Algorithm Engineering Assignments

Aron Gaden

November 2021

Contents

1 Lecture 1

1.1 Describe how parallelism differs from concurrency

The concurrent execution of a program describes that more than one action can be in progress at the same time. This does not mean that more than one action has to be active at any given moment.
Parallel execution, on the other hand, implies that more than one action is active. That is, executing at least two tasks at the same time. (wording) Parallelism is therefore a proper subset of concurrency.

1.2 Fork-Join Parallelism

Fork-Join-parallelism is a method of parallel execution employed in the OpenMP framework in which a main thread spawns a team of threads (fork), that are executed in parallel. After the parallel section, the threads apart from main thread are terminated (join).

1.3 Computer Systems - Discussion of virtual memory

interesting: each process has own virtual memory with 2ˆ(architecture)32 entries -¿ even though physical memory may be much smaller. Solves other problems such as security in sharing physical spaces in sections (define sections in memory). (each frame/tile of memory maps to physical memory ONLY if necessary). Very elegant, efficient, and powerful solution at once.

1.4 Performance Gains after Moore's Law ends

The figure depicted in the journal article highlights prospective avenues of increasing computer performance that are not related to adding more transistors onto chips ("the bottom"), as development is closing in on physical limits. These avenues are split into three technologies: *Software*, *Algorithms*, and *Hardware architecture*.

1.4.1 Software

The first optimizable process in computer performance is what the authors call "software performance engineering". More precisely, they talk about the restructuring of software such that inefficiencies are removed. The authors argue that the software engineering process is often times optimized for short development time, as opposed to short runtime, leaving room for improvement. In practice, efficiently (at times even in other programming languages leading to faster execution) implemented libraries are used to avoid having to rewrite common routines, perhaps in a suboptimal manner.
Another aspect of software performance engineering is tailoring software development to the hardware conditions, stating the usage of vector units or parallelization as an example. For instance, when performing element-wise opera-

tions on matrices, vectorization is highly applicable (through the sheer amount of independent operations) and saves a lot of time.

1.4.2 Algorithms

The second area with room for improvement is named as "Algorithms" by the authors. An example is given through the max-flow problem, where algorithmic advances led to a large part of the runtime decrease, as opposed to only hardware improvements. Improvements in this domain suffer from diminishing returns by nature, and are therefore most significant in newer fields. For example, the application of convolutions for deep neural networks allowed the handling of larger inputs such as images, while keeping the number of parameters manageable.

1.4.3 Hardware Architecture

The authors talk about the pending simplification of processor design as an avenue of further improving performance from the side of hardware architecture. That is, optimizing circuits to use the freed up transistors and on-chip space freely. Another, arguably more important, aspect is the domain-specialization of hardware. The most prominent example of this are GPUs, which can process graphics computations extremely fast compared to a traditional CPU.

2 Lecture 2

2.1 What causes false sharing?

False sharing occurs, when multiple processes access the same cache line, *and* at least one of them writes onto the cache line. This leads to a cache invalidation, where all other processes need to refresh the cache, possibly leading to a drastic decline in performance.

2.2 How do mutual exclusion constructs prevent race conditions?

Mutual exclusion constructs (Mutex) allow locking critical sections in which race conditions can occur, if exactly one . This forces all other processes to wait outside the critical section, until it is unlocked again. Note that a lock is a critical section too, so a suitable implementation such as an atomic lock is necessary.

2.3 Differences between static and dynamic schedules in OpenMP.

Static schedule: In this case, scheduling decisions are made at compile time, which means that the program execution order is predetermined.
Dynamic schedule: The dynamic schedule does not determine workload between threads until runtime. While this obviously leads to higher execution overhead, unbalanced workloads can be broken up better between threads, improving performance in such situations.

2.4 What can we do if we've found a solution while running a parallel for loop in OpenMP, but still have many iterations left?

There is no singular way to prematurely stop a for loop when using OMP. A suggested approach is to introduce a boolean that indicates whether a solution was found. If so, then the iteration can be skipped, minimizing the workload and subsequent working time inside the loop. TODO: Whats up with that other code solution? It breaks the loop but can slow it down

2.5 About std::atomic::compare_exchange_weak

The coding warmup link is at 2.5
The function atomically compares the calling object with an expected one. If the bitwise comparison is successful (i.e. *this is bitwise-equal to expected), it is replaced by the desired object. It is possible that the function returns a wrong result, where the comparison will be unsuccessful even if the objects are equal.

As a tradeoff, this version of the function is possible more performant in loops (platform-dependent).

2.6 Coding Warmup

For point 3 (Which schedule causes the unbalanced pattern): The static scheduling. The iterations are likely of unbalanced length The programs are at `https://github.com/oranium/AlgoEng/tree/master/answers/aron_gaden/lec2`

3 Lecture 3

3.1 The *ordered* clause in OpenMP

In an ordered region, all the threads will execute the code sequentially, that is, one thread enters the region after another has finished. This is equivalent to a loop in single threaded execution and avoids race conditions. When used in a parallel loop, the threads have to wait for their respective turn at each iteration. There is only one instance of the ordered region.

3.2 The *collapse* clause in OpenMP

The collapse clause allows for parallelized nested loops. Inner loops would not be executed in parallel, so reducing a loop with m steps inside a loop with n steps into one loop with $n*m$ steps allows for more efficient parallelization while keeping code readability high.

3.3 Reductions in OpenMP

The list to be reduced is copied locally to perform updates on. The initialization of the list is depending on reduction operator, as the neutral element is used. The local copy of the list is always updated, until a single values remains, which is then stored in combination with the original list.

3.4 Barriers in parallel computing

The purpose of barriers is the synchronization of threads. Execution can only continue after all threads reached the barrier. An example where this is useful is the merge sort algorithm as presented in lecture 4, where the merging can only commence after all lists have been split - otherwise the sorting might be wrong. (?TODO)

3.5 Distinction between selected OMP methods

omp_get_num_procs(): The current number of threads in the parallel region
omp_get_num_threads(): The number of logical cores (upper limit for $max\_threads$)
omp_get_max_threads()the maximum number of threads in a parallel region (can be lower or equal to $max\_threads$) $max_threads$ is a variable that can be set to limit the amount of threads that are spawned by the program.

3.6 Storage attributes: *private* vs *firstprivate*

Private: variables are uninitialized and each thread gets a local, unshared copy
Firstprivate: variables are initialized, each thread gets a local, unshared copy

3.7 Coding warmup and pseudocode

The link to the warmup code is `https://github.com/oranium/AlgoEng/tree/master/answers/aron_gaden/lec3`
My Pi parallelization code differed a bit from the reference implementation and likely performs worse, but the pseudocode is based on the former anyway.

Algorithm 1 Pi parallelization with simple threads

procedure PI_THREAD_HELPER($step, width, iterations, sum$)
    $x \leftarrow 0.0$
    $local\_sum \leftarrow 0.0$
    for $i : iterations$ **do**
        $x \leftarrow (steps * iterations + iteration + 0.5) * width$
        $local\_sum \leftarrow local\_sum$
    end for
    CRITICAL do ▷ some critical or atomic implementation
    $sum \leftarrow sum + local\_sum$
    end CRITICAL
end procedure
procedure PI_ITERATIVE(num_steps, num_threads)
    $sum \leftarrow 0.0$ iterations $\leftarrow$ num_steps / num_threads$width \leftarrow 1/num_steps$
    for $i : num\_threads$ **do**
        $create\_thread(pi\_step, i, width, iterations, sum)$
    end for
    join_all_subthreads()
    pi $\leftarrow sum * width * 4$
    return pi
end procedure

4 Lecture 4

4.1 Explanation of divide and conquer parallelization using OpenMP tasks

A task is defined as an independent unit of work. Therefore, tasks can be created (i.e. recursively), where each task can be assigned a thread for execution. As divide and conquer algorithms generally lend themselves to recursive execution, they work well with tasks in OpenMP. Each recursive subcall can then be assigned their own thread, improving execution time.

4.2 How to speed up merge sort

The following ways of speeding up merge sort were discussed:
By using tasks, merge sort's performance can be improved with OpenMP. As explained in the previous subsection, each task can get executed independently and in parallel, improving performance. There should be a lower bound (10,000 in the lecture) for the creation of a new task, as the cost of task creation may outweigh the benefits of parallelization. Similar goes for the recursive calls in merge sort, where another non-recursive algorithm might be preferable for small arrays (lower bound 32 in the lecture).
Merge sort can be further optimized by allocating the memory for the entire array upfront and then inserting the respective subarrays in their location according to the algorithm. This saves time that would otherwise be lost copying the subarrays.

4.3 Multithreaded Merging

For two arrays, the median of the larger one will be chosen as an immediate cutoff. This is just a lookup, where each of the remaining array parts is then to be merged with the corresponding parts of the smaller array. In the smaller array, a binary search can be conducted to find elements that are larger or smaller than the median of the large array. Here, the search as well as the merges can then be executed independently and therefore parallelized.

4.4 Coding Warmup Slide 17

Solutions are at
`https://github.com/oranium/AlgoEng/tree/master/answers/aron_gaden/lec4`
Interestingly, I could not run the parallel merge sort on my Mac, crashing with a segmentation fault after many parallel task creations. I do not know if it has to do with AppleClang, as I did not test it on GCC for the Mac.

4.5 Paper: What every systems programmer should know about concurrency

Atomicity It is mentioned that reads and write should be atomic when concurrency is implemented for a system. This has interesting implications on variable size. If a variable is longer than the CPU's word size, multiple load operations are required for a read operation. This can lead to torn reads (or writes). Therefore, a variable should be constrained to the CPU's word length, where possible, otherwise errors or performance reductions may occur.

Atomic Operations as Building Blocks The discussion marks atomic loads, stores and read-modify-write operations as building blocks for concurrent systems. Two different perspectives are offered towards atomic operations: *blocking* and *lockless* synchronization methods. The former are easier to handle, but can lead to dead- and livelocks fast, rendering the program useless in the best case, if not damaging. Mutex constructs can avoid this, but obviously hinder performance through blocking variables.
The latter, lockless synchronization, guarantees freedom from deadlocks and livelocks. Lockless methods are worthwhile for example in streaming operations, which should never be blocked. Threads cannot indefinitely be stopped using this paradigm. Both approaches are necessary, the authors emphasize, as they are different tools for different goals.

5 Lecture 5

5.1 What is CMake?

CMake is, in essence, a build file generator. This makes it useful for creating programs that are compilable independently of the platform or compiler. It can create recipes for building (cmake) entire projects, as well as testing (ctest) and packaging (cpack). Compile flags can be set and required depending on the configuration of the platform and compiler at hand. Technically, CMake has the same functionality as other scripting languages like bash, but the aforementioned usages are the intended and most common ones.

5.2 Targets in CMake

In CMake, targets are the desired output of the build recipe. They can be either executables or libraries. There can be multiple targets for a CMake file, as they can be constructed in an object-like fashion. This allows for alteration of the targets. Then, compile options can be added for each target, as well as the necessary source files, include directories, etc.

5.3 Approach to code optimization

> Premature optimization is the root of all evil.
>
> Donald E. Knuth

Personally, I have little experience in both C++ and explicit code optimization. Therefore, I approach these problems by implementing a solution to my problem in another programming language, using prebuilt functions if applicable. If the problem is sufficiently handled by my program (and not performant enough!), I will transcribe it to C++, and then identifying bottlenecks in the code. If these occur in an existing code snippet, it is necessary to manually implement it and then optimize the code until it meets the performance requirements or the current bottleneck is completely optimized. This could be achieved by either vectorizing the code, in the best case according to the underlying hardware architecture, if there won't be multiple. Parallelization of independent regions, for example with OpenMP or intrinsics, is another avenue of performance optimization. The process should, if necessary, be repeated for every bottleneck in the code. The bottlenecks should be evaluated and ordered by importance, as to get the most out of optimizations.

6 Lecture 6

6.1 Characteristics of selected instruction sets

All three of these instruction sets are SIMD (Single Instruction Multiple Data) vector instructions sets. That means, an instruction will be executed (element-wise) for a whole vector, instead of per element, allowing for independent execution.

6.1.1 SSE

SSE is the oldest of these instruction sets, first launching in 1999. The vector length is 128 Bit in the 16 xmm0-xmm15 vector registers, therefore allowing for vectorized operations on, allowing at once. It is primarily designed for floating point operations. Operations are executed with two operands and of the form $a := a + b$.

6.1.2 AVX2

AVX2 is currently the most widespread instruction set for Intel CPUs. It works with a vector length of 256 Bits, where the 16 vector registers are called ymm0-ymm15. It can process SSE instructions. The lower 128 bits of the ymm registers are used as xmm registers. Operations in the ymm registers can be of the form $c := a + b$, adding a third operator, avoiding copies because the source register is not overwritten. This also applies to AVX-512. They are also

6.1.3 AVX-512

AVX-512 is the most modern instruction set of the three, working with 512-Bit vectors in the zmm0-zmm31 registers, therefore having a total of 32 vector registers. It is supported on newer and more powerful devices or HPC-CPUs. It also supports SSE instructions, as well as AVX(2) instructions. Compared to AVX2, some commands were/will be added.

6.2 Effect of memory aliasing on performance

Memory aliasing describes a situation where two or more pointers (potentially) refer to the same location in memory. This can lead to code that cannot be vectorized - leading to performance decrease. Specifically, this happens when unsafe (i.e. writing) operations are executed through either of the pointers. If it is known that no pointer aliasing will occur, using the *__restrict__, compilers may apply optimizations leading to increased performance.

6.3 Advantages of unit stride memory access

Unit stride memory access refers to the sequential (or parallel) memory access of data in memory that is exactly one unit apart (as opposed to n units). This

improves bandwidth utilization and makes the code vectorizable for compilers, that can load the array into a vector register.

6.4 Favorable situations for using Structure of Arrays

A Structure of Arrays (SoA) refers to an approach of arranging record sequences in memory. Specifically, there will be a C-struct, where fields are given in the form of arrays. For example, there might be a collection of x,y,z coordinates in the struct *s*. *s* will then have the three array fields x,y and z, where the elements are respective coordinate points. The first x,y,z triple will be contained of (s.x[0],s.y[0],s.z[0]). In this situation, SoA are obviously not favorable, because three array accesses have to be made. However, if statements or calculations have to be made about individual fields, SoA becomes favorable. In other words, SoA is best applied when a data set has attributes where statements have to be made about the same field across different data points.

7 Lecture 7

7.1 Three vectorization clauses used with *#pragma omp simd*

7.1.1 safelen

The clause *safelen(len)* tells the compiler to restrict the maximum vector length (= maximum distance in the logical iteration space), up to which code can be vectorized. The length of a statement must be constant.

7.1.2 collapse

The *collapse(n)* clause can be used to vectorize nested loops with n levels. In that case, nested loops will be reduced to one loop by the compiler, which makes vectorization possible where it would otherwise not be possible.

7.1.3 aligned

aligned(list[:alignment] signals to the compiler, that the *list* argument is aligned bytewise with the given *alignment*.

7.2 Pros and cons of vectorization with intrinsics vs. guided vectorization with OpenMP

When using intrinsics, the developer takes responsibility for vectorization of code. This has the advantage of having more freedom in the development process in cases where OpenMP would just ignore the developer (as often is the case with the *simdlen* clause), but is naturally more error-prone because it is more complicated. The biggest advantage of intrinsics is the ensuing performance portability. Intrinsics will yield similar execution speeds on different compilers, whereas the results of guided vectorization can vary wildly from compiler to compiler. However, intrinsics are always tailored to a specific processor architecture. This leads to a major disadvantage of intrsics: reduced code portability between different architectures, which is a clear upside of guided vectorization as opposed to using intrinsics.

7.3 Advantages of vector intrinsics over assembly code

Vector intrinsics share similar advantages over guided vectorization as writing assembly code right away, as discussed in the previous task. However, assembly code is generally even more complicated than the usage of vector intrinsics, as the latter are nothing but a wrapper for assembly. Registers do not have to be assigned explicitly. Because they are valid C++ code, they are also more portable, as different compilers therefore operating systems can handle the code.

7.4 Vectors corresponding to selected data types

__m256 refers to a vector of 256 bit length, containing eight floating point numbers.
__m256d refers to a vector with 256 bit length, containing four double precision numbers.
__m256i also refers to a vector with 256 bit length. The vector will contain integer values: either 32 8-bit, 16 16-bit, 8 32-bit or 4 64-bit values.

8 Lecture 8

8.1 Naming conventions for intrinsic functions

Intrinsic functions will always start with an underscore (_) to differentiate them from data types starting with two underscores. they are followed by the vector size in bits, another underscore, then the instruction, another underscore and a suffix, being the data type of the input vectors. For the latter, *ps* is used for floats, *pd* for doubles and *ep*⟩ *int_type* ⟨ signifies integers, which can themselves be signed (i), unsigned (u), and have different sizes. The whole naming scheme for intrinsic functions is the following: $\_\langle vector\_size\rangle\_\langle operation\rangle\_\langle suffix\rangle$

8.2 Latency and throughput as indicators of performance in intrinsic functions

The throughput of an intrinsic function describes the number of clock cycles until a new instruction of the same kind can be started. Latency describes the number of cycles until the result is available for use. High latencies of an intrinsic function can be mitigated by software pipelining, where the instruction can be repeatedly started, so results will be continuously available after one latency period. This method assumes a high(er) throughput for the function.

8.3 Instruction-level parallelism (ILP) in modern processors

ILP or superscalar execution refers to the possibility of starting or evaluating multiple instructions in the same clock cycle. This necessitates different functional units in the execution engine of a core, which is realized through adding multiple ports to the scheduler, one per functional unit. Each of those ports can be used for a set of micro-operations, such as branching, floating point addition or loading data.

8.4 Effect of loop unrolling on the execution time of compiled code

Loop unrolling can aid in the application of ILP. The body of a loop with n iterations is replicated multiple times, by a factor between 1 (no unrolling) and n (complete unrolling) and the iteration number adjusted accordingly. Because it exploits ILP better, it will decrease execution time, at the cost of compile time, program size and instruction cache space requirements.

8.5 Relation between Instructions per Cycle (IPC) and algorithm performance

In general, a high number of IPC signals efficient utilization of the CPU (via ILP). However, many IPC do not signify a performant algorithm per se. Other

factors, such as quality of branch prediction (which can indirectly be influenced in the code), cache misses (efficient memory access) and the used functions influence the performance, too. A highly inefficient algorithm that perfectly utilizes ILP, leading to high IPC, may still be slower than an unoptimized, efficient algorithm for the same problem.

9 Lecture 9

9.1 Bandwidth-bound computations vs compute-bound computations

Bandwidth-bound computations refer to programs or parts of programs where the execution speed is bottlenecked by the memory access. Compute-bound computations, on the other hand, will have the bottleneck at the CPU. The former of the two is much more common currently, which makes it very important to use memory as efficiently as possible by reducing load operations.

9.2 Effect of temporal locality and spatial locality on program performance

Because bandwidth-bound computations are commonplace, it is important to use as little load and store operations as possible. If a program adheres to the principles of temporal and spatial locality, already loaded data can be used most efficiently, therefore reducing the bottleneck. Temporal locality refers to the reuse of a memory location multiple times in a short period of time - thus reusing data already loaded into the cache. When a cache line is no longer necessary, it can then be discarded without reducing performance. Spatial locality refers to the surrounding memory locations. Optimally, memory locations close to the referenced location will appear in the near future. A good example of proper spatial locality is the previously discussed unit stride memory access.

9.3 Data-oriented design (DOD) vs object-oriented design (OOD)

In DOD, the final format is considered and used to achieve optimal transformations of the input to the desired output. The aim is to structure the data in a way that reduces the transformation effort applied in functions. This optimizes the CPU cache usage while making the program easier to parallelize. Commonly used data will therefore often be stored in a SoA.
OOD, in contrast, approaches the program design by modelling objects close to the real world concepts. When opposed to DOD, generally OOP may be easier to write and understand by virtue of being more verbose. Individual transformations, functions, and representations will likely not be efficient however, leading to a performance loss in execution.

9.4 Streaming stores

Streaming stores bypass the cache and writes data directly to RAM. Obviously, this saves the scarcely available cache, possibly improving performance. It is useful in situations where the data will not be read before or soon after being written. Otherwise, performance may even be negatively affected.

9.5 Typical cache hierarchy used in Intel CPUs

The Intel Core i7 Cache hierarchy has two L1 caches per core, with one for doubles and one for integers, respectively. There is one L2 cache per core, shared between doubles and integers (unified). All cores share the unified L3 cache, which is derived from the main memory. Naturally, the higher level caches are bigger.

9.6 Cache conflicts

Cache conflicts occur, when more memory is accessed than can be stored in cache lines, leading to cache accesses to the same cache locations, forcing constant write operations as well as reads in higher level caches on each access (thrashing) .

10 Lecture 10

10.1 Useful compiler flags during debugging

-Wall The flag activates all warnings during compilation, and can point to mistakes of all sorts, including stylistic mistakes (e.g. /* inside a multiline-comment)

-fsanitize when set to -fsanitize=address, shows memory leaks, as well as out-of-bounds and use-after-free accesses. Those may lead to undefined behavior instead of immediately crashing the program, obscuring critical errors in a program. When set to -fsanitized=undefined, all undefined behaviors will be marked at runtime.

-g The -g flag instructs the compiler to keep debug information in the executable, which necessary for debuggers and profilers.

10.2 Using Intel oneAPI for writing better programs

The Intel oneAPI toolkit has several tools for a better programming experience. Apart from containing the Intel C++ compiler, which can produce very performant, optimized code, it also contains several profiling and analysis tools:

- Intel Inspector
  The intel inspector can analyze multithreading programs. This can unveil deadlocks and data races in the program, which may not be obvious at first sight.
- Intel VTune Amplifier
  The program can diagnose several performance metrics in the analyzed program. This includes, but is not limited to cycles per instruction (are CPU calls chained well) or rate of cache misses (does the program lead to thrashing?). The amplifier also detects bottlenecks in the code, which take the most execution time, similarly to valgrind. This may point to worthwhile optimization targets.

10.3 Premature optimization is the root of all evil

(Oops, I wrote 5.2 before this lecture)
First, there is no point in optimizing code that does not run (robustly). Optimization can lead to more complicated code, which will make fixes down the line more complicated. Also, sometimes the effort of achieving minor improvement outweighs the benefit. Conversely, low effort changes sometimes lead to large improvements. It is therefore important to analyze a program's bottlenecks before blindly optimizing, perhaps with one of the tools discussed in this assignment batch.

11 Lecture 11

11.1 What is Cython?

Cython is a superset of the Python programming language, which can be compiled, instead of interpreted, like traditional Python would be. Cython also provides options for typization of variables (via the *cdef* keyword before a variable declaration/definition, or when specifying function parameters. Cython also offers an interface to C/C++, allowing the use of their libraries. Cython code gets compiled to C code, and the resulting library can be used in Python, optimally leading to performance similar to a C program.

11.2 Accelerating Python programs with Cython

As Cython is a superset of Python, the easiest way of accelerating a given Python program is by simply compiling it with the Cython compiler. By virtue of not having to be interpreted, the execution speed will already be improved. There is a high potential for performance increases by adding static typization to Python programs, as types don't have to be inferred when creating the C code from Python.
Other avenues of performance include the use of (efficient) C/C++ libraries, and the usage of Cython compiler directives (see 11.4) and reducing function call overhead with *cdef*/*cpdef* functions (see 11.5.

11.3 2 Ways of compiling .pyx modules

Cython modules can be compiled ahead of time by using the *cythonize* command line tool. Another option is using the *pyximport* library as shown below. This compiles the Cython modules in the working directory on the fly.

```
import pyximport
pyximport.install()
```

11.4 Selected compiler directives in Cython

Compiler directives in Cython are added at the top of a module in a header comment or through the command line (as explained in the Cython Docs). They enable/disable Python features, with varying impact on performance.

11.4.1 cdivision

This boolean directive allows for faster division according to C standards, for example skipping over zero division checks. According to the Cython docs, this can lead to up to a 35% performance increase for division when set to True.

11.4.2 language_level

This directive can be set to *2*, *3*, or *3str*, referring to the language version of Python. As the default setting is 2, and new Python projects rarely still use Python 2, it is a useful directive.

11.5 def, cdef, and cpdef

The keywords *def*, *cdef* and *cpdef* are all used in Cython to define functions (note: *cdef* is also used for assigning or defining variable types). *def* functions are equivalent to Python functions and can be used as such, even in Python code.
cdef functions, on the other hand, are not compatible with vanilla Python. This leads to reduced overhead in function calls, but does not provide an interface to pure Python programs.
cpdef functions define both a Cython and Python interface, thus providing the faster call convention in Cython, while still exposing the functions for usage in pure Python modules, where the performance gains are not realized.

11.6 Typed Memoryviews and their use in Cython

Typed memoryviews are a method of using vanilla or NumPy arrays and avoiding the overhead that would result from using the Python interface. The syntax is $< type >$ [:, ..., :] where each : indicates another dimension .

12 Lecture 12

12.1 Extension Types in Python

Extension types look and behave just like normal Python objects. In the background, they are compiled code, just like the built-in object types - *dict, list, tuple, etc.*, which improves performance. They can aid in wrapping C and C++ code, giving it a pythonic interface.

12.2 Extension Data Fields in Cython vs. Data Fields in Python Classes

Data fields of extension types are not accessible in Python, only Cython. This can be avoided by setting the *readonly* or *public* flag. The access speed for data fields in Cython is native (incurs no additional overhead). Python class fields are not as performant.

12.3 Wrapping C and C++ Code in Cython

From .pyx files, C or C++ header files (including necessary data types) can be imported. By supplying the Cython compiler with the necessary directives. The C/C++ functions are marked via *cdef* . *def* demarks Python-accessible functions, as per usual. From this point on, the functions can be imported through the Cython module.

13 Lecture 13

13.1 Cells vs. Pages vs. Blocks in SSDs

Cells: In an SSD, a cell is the smallest unit at which data is stored. Per cell, between 1 and 4 Bits are stored, depending on the use case. Generally, more bits per cell are cheaper, but less durable.
Pages: The smallest unit which can be written or read. Generally, a page contains 4 KB of data. Interestingly, there is no overwrite operation for pages, such that modified data gets stored as an altered copy on another page. Pages that have been 'modified' get marked as stale for later erasure.
Blocks: The smallest unit of erasure is the block. Typically, a block contains 128 or 256 pages, which amounts to 512 KB and 1 MB, respectively.

13.2 The Purpose of Garbage Collection in SSDs

Because pages of data cannot be deleted individually on SSDs, a high number of stale (updated) pages, especially if distributed among different blocks, can lead to a large amount of 'dead space'. If there are more invalid pages than a given threshold, the valid pages get moved to another block, to free up the old block for deletion.

13.3 The Purpose of Wear Leveling in SSDs

In SSDs, data is encoded through voltage in a cell. Over time, it becomes increasingly harder to differentiate 0s from 1s (this also explains why single level cells are more durable - as there are more threshold voltages). In order to not have some part of the SSD fail faster than others, data is written on spread pages among the SSD, which is controlled by the controller via the Flash Translation Layer (FTL). Data blocks that persist for a long time will also be read and rewritten periodically, to ensure the pages are not underutilized.

13.4 Features of SSDs with the M.2 Form Factor

For M.2 SSDs, NVMe can be used as a communication protocol via the host interface PCIe, which is much more performant compared to SATA interfaces, which use AHCI as a communication protocol. NVMe offers higher bandwidth, as well as larger transfer rates. But there also exist M.2 SSDs, which are used via SATA, not offering the improved performance, so the form factor alone does not necessarily imply the usable standard. Connector slots are also relevant for the connectivity. There are B key, M key, and B&M key connectors with different pin arrangements, which themselves may or may not support PCIe or SATA.

13.5 The Influence of Garbage Collection and Wear Leveling on Write Amplification

Write amplification describes the factor of data that has to be written logically versus data that is physically written to the SSD. Garbage collection and wear leveling both influence write amplification negatively. Through wear leveling, writes will be spread over different blocks and pages. When some of these pages are updated, that is copied, modified, and written to another page, garbage collection may come in to rewrite the blocks. As physical storage is generally abstracted from through logical files in operating systems, this could possibly have negative synergy effects concerning logically connected data that is physically distributed over different blocks (updating may lead to more garbage collection). This something the controller might take into account, though. Another scenario where wear leveling leads to higher write amplification is the reallocation of longstanding files.

13.6 Writing Code for SSDs

Using compact data structures: It is better to use files that are at least 4 KB or multiples of 4 KB large. This is because no read or write operations for units smaller than pages can be performed. Using one large file instead of multiple small files is also more performant, because data can be read from multiple threads better, utilizing bandwidth more optimally.
Multithreading for small IO (KB range): When data is not stored in large files, it is preferable to use 16-64 threads for the actions. As mentioned in the first recommendation, this leads to proper utilization of the SSDs queue and available bandwidth. The concrete number of threads depends on the SSD and its communication protocol.
Multithreading for big IO (¿10 MB): Because SSDs have an internal parallelism mechanism, not as many threads (2-4) are necessary to perform IO on larger files. A too high number of threads will actually lead to a reduced throughput.

13.7 On Reducing CPU load for IO

Synchronous IO operations on the SSD block threads in the same way that a spinning lock might otherwise, leading to a large CPU load without any benefits. In this case, using asynchronous calls to the system can greatly reduce the load on the CPU. Streaming stores and reads can also reduce CPU load by not caching and letting the application manage that. The latter point only applies primarily to DBMS.

13.8 Solve Problems that do not fit in DRAM without (major) code adjustments

As mentioned in the lecture, NVMe SSDs can be utilized to provide near in-memory speeds for calculations. Subramanya et. al achieved this via pipeline IO (loading the next data while calculating) with the computations of a linear algebra library. The pipelining can partially hide the latency of SSDs in comparison to RAM access. A similar, easier approach to efficiently utilizing large amounts of flash disk data is memory mapping files. In this case, files can be interpreted in the code as usual arrays, with the operating system handling all memory-related issues.