6] involves a multi-lingual platform for simple and straightforward implementation of MapReduce [4], which is a powerful model for distributed computing. This is a high-level application of parallelism, and the authors report success in using it for teaching parallelism in introductory and advanced courses alike. Similarly, Ortiz-Ubarri and Arce-Nazario [13] describe an interface for MapReduce through Python modules.

The work of Curtsinger [3] teaches students of an Operating Systems class how to use concurrency and multi threading as parallelism in practice. In this case the students are experienced with C and have previous knowledge on Computer Architecture. Our work focuses on first year students and just assumes basic programming and algorithmic knowledge. Our module is introduced as part of programming optimizations which include simple algorithmic analysis and finding run time hotspots.

Our focus is on using SIMD and dependence analysis as the main catalyst to learning parallelism. Ortiz [12] teaches the SIMD execution model in an introductory computer architecture course. However this is done at the assembly level, focusing on getting the student familiarized with SIMD idioms.

This paper presents the continuation of the work in Watkinson et al. [18]. Students were given loops written in C++. Then they transformed the loops and used Intel's C compiler (ICC) vectorization report to identify if vectorization was successful. On a following experience [19], we used Python as our language and tested students of Intermediate Programming. We tried to use native Python libraries so we relied on the Multiprocessing module to exploit parallelism. The main drawback, however, is that it requires manual workload distribution, which is an error prone task. While students were able to absorb theoretical knowledge in parallel computing, their practical implementation was moderately successful. We use a very similar theoretical teaching for this work, but in order to help students focus on exposing parallelism, we opted for using Numba's automatic vectorization. NumbaSummarizer enables Numba's automatic vectorization, and vectorization reports while keeping a simple interface. We also refactored the loops to make them representative of the loop transformations that we taught.

III. Design Overview

Our contribution is two fold:

1) NumbaSummarizer, a wrapper that works on top of Numba [1] module that users can import into their code so they can analyze functions for vectorization and the presence of data dependence.
2) Lecture notes and sample problems for implementing NumbaSummarizer in a classroom. [2]

NumbaSummarizer eases the implementation of Numba's vectorizing functionality by encapsulating all required Numba's specific syntax into custom functions. NumbaSummarizer will instruct Numba to request a vectorization report from LLVM. Once that report is generated, NumbaSummarizer will parse through it and extract information pertaining the targeted functions. Figure 1 shows an abstract overview of how the three of them interact. Numba translates Python functions into an intermediate language (IR) that LLVM can interpret and pre-compile. For this reason, many details about the original function, such as the function name, gets lost in the report. Also, the report contains plenty of information relevant for the compiler but not necessarily for the Python user. Because of this, we need to filter the information and generate a new summarized report that focuses only on the vectorization results for the given function, adding our own identifiers as well. Since our annotated functions have a simple structure with one or two loops, we show students how to map the LLVM report to them. NumbaSummarizer also has a correctness checker which analyzes the output of the original loop and the one from the loop transformed by the programmer to verify that they are the same. This behavior is accessible through the following NumbaSummarizer's functions:

A. NumbaSummarizer's functions

- **Init_diagnostics()** enables the LLVM flag *debug-only = loop-vectorize* that allows for the creation of vectorization reports. As of now, this function is private and called by ***Vector_wrapper()*** when needed.
- **Vector_wrapper(Original_Function)** takes in the function to be analyzed (either through a decorator or by calling the function directly), executes it twice: First with the environment's Python interpreter, and then pre-compiled with Numba's JIT compiler; and prints out the comparison of the execution times. Afterwards, it generates the vectorization report through ***Init_diagnostics()*** by running the pre-compiled function again. Finally, it returns a decorated function.
- **Vector_print(Decorated_function)** requires the report and decorated function from ***Vector_wrapper()***. This function reads LLVM's vectorization report and prints

[1]NumbaSummarizer is available through PyPi by executing the command: $pip install NumbaSummarizer

[2]The source code and sample lecture notes are available from the GitHub repository: https://github.com/neftaliw/NumbaSummarizer/

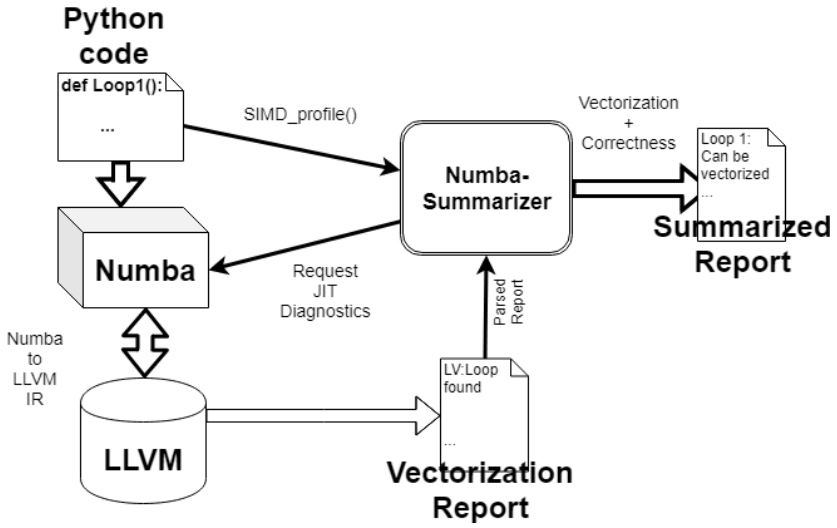

Fig. 1. Graphical representation of how each component interacts with NumbaSummarizer to produce the final output. The thick arrows represent internal processes and communication between Numba and LLVM, and NumbaSummarizer with the report.

out the information that is relevant for the student. This includes the loop identifier for every loop present in the function, whether that loop can be vectorized, and if legality cannot be proven for the loop (which means that there's data dependence).

- **SIMD_profile(Original_Function)** streamlines the vectorization report workflow. While each function can be called separately, this function will implement them in the right order.
- **Compare_loops(Original_Output, Solution_Output)** is not part of the vectorization analysis workflow. It is used to verify the output for manually vectorized functions. The analysis works better if the outputs are formatted as a list. The function prints out a message stating whether the outputs are identical or not. If they are not, it will raise an error stating that the outputs are not the same, which usually means an improper transformations was used or not implemented correctly.

B. Requirements

The required hardware and python libraries for using NumbaSummarizer are similar to the requirements for Numba:

1) Hardware Requirements:

- Operating systems and CPU:
  - Linux: x86 (32-bit), x86_64, ppc64le (POWER8 and 9), ARMv7 (32-bit), ARMv8 (64-bit)
  - Windows: x86, x86_64
  - macOS: x86_64
- CPU with SIMD instructions
- Python $\geq$ 3.5

2) Python Dependencies: These can be obtained through PyPi (i.e. pip install):

- Numba $\geq$ 0.45
- Llvmlite $\geq$ 0.28
- Numpy $\geq$ 1.9

NumbaSummarizer also uses the following Python built-in[3] libraries: OS, Time, Random, IO, Functools, and ContextLib.

C. Summarized Report

As of now the report is printed out entirely on the user's terminal. We find this to be simpler for the student, though the code can be easily modified to generate a log file instead. As per usage, there are two types of report that NumbaSummarizer can generate:

1) Correctness report: Listing 1 shows a sample of how the correctness report looks like. First line displays the name of the function that is being tested. For this specific case there are three objects as part of the function's output. The first two objects are identified as being Numpy arrays but the third object has an unsupported type, this is because the function Compare_loops() only supports Numpy arrays, float, and integer values. It is also reporting that for object 3 the objects are not equal between the original function and the transformed one.

2) Vectorization Report: The vectorization report is generated per function. That means that if two functions are being compared (original and transformed), the user has to generate a report for each one separately. As seen in Listing 4, the report starts with the name of the function, followed by a comparison of running times with optimizations and without them. This is not a comparison between a manually transformed function and the original, but between a JIT compiled function and the one executed using the default Python distribution. This is immediately followed by a breakdown per loop found in the function, and whether it can be vectorized or not. The message *Cannot prove legality* points towards the presence of data dependence that is impeding vectorization. It is interesting

[3]Python's built-in functions, libraries and objects are already included in the standard target distribution and therefore are not generally required to be installed separately.

Listing 1. Correctness report generated by the function **Compare_loops()**

```
Testing Function_Loop1
*****************************
Checking element 1:
Element 1 is a Numpy Array
Check passed
*****************************
Checking element 2:
Element 2 is a Numpy Array
Check passed
*****************************
Checking element 3:
Check failed, both items are not equal
Element 3 is not a type that is supported
---------------------------------------
```

Listing 2. Example on how to use Numba's native interface to achieve a vectorization report.

```
from numba import jit
import llvmlite.binding as llvm
llvm.set_option('',
    '--debug-only=loop-vectorize')
@jit
def loop(A,B):
    ...
```

to note that each loop has an identifier. These are generated during the translation to the IR between Numba and LLVM and we have no control over it nor sure way to map it to the Python code. After generating the report, the returned function is already pre-compiled by Numba's JIT and can be reused in the rest of the code for further performance testing.

D. Comparison with Numba

While Numba itself is simple enough to be used by intermediate level programmers, there are some advantages that NumbaSummarizer has over it for learning purposes. For example:

- Numba's @JIT decorator will optimize and parallelize functions when possible. However, there's little feedback on whether these optimizations were successful. Numba is not meant to be used to debug vectorization.
- There's a workaround to generate a vectorization report, which requires setting LLVM to debug mode.
- The vectorization report generated by LLVM through Numba is hard to read for anyone not familiarized with the IR. NumbaSummarizer simplifies the report significantly and makes it user friendly.

To further contrast the difference, we include Listing 2 to show how it would work using Numba's interface. This code requires for the LLVM compiler to be set to debug mode. On the other hand, Listing 3 exemplifies how our wrapper simplifies the implementation. It is important to note that our library will set the compiler to debug mode and sets it back to

Listing 3. Example on how NumbaSummarizer simplifies the interface

```
from NumbaSummarizer import Simd_profile
@Simd_profile
def loop(A,B):
    ...
```

Listing 4. Vectorization report generated by the function **Simd_profile()**

```
Function_Loop1
        -without optimizations took
         0.257 seconds to run
        -with optimizations took
         0.007 seconds to run
It is 36.196 times faster
with optimizations

- Found a loop: B48.us
- Not vectorizing:
    Cannot prove legality.
- Found a loop: B58.us
- We can vectorize this loop
```

compiling mode automatically whereas with the code shown from Numba, the programmer would need to do it manually. An additional difference is that NumbaSummarizer generates a report file with the contents of the full report, which then parses to the command line. Numba's LLVM report can only be displayed on the command line.

As part of our lesson plan, we teach the student how to use Numba's @JIT decorator. We clarify that our tool is meant for educational purposes, but for production code Numba should be the tool of choice.

IV. Experience

We used NumbaSummarizer in a class session of UCI's ICS33: Intermediate Programming with 64 students. This is the third quarter of Python programming that undergrads go through. We had 64 students, most of them in their first year of studies with approximately 15% of them being in their second year. Only two students had previous experience with C and with Introduction to Computer Architecture, which deals with assembly programming. Over the course of one week, we taught two lectures where we go over concepts of high performance computing, JIT compilers, Data Dependence, Loop Transformations and usage of both Numba and NumbaSummarizer. Afterwards, students had a take home assignment where they need to analyze and try to achieve vectorization for 8 functions. The main body of each function is a loop that does arithmetic computations mainly involving Numpy arrays. The students have to figure out how to transform the loops to expose parallelism. They rely on NumbaSummarizer's report for their tasks.

The structure of the lectures is very similar to [19]. We used the same material to introduce the concepts and the

Listing 5. Sample of loops used in assignment. Some syntax has been omitted for clarity

```
Loop1:
    for i in range (n-1):
        A[i+1] = B[i]+C[i]
        B[i]   = C[i]*E[i]
        D[i]   = A[i]*E[i]
    return A,B,D
Loop3:
    for i in range (n-1):
        A[i] = B[i] + C[i] * C[i]
             + B[i]*b[i] + C[i]
        D[i] = A[i] + A[i+1]
    return A,D
Loop6:
    for i in range (nn):
        for j in range (1,nn):
            AA[j][i] = AA[j - 1][i]
                     + BB[j][i]
    return AA
Loop8:
    for i in range (n-1):
        vsum += A[i]
        B[i] = vsum
    return vsum, B
```

same quizzes. Students had comparable success with written quizzes, in this paper we will not focus on that. We teach them how to identify data dependence and we show them the transformations that will be relevant to their assignment which include: loop invariant code motion, loop distribution, loop interchange, loop peeling, loop reversal, and, using temporary arrays to relax dependence. All the information pertaining data dependence is sourced from the work of Padua and Wolfe [14] and Kennedy and Allen [8].

Listing 5 shows 4 out of the 8 loops given to students. Each one was chosen as representative of the type of loops that benefit from each transformations that we taught. While students are not required to report on their dependence analysis, they have to perform it in order to apply transformations properly. It is important to note that NumbaSummarizer is completely agnostic as to which transformation was applied, it can only compare the outputs of the original and the transformed loop to check for correctness.

Besides the vectorization report, the students had feedback from an auto-grader hosted on Gradescope [17] that would let them know (through the use of unit tests) if their amount of vectorization (full, partial or none) was enough for every given loop. Listing 6 shows the optimal solution for each loop. Loop 6 is a very peculiar case. When testing the loops on their own machines, students found out that the solution would vectorize for some distributions of LLVM and not for others. This is the only loop that had such inconsistency. Also, Loop 6 (shown here) and Loop 7 are the only ones that deal with

Listing 6. Sampled loops after manual transformations

```
Loop1 Solution:
## Loop Distribution
    for i in range (n-1):
        A[i+1] = B[i]+C[i]
        B[i]   = C[i]*E[i]
    for i in range (n-1):
        D[i]   = A[i]*E[i]
    return A,B,D
Loop3 Solution:
## Temporary array
    temp=A
    for i in range (n-1):
        A[i] = B[i] + C[i] * C[i]
             + B[i]*b[i] + C[i]
        D[i] = A[i] + temp[i+1]
    return A,D
Loop6 Solution:
## Loop interchange
    for j in range (1,nn):
        for i in range (nn):
            AA[j][i] = AA[j - 1][i]
                     + BB[j][i]
    return AA
Loop8 Solution:
## Can't be vectorized
    for i in range (n-1):
        vsum += A[i]
        B[i] = vsum
    return vsum, B
```

two-dimensional arrays. While generally a RAW dependence would not allow for vectorization, interchange will relax the dependence enough so that the compiler can exploit it.

The students had 5 days to work on their code and two lab sessions for troubleshooting. Besides some technical questions, none of them expressed any concerns or difficulties in understanding the task at hand. Table I shows the grade distribution for all 8 loops. Considering Loop 6's issue, we still considered it correct if students reported it as not vectorizable. 64% of the students applied interchange to it. Loop 5 has a conditional statement that determines the presence of a WAR dependence or a RAW dependence. Hence being represented as 1.5 and 0.5 respectively.

V. Discussion

A week after the assignment, the students had a midterm. We added two sections related to vectorization and dependence analysis. First, they were given five loops (Listing 7, and they had to identify which types of dependence was present. They also had 8 questions on concepts of high performance computing, loop transformations, and JIT compilers. Following are the major takeaways:

TABLE I
DESCRIPTION AND RESULTS FOR EACH LOOP IN THE ASSIGNMENT

Name of Loop	Dependence present	Transformation needed	Student Success Percentage
Loop1	WAR(x2),RAR(x2)	Distribution	96%
Loop2	WAR(x2)	Distribution	95%
Loop3	WAR, RAW	Temporary Array	89%
Loop4	RAW(x2), RAR	Distribution (Partial)	97%
Loop5	WAR(x0.5),RAW(x1.5) RAR	Distribution, Peeling (Partial)	96%
Loop6	RAW	Interchange	100% (64%)
Loop7	RAW(x2)	Distribution, Interchange	94%
Loop8	RAW	NA	100%

Listing 7. Loops used in the midterm

```
Loop1:
    for i in range (len(A)):
        A[i]=A[i]
Loop2:
    for i in range (1,len(A)-1):
        A[i]=A[i-1]
        A[i+1]=A[i]
Loop3:
    for i in range (1,len(A)):
        A[i]=B[i-1]
        A[i-1]=B[i+1]
Loop4:
    for i in range (len(A)):
        A[i]=B[i]
        B[i]=A[i]+B[i-1]
Loop5:
    for i in range (len(A),1, -1):
        A[i]=B[i]+A[i+1]
```

- 70% of the students correctly analyzed the loop with only WAR dependence
- 60% got the loop with WAW dependence correct.
- 40% were able to identify the two loops with RAW dependence
- 80% identified the loop that had both WAR and RAW dependence
- 95% of the students answered questions about concepts of parallel computing and loop transformations, with the only exception being loop distribution, where only 50% of the students answered correctly. Most common mistake was confusing it with loop peeling.

The most common mistake in identifying dependence was that students reported more than the expected dependence. One of the loops had a RAW dependence but because it iterated in inverse order, students misidentified it as a WAR dependence. However, they were able to answer most of the conceptual questions correctly.

We believe our experience was an improvement over [19] for the following reasons:

- Our assignment is mandatory. All minus one student turned in their code.
- Students only need to focus on transforming loops and identifying data dependence without worrying about workload distribution. The code for testing the loops is already given to them thanks to NumbaSummarizer's functions. This makes the experience less frustrating since the students have little to no experience with compilers and computer architecture.
- Our students had two sources of feedback. The report generated from NumbaSummarizer to identify if there were any data dependence, and the one from Gradescope's autograder to know if their resulting optimization was similar to what was expected by the instructor.

It is important to emphasize that NumbaSummarizer is meant for early exposure to parallel computing. Providing students with a simple interface allows for presenting the simple task of optimizing code without having to delve deep into details about compilers.

A. Future work

We plan to expand NumbaSummarizer to include dependence analysis as well, that will let students know which data dependence is present. This is not a trivial task since it requires a source code analyzer or run time dependence checker. There are some experimental tools that can do it for other languages like C, but for Python we would need to design our own.

Our experience could be improved if the students are provided with a common server where they can run their code. This should mitigate some of the irregularities over different distributions of Python and LLVM.

Finally, we'd like to test NumbaSummarizer within a Parallel Computing course. This has the potential of a richer experience, and it could set the basis for teaching multi-lingual parallel computing that will allow students who prefer Python to use it for the course, instead of C which currently is prerequisite at UCI. Eventually we want NumbaSummarizer to be useful in production code, so programmers can use it to further improve code optimized by Numba.

VI. CONCLUSION

NumbaSummarizer is a Python library that functions as a wrapper for Numba JIT compiler. It generates vectorization and correctness reports that can be used to teach parallel computing with a focus on vectorization. Along with the provided set of lectures and teaching material, students are able to perform dependence analysis and loop transformations. In a reported experience, students had to analyze 8 loops and

transform them to obtain performance improvement through vectorization by using NumbaSummarizer's report as their main guide. The students successfully found opportunities for vectorization with an average success of 95% of top performance. They were also tested on concepts of high performance computing and dependence analysis and obtained an 80% grade average. The academic experience is planned for two lectures followed by a 5 day assignment. The purpose is to make it easy to adopt in any programming course.

References

[1] Nabeel Alzahrani, Frank Vahid, Alex Edgcomb, Kevin Nguyen, and Roman Lysecky. Python versus c++: An analysis of student struggle on small coding exercises in introductory programming courses. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 86–91, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5103-4. doi: 10.1145/3159450.3160586.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb 1993. ISSN 0018-9219. doi: 10.1109/5.214548.

[3] Charlie Curtsinger. Parallelism in practice: experiences teaching concurrency and parallelism in an undergraduate os course. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pages 1–6, 2019.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Vladimiras Dolgopolovas, Valentina Dagienė, Saulius Minkevičius, and Leonidas Sakalauskas. Python for scientific computing education: Modeling of queueing systems. *Scientific Programming*, 22(1):37–51, 2014.

[6] Patrick Garrity, Timothy Yates, Richard Brown, and Elizabeth Shoop. Webmapreduce: An accessible and adaptable tool for teaching map-reduce computing. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 183–188, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0500-6. doi: 10.1145/1953163.1953221.

[7] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA, 2013. ISBN 9781450323093.

[8] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[9] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, page 7. ACM, 2015.

[10] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[11] Ami Marowka. On parallel software engineering education using python. *Education and Information Technologies*, 23(1):357–372, 2018.

[12] Ariel Ortiz. Teaching the simd execution model:: assembling a few parallel programming skills. *ACM SIGCSE Bulletin*, 35(1):74–78, 2003.

[13] José Ortiz-Ubarri and Rafael Arce-Nazario. Modules to teach parallel computing using python and the littlefe cluster. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013.

[14] D. A. Padua and M. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12): 1184–1201, 1986.

[15] Sushil K Prasad, Almadena Yu Chtchelkanova, Sajal K Das, Frank Dehne, Mohamed G Gouda, Anshul Gupta, Joseph Jaja, Krishna Kant, Anita La Salle, Richard LeBlanc, et al. Nsf/ieee-tcpp curriculum initiative on parallel and distributed computing: core topics for undergraduates. In *SIGCSE*, volume 11, pages 617–618, 2011.

[16] Brian Rague. Teaching parallel thinking to the next generation of programmers. *Journal of Education, Informatics and Cybernetics*, 1(1):43–48, 2009.

[17] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. Gradescope: a fast, flexible, and fair system for scalable assessment of handwritten work. In *Proceedings of the fourth (2017) acm conference on learning@ scale*, pages 81–88. ACM, 2017.

[18] Neftali Watkinson, Aniket Shivam, Zhi Chen, Alexander Veidenbaum, and Alexandru Nicolau. Using data dependence analysis and loop transformations to teach vectorization. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1143–1148. IEEE, 2017.

[19] Neftali Watkinson, Aniket Shivam, Alexandru Nicolau, and Alexander Veidenbaum. Teaching parallel computing and dependence analysis with python. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 320–325. IEEE, 2019.

[20] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, Apr 1987. ISSN 1573-7640. doi: 10.1007/BF01379099.

[21] Giancarlo Zaccone. *Python Parallel Programming Cookbook*. Packt Publishing Ltd, 2015.