Exam Assignments

1. Vorlesung

1. Describe how parallelism differs from concurrency.

Concurrency gilt, wenn es mehrere Handlungsstränge gibt, in denen gleichzeitig Prozesse durchgeführt werden. An diesen muss jedoch nicht tatsächlich zeitgleich gearbeitet werden. Um nur *Concurrency* zu erfüllen, reicht es, mehrmals zwischen den Strängen zu wechseln und dabei je Teilprozesse auszuführen. Dadurch wirkt es zwar so, als würden die Prozesse zeitgleich ablaufen, in Wahrheit tun sie dies aber nicht.
Parallelism ist eine Unterkategorie der *Concurrency*. Hier muss zusätzlich gelten, dass die Prozesse auf den einzelnen Handlungssträngen tatsächlich gleichzeitig ablaufen.

2. What is fork-join parallelism?

Fork-join parallelism ist ein Modell zum Parallelisieren von Prozessen. Hierbei gibt es zunächst einen Master Thread, der durchgehend läuft. An bestimmten Stellen im Programm erzeugt dieser eine bestimmte Anzahl weiterer Threads (*fork)*, welche im Folgenden parallel an einem Task arbeiten. Sobald dieser abgeschlossen ist, werden die Threads wieder zusammengeführt (*join)* und der Master Thread arbeitet allein weiter.
Dieser Prozess kann bei Bedarf auch geschachtelt werden. Das heißt, dass jeder der parallelen Threads ebenfalls die Möglichkeit hat, seinen Task auf weitere parallele Threads zu verteilen.

3. Read Chapter 1 from *Computer Systems: A Programmer's Perspective*

a. Link: A Tour of Computer Systems

b. Discuss one thing you find particularly interesting. (google it to find more information)

Ich fand es interessant, genaueres darüber zu erfahren, was beim Kompilieren eines Programms im Hintergrund passiert. Ich wusste zwar grob über den Compiler und den Assembler Bescheid, hatte jedoch bisher keine Ahnung von den weiteren Phasen, die beim Kompilieren durchlaufen werden.
Beispielsweise kannte ich die *Preprocessing*-Phase nicht, die als erstes durchgeführt wird. In dieser wird der Code der importierten Bibliotheken direkt in das Programm eingefügt.
Erst im nächsten Schritt kommt der Compiler an die Reihe, der den so entstandenen Code in Assembler-Sprache übersetzt. Dabei wird oft eine Operation des ursprünglichen Codes auf mehrere Assembler-Instruktionen aufgeteilt. Im Gegensatz zu *high-level* Code steht in Assembler-Code nämlich jedes Statement für genau eine einzige *low-level* maschinensprachliche Instruktion. Der Code ist nach diesem Schritt jedoch immer noch in Textform und damit für menschliche Programmierer lesbar. Was die Assembler-Sprache außerdem praktisch macht, ist, dass sie von einer Vielzahl an Compilern verwendet wird und damit eine gemeinsame Ausgabesprache für verschiedene Compiler und Programmiersprachen bietet.

Nach der Compiler-Phase folgt die Assembly-Phase. In dieser wird der Assembler-Code in maschinensprachliche Instruktion übersetzt, in ein sogenanntes *relocatable object program* verpackt und anschließend in eine binäre Objektdatei gespeichert. Ab diesem Schritt ist der Code für menschliche Programmierer nicht mehr lesbar, da er nicht mehr im Textformat gespeichert ist.
In der letzten Phase, der *Linking*-Phase, werden Funktionen der *Standard Library*, wie beispielsweise `printf()` mit dem Programm gemergt. Diese Funktionen sind standardmäßig in eigenen vorkompilierten Objektdateien.
Nach all diesen Schritten erhält man eine ausführbare Objektdatei, ein sogenanntes *Executable*.

4. Read the paper *There's plenty of room at the Top: What will drive computer performance after Moore's law?*

a. Click here to download the paper

b. Explain in detail the figure *Performance gains after Moore's law ends*. (on the first page)

Die Abbildung stellt den "*Computertechnik-Stack*" dar, an dessen Grund (*Bottom*) sich die kleinsten Komponenten wie bspw. *Semiconductor*-Technologie befinden. Die Spitze (*Top*) hingegen bilden Software, Algorithmen und Hardwarearchitekturen. Nachdem die Wachstumsmöglichkeiten am *Bottom* so gut wie ausgeschöpft sind, wendet man sich nun der Spitze zu. Die Abbildung geht auf die einzelnen Komponenten, die die Spitze bilden, ein und zählt dabei anhand von Beispielen Optimierungsmöglichkeiten dieser auf.
Für Software wird *Performance Engineering* als Möglichkeit genannt, Programme effizienter zu machen. Dabei wird die Software in einer Art umstrukturiert, dass ineffiziente Programmteile, sogenannte *Software Bloats*, entfernt werden. Weiterhin kann die Software an die Bedingungen der verwendeten Hardware angepasst werden, um bspw. parallele Prozessoren oder Vektoreinheiten auszunutzen.
Als Möglichkeit, Algorithmen effizienter zu machen, wird das Entwickeln neuer Algorithmen aufgezählt. Da die Entwicklung neuer Algorithmen für spezifische Probleme jedoch sehr ungleichmäßig und sporadisch geschieht, kann man sich nicht darauf allein verlassen. Stattdessen sollte man sich auf Algorithmen für neue Problemdomänen, wie bspw. *Machine Learning*, konzentrieren. Auch die Entwicklung neuer Maschinenmodelle, die die Hardware besser reflektieren, sind eine gute Möglichkeit.
Zu guter Letzt widmet sich die Abbildung der Hardwarearchitektur. Hier wird das *Streamlinen* dieser durch bspw. die Vereinfachung des Prozessors vorgeschlagen. Ein komplexer Prozessorkern könnte durch einen einfacheren Kern mit weniger Transistoren ersetzt werden. Das dadurch entstandene Budget könnte dann bspw. dafür verwendet werden, die Anzahl der parallel laufenden Kerne zu erhöhen. Dadurch würde die Parallelisierung von Programmen deutlich effizienter werden. Eine weitere Möglichkeit ist die Domänenspezialisierung. Hier wird die Hardware auf ein bestimmtes Anwen-

dungsgebiet angepasst, wodurch Hardwarefunktionalitäten, die dort nicht benötigt werden, weggelassen werden können.

5. Do the coding warmup on slide 20. Parallelize the program of slide 22.

Code s.: `./Antworten/CodingWarmups_1/`
Output:

		Numeric_Integration	Numeric_Integration_2	Monte_Carlo
ohne -Ofast	π	3.141592653589746 oder 3.1415926535897452	3.1415926535896244 oder 3.1415926535896248 oder 3.1415926535896253	3.14197
	Dauer	ca 0.084 (± 0.003)	ca 0.08 (± 0.003)	ca. 2.64 (± 0.01)
mit -Ofast	π	3.1415926535897452	3.1415926535898491	3.14197
	Dauer	ca. 0.038 (± 0.002)	ca. 0.022 (± 0.002)	ca. 0.47 (± 0.03)

2. Vorlesung

1. What causes false sharing?

False Sharing tritt auf, wenn von mehreren Prozessoren aus auf dieselbe Cache Line zugegriffen und ein Wert auf dieser geändert wird. Der geänderte Wert muss anschließend überall, wo die entsprechende Cache Line verwendet wird, aktualisiert werden. Dies passiert unabhängig davon, ob der geänderte Wert an der entsprechenden Stelle gebraucht wird. Als Folge können andere Prozesse erst auf Werte der Cache Line zugreifen, nachdem diese bei ihnen aktualisiert wurde, was zu unnötigen Wartezeiten führt. Um diese Problematik zu umgehen, sollte das unnötige Teilen von Cache Lines vermieden werden.

2. How do mutual exclusion constructs prevent race conditions?

Bei einer *Race Condition* versuchen mehrere Threads gleichzeitig den Wert derselben geteilten Variable zu ändern. Das kann dazu führen, dass es zu einem *Lost Update* kommt. Das heißt, dass die Schreiboperation des ersten Threads durch einen zweiten überschrieben wird und dadurch verloren geht.
Um dieses Problem zu vermeiden, gibt es sogenannte *Mutual-Exclusion*-Konstrukte. Diese stellen sicher, dass zu jedem Zeitpunkt nur ein Thread exklusiven Zugang auf die betroffene geteilte Variable hat. Nur dieser kann in diesem Zeitraum auf die Variable zugreifen und sie ändern. Alle weiteren Threads müssen solange warten, bis ihnen selbst dieser exklusive Zugang erteilt wird.

3. Explain the differences between static and dynamic schedules in OpenMP.

Die `Schedule`-Klausel kann verwendet werden, um die Aufteilung der Threads bei Schleifen zu optimieren.
Bei der statischen Variante werden die Threads zur Kompilierzeit gleichmäßig auf die Schleife verteilt. Das heißt, jeder Thread bekommt (soweit möglich) dieselbe Anzahl an Iterationen zugewiesen. Der tatsächliche Arbeitsaufwand der einzelnen Iterationen wird dabei nicht berücksichtigt. Diese Variante ist gut, wenn alle Iterationen denselben Arbeitsaufwand haben.
In der dynamischen Variante werden die Iterationen erst zur Laufzeit zugewiesen. Das heißt, erst sobald ein Thread mit seiner Iteration fertig ist, wird ihm die nächste Iteration zugewiesen. Dies ist von Vorteil, wenn die Iterationen einen unterschiedlich großen Arbeitsaufwand haben, da dieser so besser und gleichmäßiger auf die einzelnen Threads verteilt werden kann.

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?

Es ist in diesem Fall nicht möglich, die `for`-Schleife abzubrechen. Stattdessen kann man aber eine passende `If`-Abfrage (bspw. mit einem beim Finden der Lösung gesetzten

boolean) nutzen. Innerhalb dieser kann man mit einem `continue` alle folgenden Iterationen übergehen.

5. Do the coding warmup on slide 22.

a) Fix the race condition bug on page 13 with a std::mutex.

Code s.: `./Antworten/CodingWarmups_2/`

b) Reduce the runtime of the image denoising program on slide 16 by adding an appropriate schedule.

Code s.: `./Antworten/CodingWarmups_2/`

Outputs:

Schedule	Dauer
schedule(static)	1,95 sek
schedule(static, 4)	1,09 sek
schedule(static, 2)	0,96 sek
schedule(static, 1)	0,92 sek
schedule(dynamic)	0,85 sek
schedule(dynamic, n) mit $2 \leq n \leq 6$	0,86 sek

c) What schedule on slide 18 produces the following (bad) pattern?

Dieser Output wird durch `schedule(static, 22)` produziert.

d) Modify the program on slide 21 so that it always finds the smallest possible solution.

Code s.: `./Antworten/CodingWarmups_2/`

6. Explain in your own words how std::atomic::compare_exchange_weak work.

Um diese atomare Funktion zu verwenden, ruft man sie auf einer atomaren Variable auf. Dabei übergibt man ihr zwei Variablen: die Referenz einer *expected*-Variable und eine *desired*-Variable. Erstere enthält den Wert, von dem man erwartet, dass ihn die atomare Variable aktuell hat. Da dieser nur per Referenz übergeben wird, kann er innerhalb der Operation atomar verändert werden. Die zweite Variable beinhaltet den Wert, mit dem man die atomare Variable überschreiben will. Dies geschieht jedoch nur, wenn die atomare Variable tatsächlich denselben Wert hat, der auch in der *expected*-Variable gespeichert ist. In diesem Fall wird die atomare Variable überschrieben und *true* zurückgegeben. Sollten sich die Werte jedoch unterscheiden, wird stattdessen der aktuelle Wert der atomaren Variable in die *expected*-Variable geschrieben und *false* zurückgegeben.

7. **Rewrite the parallel program for estimating π from the last exercise to use the construct #pragma omp for.**

   Code s.: `./Antworten/CodingWarmups_2/`

3. Vorlesung

1. How does the ordered clause in OpenMP work in conjunction with a parallel for loop?

Um diese Klausel zu benutzen, muss man innerhalb der parallelen `for`-Schleife mit `#pragma omp ordered` eine `ordered`-Region erstellen. Der Code innerhalb dieser Region wird dann so ausgeführt, als wäre er innerhalb einer gewöhnlichen, sequentiellen `for`-Schleife. Das Nutzen einer solchen Region bietet sich an, wenn die Operationen innerhalb ihr nicht rechenintensiv sind, aber in einer bestimmten Reihenfolge ausgeführt werden sollten. Man kann so also den rechenintensiven Teil parallelisieren, während gleichzeitig weniger rechenintensive Operationen sequentiell und geordnet ausgeführt werden. Um die Effizienz zu steigern, ist es empfehlenswert `ordered` zusammen mit `schedule(dynamic)` zu verwenden.

2. What is the collapse clause in OpenMP good for?

Mit dieser Klausel können verschachtelte for-Schleifen in eine einzige Schleife kollabiert werden. Die Threads werden dann also nicht nur auf die Iterationen der äußeren for-Schleifen verteilt, sondern auf das Produkt der Iterationsanzahl aller angegebenen for-Schleifen. Die Anzahl an zu parallelisierenden for-Schleifen wird dabei in *collapse(x)* an der Stelle *x* angegeben.

3. Explain how reductions work internally in OpenMP.

Bei *Reductions* handelt es sich um Operationen, die assoziativ und kommutativ sind. Sie können genutzt werden, um eine Vielzahl an Werten auf ein einziges Ergebnis zu reduzieren. Man erspart sich dadurch also u.a. das Anlegen lokaler Zwischenvariablen für jeden Thread und das bspw. atomare Zusammenrechnen dieser.
Zur Verwendung von *Reductions* schreibt man `reduction(op : list)` und gibt dabei als `op` den gewünschten Operator (bspw. '`+`', '`max`' oder '`&`') und anstelle des `list` eine Aufzählung der gewünschten Variablen an.
Intern wird dabei für jeden Thread eine lokale Kopie jeder der aufgelisteten Variablen angelegt. Abhängig von der angegebenen Operation wird diese entsprechend initialisiert. Der Thread aktualisiert nun zunächst seine lokale Kopie und rechnet den Wert dieser zum Schluss auf die globale Variable.

4. What is the purpose of a barrier in parallel computing?

Befindet sich im parallelen Bereich eine mit `#pragma omp barrier` gekennzeichnete Barriere, so wird der Code nach dieser erst ausgeführt, wenn jeder Thread die Barriere erreicht hat. Bis dahin müssen alle bereits an der Barriere angekommenen Threads warten. Eine solche Barriere kann zur Synchronisation der Threads explizit gesetzt werden.

Es gibt jedoch auch implizite Barrieren, die beim Verwenden bestimmtere Konstrukte automatisch von OpenMP gesetzt werden. Diese Konstrukte sind `for`, `sections`, `single` und `parallel`. Nur im Falle der ersten drei Konstrukte kann diese Barriere durch eine `nowait` Klausel deaktiviert werden.

5. Explain the differences between the library routines *omp_get_num_threads()*, *omp_get_num_procs()* and *omp_get_max_threads()*.

`omp_get_num_threads()` gibt die Anzahl der aktiven Threads innerhalb der aktuellen parallelen Region zurück. Außerhalb einer parallelen Region gibt die Funktion stets 1 zurück.
`omp_get_num_procs()` gibt die Anzahl der logischen Cores zurück. Diese kann von der Anzahl der physischen Cores abweichen. Mit dieser Information kann beispielsweise von Hand festgelegt werden, wie viele Threads in einer parallelen Region verwendet werden sollen.
`omp_get_max_threads()` kann außerhalb einer parallelen Region aufgerufen werden, um zu ermitteln, wie viele Threads innerhalb einer parallelen Region maximal verwendet werden. Wurde dieser Wert nicht durch `omp_set_num_threads()` geändert, so entspricht er der Anzahl an Threads, die maximal von der Hardware unterstützt werden.

6. Clarify how the storage attributes private and firstprivate differ from each other.

Übergibt man eine Variable mit `private`, so bekommt jeder Thread eine nicht-initialisierte Kopie dieser Variable. Die Threads können also nicht auf den ursprünglichen Wert der Variable zugreifen.
Übergibt man Variablen hingegen mit `firstprivate`, so bleiben sie initialisiert. In diesem Fall bekommt jeder Thread eine lokale Kopie der ursprünglichen Variable.
In beiden Fällen wird die ursprüngliche globale Variable nicht verändert.

7. Write in pseudo code how the computation of π can be parallelized with simple threads.

```
1   num_steps = n
2   num_Treads = m
3   width = 1.0 / num_steps
4   sum = 0.0
5
6   DO parallel WITH "num_Treads" Threads:
7      sumLoc = sum  //initialisiert lokale "sum" Variable
8      //thread_Id ... Id des aktuellen Threads
9      FOR i = thread_Id TO num_steps:
10        x = (i + 0.5) * width
11        sumLoc = sum + (1.0 / (1.0 + x * x))
12        i = i + num_Treads //inkrementiert um die Anzahl an Threads
13     END FOR
14     START CRITICAL_SECTION:
15        sum = sum + sumLoc
16     END CRITICAL_SECTION
17  END parallel
18
19  pi = sum * 4 * width
```

4. Vorlesung

1. Explain how divide and conquer algorithms can be parallelized with tasks in OpenMP.

Um einen Divide-and-Conquer-Algorithmus mit OpenMP *Tasks* zu parallelisieren, führt man diesen zunächst innerhalb eines parallelen Bereichs (`#pragma omp parallel`) aus. Dabei ist es jedoch wichtig, dass man mit nur einem Thread beginnt (`#pragma omp single`). Anschließend wird innerhalb des Algorithmuses für jeden rekursiven Aufruf ein eigener Task mit `#pragma omp task` erstellt. Diese Tasks werden nun parallel laufen und rekursiv weitere parallele Tasks erstellen. Beim Beispiel von Merge-Sort hieße das, dass man für jede der beiden gesplitteten Arrayhälften einen neuen Task erstellt, in dem erneut der Algorithmus (Merge-Sort) ausgeführt wird. Bei Merge Sort braucht man zusätzlich noch ein `#pragma omp taskwait` im Anschluss an die Tasks. Dieses bewirkt, dass der darauf folgende Code erst ausgeführt wird, nachdem alle vorherigen Tasks abgeschlossen sind. An dieser Stelle werden die nun sortierten Arrayhälften wieder zusammengemergt. In anderen Algorithmen könnten ebenfalls noch weitere Operationen hinter einem solchen *Taskwait* stehen.

2. Describe some ways to speed up merge sort.

Eine Möglichkeit ist die Parallelisierung durch bspw. das Verwenden von OpenMP wie in Aufgabe 1) beschrieben. Das ist besonders bei sehr großen Arrays hilfreich. Zusätzlich kann man noch hinzufügen, dass Tasks nur für große Arrays erstellt werden, da das Erstellen von zu vielen Tasks die Performance letztendlich negativ beeinflussen kann.
Weiterhin kann man kleinere Arrays mit *Insertion Sort* sortieren, anstatt weiterhin rekursiv vorzugehen. Da die Rekursion selbst auch teuer ist, ist dieser Ansatz für kleine Arrays deutlich effizienter.
Auch nachdem man diese Schritte angewendet hat, wird jedoch weiterhin die teuerste Operation, das Mergen, *single threaded* gemacht. Um das Mergen trotzdem zu beschleunigen, kann man die Liste, in die gemergt wird, bei kleineren Arrays auf dem Stack erstellen statt auf dem Heap. Das ist schneller, da der Stack bei jedem Funktionsaufruf automatisch allokiert und nach dem *return* automatisch freigegeben wird. Bei zu großen Arrays kommt es jedoch zu einem *Stack-Overflow-Fehler*, weshalb man das nur bei kleineren Arrays machen kann. Eine gute Grenze dazu ist eine Arraygröße von 8192.

3. What is the idea behind multithreaded merging?

Das Ziel des Merging ist, zwei sortierte Arrays zu einem einzelnen sortierten Array zu kombinieren. Dazu wird zunächst das mittlere Element des größeren der beiden Arrays bestimmt. Anschließend werden im zweiten Array mithilfe von binärer Suche alle Elemente bestimmt, die kleiner als dieser Median sind. Summiert man die Anzahl dieser mit der Anzahl an Elementen, die im ersten Array vor dem Median stehen, so erhält man

die Position, an welcher der Median im gemergten Array stehen wird. Damit kann der Median bereits in das neue Array geschrieben werden.
Dieses Verfahren wird mit den so entstandenen Subarrays auf beiden Seiten des Medians so lange rekursiv wiederholt, bis der Mergeprozess abgeschlossen ist. Da die Subarrays mit den Elementen, die kleiner als der Median sind, unabhängig von denen sind, die größer als der Median sind, kann der Prozess auf verschiedene Threads aufgeteilt werden. Durch das einfache Lookup des Medians und die binäre Suche hat der Algorithmus eine schnelle *Divide*-Phase und verzichtet dank dieser auf eine teure *Combine*-Phase.

4. **Do the coding warmup on slide 17.**

    a. **Try to speed up the naive merge sort implementation on page 7.**
    b. **On page 15 you have become acquainted with multithreaded merging. On page 18 is an actual singlethreaded implementation of the algorithm. Parallelize the singlethreaded implementation with tasks.**

5. **Read *What every systems programmer should know about concurrency*.**
   https://assets.bitbashing.io/papers/concurrency-primer.pdf
   Discuss two things you find particularly interesting.

    Ich fand es interessant, wie viele atomare *Read-Modify-Write*-Operationen es in C und C++ gibt. Die *Exchange*-Operation liest einen Wert und ersetzt ihn atomar durch einen neuen Wert. Dadurch kann man Race Conditions zwischen den Lese- und Schreiboperationen vermeiden. *Test-and-Set* ist eine Operation für Booleanwerte. Der aktuelle Wert wird gelesen, auf `true` gesetzt der vorherige Wert zurückgegeben. Das kann genutzt werden, um ein einfaches *Spinlock* zu bauen. Eine weitere Operation ist *Fetch-and-...*, welche atomar einen Wert liest und anschließend eine simple Operation wie bspw. Addition oder bitweises `AND` mit ihm ausführt. Danach wird der vorherige Wert wieder zurückgegeben. Die letzte Operation ist *Compare-and-Swap* (CAS), auch *Compare-and-Exchange* genannt. Diese Operation tauscht einen Wert aus, wenn dieser einem Erwartungswert entspricht.

    Weiterhin fand ich es interessant, dass Compiler unter bestimmten Umständen *Loop Unrolling* auf eine Schleife anwenden können. Dazu kann es kommen, wenn man eine *relaxte* atomare Ladeoperation in der Schleifenbedingung durchführt. Dieses Phänomen nennt man *Atomic Fusion*. Will man es verhindern, kann man `volatile`-Casts oder `asm volatile("" ::: "memory")` verwenden. Linux stellt zu diesem Zweck explizit die Makros `READ_ONCE()` und `WRITE_ONCE()` zur Verfügung.

5. Vorlesung

1. What is CMake?

Bei CMake handelt es sich um eine kommandobasierte, plattformunabhängige Skriptsprache, mit der man Build-Skripte für verschiedene IDEs schreiben kann. In diesen kann man unter anderem die Source-Dateien, die verwendeten Bibliotheken oder die Compilierflags eines Projektes festlegen. Auch kann man mit CMake Projekte bauen (*cmake*), testen (*ctest*) oder sie in *Packages* verpacken (*cpack*). Letzteres kann zum Erstellen von bspw. Webdateien oder Windows-Executables verwendet werden. CMake ersetzt jedoch keinen Compiler, da es Projekte nicht selbst bauen kann, sondern nur die dazu benötigten Skripte erstellt.
Um CMake in einem Projekt zu verwenden, wird mindestens eine *CMakeList.txt*-Datei benötigt. Bei Bedarf können auch mehrere dieser Dateien für verschiedene Ordner verwendet werden.

2. What role do targets play in CMake?

Targets sind das, was man mit CMake bauen will. Das können bspw. Executables sein oder auch eine Bibliothek oder Testfälle.
Wenn man Targets mit objektorientierter Programmierung vergleicht, kann man sie sich wie Objekte vorstellen. Wie diese haben sie ebenfalls Konstruktoren, denen bestimmte *Properties* (vgl. *member variables*) übergeben werden, um die Attribute des targets zu verändern. Im Falle von Targets können das bspw. *Source*-Dateien, Flags, eingebundene *Directories* oder zu verwendende Bibliotheken sein.

3. How would you proceed to optimize code?

Zunächst schreibt man einen korrekten und funktionierenden Code, ohne sich Gedanken über Optimierungen zu machen. Anhand von Testfällen, die man währenddessen oder danach schreibt, kann man diese Korrektheit nachweisen und prüfen. Sollte anschließend die Performance des Programms zufriedenstellend sein, kann man an diesem Punkt aufhören. Andernfalls beginnt man mit der Optimierung. Fällt einem keine Möglichkeit zum Verbessern des Algorithmus selbst ein, so sollte man nach *Bottlenecks* suchen. Das kann man mit einem Profiler machen. Die Optimierung dieser kann die Performance bereits um einiges verbessern. Anschließend kann man das Programm parallelisieren oder vektorisieren, um die Performance weiter zu steigern. Ist der Optimierungsprozess abgeschlossen, testet man das Programm erneut, um sicherzustellen, dass es weiterhin korrekt ist. Diese Schritte wiederholt man so lange, bis eine zufriedenstellende Performance erreicht ist.

6. Vorlesung

1. Name some characteristics of the instructions sets: SSE, AVX(2) and AVX-512.

Das älteste der vorgestellten Instruktionssets ist SSE, welches zwischen 1999 und 2009 veröffentlicht wurde. Die Vektoren dieses Sets haben eine Größe von je 128 Bit, was 4 *Floats* bzw 2 *Doubles* pro Vektor entspricht. Es gibt insgesamt 16 Register, welche die Namen `xmm0` bis `xmm15` tragen.
Das auch heute noch am weitesten Verbreitete Instruktionsset ist AVX2, welches 2013 vorkam. Wie auch sein Vorgänger AVX (2011), hat dieses Set 16 Vektorregister mit Namen `ymm0` bis `ymm15`. Die einzelnen Vektoren haben je eine Größe von 256 Bits und können somit je 8 *Floats* bzw 4 *Doubles* speichern. Im Vergleich zu AVX gibt es jedoch mehr Instruktionen.
Das neueste Set ist AVX-515, welches 2017 veröffentlicht wurde. Es weist ganze 2 Register mit den Namen `zmm0` bis `zmm31` auf. Wie bereits der Name vermuten lässt, haben die Vektoren dieses Sets eine Größe von 512 Bits. Sie können somit je 16 *Floats* bzw 8 *Doubles* speichern. Auch gibt es eine Vielzahl neuer Vektoroperationen, was den Einstieg entsprechend erschwert.
Eine Besonderheit dieser Sets ist, dass sie nach oben hin kompatibel sind. So kann beispielsweise ein in SSE geschriebenes Programm auf einer AVX-515 CPU ausgeführt werden. Zu beachten ist jedoch, dass dabei natürlich nicht die maximale Performance erreicht werden kann. Andersherum gilt diese Kompatibilität jedoch nicht, da bspw. ein 512 Bit großer Vektor nicht in einem 128 Bit großen gespeichert werden kann.

2. How can memory aliasing affect performance?

Memory Aliasing tritt auf, wenn eine Funktion zwei oder mehr Pointer oder Referenzen übergeben bekommt. Der Compiler kann dann nicht entscheiden, ob diese auf denselben oder verschiedene Speicherbereiche zeigen. So weiß er bei Arrays beispielsweise nicht, ob diese sich überlappen oder sogar gleich sind. Das hat zur Folge, dass der Compiler sicherheitshalber annehmen muss, dass auf den gleichen Speicherbereich gezeigt wird. Dadurch kann das Programm nicht optimal optimiert werden und nicht die maximale Performance erreicht werden.
Um dies zu umgehen, kann man dem Compiler durch ein `restrict` die Information geben, dass sich die Speicherbereiche nicht überlappen. So kann eventuell performanterer, vektorisierter Code produziert werden.

3. What are the advantages of unit stride (stride-1) memory access compared to accessing memory with larger strides (for example, stride-8)?

Bei *stride-1* liegen die Datenelemente direkt hintereinander, wodurch sequentiell auf sie zugegriffen werden kann. Bei *stride-8* hingegen wird nur jedes achte Element verwendet. Das resultiert darin, dass bei *stride-8* die Bandbreite nicht voll ausgenutzt werden kann. Im Gegenteil, es kann sogar zu einem Bottleneck durch die maximal unterstützte Bandbreite kommen. Der Grund dafür ist, dass bei *stride-8* die Elemente auf verschiedenen Cachelines liegen. Es werden also ständig neue Daten geladen, wovon der Großteil nicht einmal benutzt wird. Bei *stride-1* jedoch liegen die Daten direkt hintereinander, wodurch jedes geladene Element auch tatsächlich verwendet wird. Dadurch kann die Bandbreite maximal ausgenutzt werden.
Ein weiterer Vorteil von *stride-1* ist, dass es die Vektorisierung durch den Compiler vereinfacht. Die Daten können hier einfach in einen Vektor gespeichert werden. Bei größeren *strides* muss dazu zunächst eine teure *gather*-Funktion aufgerufen werden, wodurch das Programm eventuell sogar langsamer wird.

4. When would you prefer arranging records in memory as a Structure of Arrays?

Bei *Structure of Arrays* (SoA) werden Objekte im Speicher so angeordnet, dass zunächst ein *Struct* erstellt wird. Innerhalb dieses *Structs* wird dann für jedes Attribut dieser Objekte ein einzelner Array angelegt. Hat man beispielsweise zwei Attribute, x und y, so wird in einem Array das x-Attribut aller Objekte gespeichert und in einem anderen ihr y-Attribut.
Dieser Ansatz ist von Vorteil, wenn man Berechnungen auf den einzelnen Attributen durchführen will, da man dann einen *stride* von eins hat. Beispiele dafür sind die Berechnung des Minimums des x-Attributs oder der Summe aller y-Attribute. Diese Berechnungen lassen sich mit SoA gut vektorisieren und Speicherzugriffe mit hohem *stride* werden vermieden.
SoA ist jedoch von Nachteil, wenn man alle Attribute eines einzelnen Objektes haben will. Da die Lokalität des Objektes zerstört ist, muss man hierfür in einer Vielzahl an Arrays nachsehen. *Array of Structures* (AoS) ist in diesem Fall deutlich performanter.

Zusatz:
Bei *Array of Structures* (AoS) werden alle Attribute aller Objekte in einem einzelnen Array gespeichert. Dieser ist nach Objekt sortiert, sodass alle Attribute eines bestimmten Objektes stets aufeinander folgen. Das Speicherformat ist also gut geeignet, wenn man auf verschiedene Attribute eines einzelnen Objekts zugreifen will, da diese im Speicher aufeinander folgend vorkommen. Will man jedoch für alle Objekte eine Berechnung auf einem bestimmten Attribut durchführen, so hat man einen *stride* von *n*, wobei *n* die Anzahl an unterschiedlichen Attributen ist (im Beispiel von oben hätte man also *stride-2*). Dies wird bei vielen Attributen sehr teuer.

7. Vorlesung

1. Explain three vectorization clauses of your choice that can be used with *#pragma omp simd*.

Bevor man `#pragma omp simd` verwendet, sollte man man sicherstellen, dass die entsprechende for-Schleife auch wirklich vektorisierbar ist. Es darf also keine Abhängigkeiten zwischen den in ihr verwendeten Elementen geben.
Eine sehr nützliche Klausel ist die *`aligned`*-Klausel. Mit dieser wird dem Compiler die Information übergeben, dass die in ihr angegebenen Pointer auf der ebenfalls angegebenen Anzahl an Bytes *aligned* sind. Die Zeile `#pragma omp simd aligned(a, b: 64)` gibt also an, dass die Pointer `a` und `b` auf je 64 Byte aligned sind. Wenn man mit Variablen arbeitet, die aligned sind, ist es empfehlenswert dies auch auf diese Weise anzugeben. Man muss jedoch sicherstellen, dass die angegebenen Pointer auch tatsächlich mit der angegebenen Bytezahl aligned sind. Andernfalls kommt es zu einem Fehler.
Nutzt man verschachtelte *`for`*-Schleifen, kann man wie auch bei `#pragma omp parallel for` die *`collapse`*-Klausel benutzen indem man `#pragma omp simd collapse(x)` direkt über die äußerste Schleife schreibt. Dabei ist `x` die Anzahl an for-Schleifen, die kollabiert werden sollen. Die so markierten *`for`*-Schleifen werden durch das Pragma in eine einzige Schleife kollabiert, wobei auf jeder Iteration dieser *simd* angewandt wird.
Die *`reduction`*-Klausel kann verwendet werden, wenn man innerhalb einer Schleife eine bestimmte Operation auf allen Elementen durchführen und das Ergebnis zurückbekommen will. Beispiele dafür sind Addition, Multiplikation oder logische Operationen wie *`AND`* und *`OR`*. Im Gegensatz zur *`reduction`*-Klausel bei `#pragma omp parallel` kann diese bei *simd* jedoch nicht für *`min`*- und *`max`*-Operationen verwendet werden. Benutzen kann man die Klausel durch folgende Zeile: `#pragma omp simd reduction(+: sum)`. In diesem Beispiel wird der Operator "+" auf die Variable *`sum`* angewendet. Zur Berechnung der Summe erhält jeder parallele Thread eine als *`private`* initialisierte lokale Kopie von *`sum`* und rechnet zunächst nur auf dieser. Am Ende werden alle lokalen Kopien aufsummiert und die Gesamtsumme im globalen *`sum`* gespeichert.

2. Give reasons that speak for and against vectorization with intrinsics compared to guided vectorization with OpenMP.

Ein Vorteil von OpenMP ist, dass es unkompliziert und damit einfach in den Code einzuarbeiten ist. Allerdings hängt es vom Compiler ab, ob und inwiefern die so gesetzten Flags beachtet werden. Man hat also keine Garantie, dass die Instruktionen tatsächlich umgesetzt werden. Abhängig von der Komplexität der Berechnungen und der Qualität des Compilers kann es daher passieren, dass theoretisch vektorisierbarer Code nicht vektorisiert wird.

Da Vektor-Intrinsics compilerunabhängig sind, haben sie diese Probleme nicht und laufen auf allen Systemen gleich performant. Sie sind dafür allerdings etwas komplizierter in der Verwendung.

3. What are the advantages of vector intrinsics over assembly code?

Einer der deutlichsten Vorteile ist, dass Vektor-Intrinsics einfacher zu lernen und zu nutzen sind. Auch ist der Code deutlich lesbarer als Assembly Code. Da man außerdem die zu verwendenden Register nicht explizit angeben muss, braucht man sich keine Sorgen zu machen, aus Versehen ein falsches Register zu überschreiben oder zu verwenden. Das macht sie auch portabler, da man sich einerseits keine Gedanken um die verfügbare Menge an Registern machen muss und sie außerdem compiler- und betriebssystemunabhängig sind. Des Weiteren entsprechen Vektor-Intrinsics oft einer oder mehrerer Assembly Instruktionen, die innerhalb der Wrapperfunktion nacheinander, teils sogar optimiert, ausgeführt werden.
Alles in Allem sind Vektor-Intrinsics also leichter zu verwenden als Assembly Code, ohne dabei Performance einbüßen zu müssen.

4. What are the corresponding vectors of the three intrinsic data types: *__m256*, *__m256d* and *__m256i*.

Die `256` in `__m256` weist auf einen Vektor mit 256 Bit hin, wie sie in AVX oder AVX2 verwendet werden. Diese Variante ohne einen Schlussbuchstaben ist für Floats. Sie kann 8 Float-Werte halten. `__m256d` wird für doubles verwendet und kann 4 solcher Zahlen speichern. Das *i* in `__m256i` weist auf einen Integervektor hin. Wie viele Bit die Integer intern haben, ist nicht festgelegt und wird erst anhand der auf dem Vektor ausgeführten Operationen entschieden. Man kann in diesem Datentyp also sowohl short Integer als auch long long Integer speichern und sowohl unsigned als auch signed. Abhängig von der Art der Integer ändert sich daher auch die Anzahl an Integern, die in den Vektor passen.

8. Vorlesung

1. Explain the naming conventions for intrinsic functions. (_<*vector_size*>_<*operation*>_<*suffix*>)

Der erste Namensteil gibt an, was für eine Art von Vektor zurückgegeben wird. Das könnte beispielsweise `mm256` für einen 256-Bit-Vektor oder `mm` für einen 128-Bit-Vektor sein.
Der mittlere Teil gibt an, was für eine Operation ausgeführt wird. Das kann beispielsweise so etwas wie `add`, `and` oder `extract` sein.
Der letzte Teil gibt den Typ der Eingabevektoren an. Floats werden dabei mit `ps` und Double mit `pd` abgekürzt. Für Integer unterscheidet sich diese Abkürzung abhängig von der Art des Integers. Aufgebaut ist sie nach folgendem Prinzip: Zunächst kommt `epi` für signed (vorzeichenbehaftete) Integer oder `epu` für unsigned (vorzeichenlos). Danach folgt die Bitgröße der Integer. `epi8` steht beispielsweise für signed 8-Bit-Integer und `epu32` für unsigned 32-Bit-Integer.

2. What do the metrics latency and throughput tell you about the performance of an intrinsic function?

Die Latenz gibt an, wie viele Taktzyklen benötigt werden, bis das Ergebnis einer Funktion verfügbar ist. Der *Throughput* hingegen gibt Auskunft darüber, wie viele Zyklen gewartet werden muss, bis die nächste intrinsische Funktion derselben Art gestartet werden kann. Bei einem *Throughput* von 0,5 können also 2 Funktionen derselben Art in einem Takt gestartet werden.
Bei niedrigem *Throughput* kann man eine hohe Latenz verbergen, indem man mehrere gleiche intrinsische Funktionen direkt hintereinander startet. Nach der entsprechenden Anzahl an Takten erhält man dann die Ergebnisse wie bei einer Pipeline eins nach dem. Man muss dabei jedoch darauf achten, dass die Eingaben der einzelnen Funktion nicht von der Ausgabe ihrer Vorgänger abhängen.

3. How do modern processors realize instruction-level parallelism?

Bei instruction-level Parallelismus können pro Taktzyklus mehrere Instruktionen gleichzeitig verschiedenen funktionalen Einheiten gestartet werden und laufen. Das Ganze geschieht auf nur einem einzelnen CPU Kern.
In der Execution Engine des CPU Kerns gibt es dafür einen Scheduler. Dieser teilt in jedem Takt die einzelnen ankommenden Mikrooperationen auf die vorhandenen Ports auf und gibt sie gleichzeitig an jeden von ihnen weiter. Jeder der Ports bedient dabei eine einzelne funktionale Einheit und kann nur bestimmte Operationen ausführen. So gibt es für seltenere Operationen wie Float Division nur einen Port, aber mehrere, die Vektoroperationen ausführen können.

4. How may loop unrolling affect the execution time of compiled code?

Loop unrolling ist eine Technik der Transformation von Schleifen, um sich instruction-level Parallelismus zu Nutze zu machen. Dazu wird die Schleife so angepasst, dass die Operationen innerhalb einer Iteration repliziert werden. Statt nur auf einem Element wird die Operation also in jeder Iteration auf k Elementen ausgeführt. Die Anzahl an Iterationen wird dazu ebenfalls entsprechend angepasst.
Durch das aufeinanderfolgende Ausführen gleicher Operationen kann, wie bereits oben in Aufgabe 2) beschrieben, durch den *Throughput* die Latenz einer Operation verborgen werden.
Loop Unrolling kann jedoch bei entsprechend großem k die Kompilierungszeit merklich verlängern. Des Weiteren benötigt es zusätzlichen Speicher im Instrunktionscache, wodurch andere wichtige Operationen aus dem Cache verdrängt werden können. Auch dadurch kann die Performance verschlechtert werden. Im Fall, dass man temporäre Variablen für die Instruktionen verwendet, werden außerdem entsprechend viele zusätzliche Register benötigt.

5. What does a high IPC value (instructions per cycle) mean in terms of the performance of an algorithm?

Um den IPC-Wert zu berechnen, wird die Anzahl an benötigten Instruktionen durch die Anzahl an zu durchlaufenden Zyklen geteilt. Ein hoher Wert deutet daher auf viel instruction-level Parallelismus und damit eine effiziente Ausnutzung der CPU hin. Dies bedeutet aber nicht automatisch, dass auch der Algorithmus selbst gut ist, sondern lediglich, dass er effizient implementiert ist.

9. Vorlesung

1. How do bandwidth-bound computations differ from compute-bound computations?

Bei *compute-bound* Berechnungen sind nur wenige Variablen involviert, welche sich meist in Registern befinden. Sie greifen somit nur sehr wenig bis gar nicht auf den Arbeitsspeicher zu. Folglich spielt dieser keine Rolle für die Performance. Stattdessen ist hier die CPU das Bottleneck und stellt somit das Geschwindigkeitslimit der Berechnung. Beim Optimieren der Performance von *compute-bound* Berechnungen spielen daher vor allem Vector Intrinsics eine große Rolle.
Bandwidth-bound Berechnungen auf der anderen Seite beruhen größtenteils auf Variablen. Ein Beispiel ist die Berechnung der Summer aller Elemente einer Matrix. Da die einzelnen Matrixelemente dabei nicht in den Registern liegen, erfolgt der Zugriff auf diese über den Speicher. Der Speicherzugriff stellt somit das Bottleneck der *bandwidth-bound* Berechnungen. Will man den Cache optimal ausnutzen, kann man hier viel optimieren, um das zu erreichen.

2. Explain why temporal locality and spatial locality can improve program performance.

Zeitliche Lokalität heißt, dass man eine Cache Line in einem bestimmten Zeitraum mehrmals benutzt. Sie wird erst dann wieder verworfen, wenn die in ihr gespeicherten Werte tatsächlich nicht mehr gebraucht werden.
Räumliche Lokalität hingegen bedeutet, dass nicht nur einer, sondern möglichst viele Werte einer Cache Line verwendet werden. Es werden also möglichst Elemente verwendet, die nah am zuvor geladenen Element gespeichert sind. Eine gute Vorgehensweise dafür ist *unit-stride* Speicherzugriff. Gleichzeitig wird dadurch auch die Vektorisierung des Programms vereinfacht.
Stellt man sicher, dass diese beiden Eigenschaften gut umgesetzt sind, so kann der Cache effizient genutzt werden. Sie stellen sicher, dass eine Cache Line möglichst lange verwendet wird und ersparen damit teure Speicherzugriffe.

3. What are the differences between data-oriented design and object-oriented design?

Beim objektorientierten Design orientiert man sich an der realen Welt und versucht, diese in Klassen bzw. Objekten abzubilden. Solche Objekte enthalten oft eine Vielzahl an Attributen unterschiedlicher Datentypen. Weil ein solches Objekt eine ganze Cache Line belegen kann, sind sie sehr Cache ineffizient. Will man beispielsweise auf ein Attribut eines Objekts zugreifen, muss dafür extra eine ganze Cache Line geladen werden. Alle Elemente, außer dem zu lesenden Attribut, werden jedoch nicht benötigt und im Anschluss direkt verworfen. Oft werden Objekte in *Array of Structures* gespeichert. Will man über einen solchen Array iterieren, um beispielsweise den Durchschnittswert eines

Attributes zu berechnen, stößt man durch die ineffiziente Cachenutzung auf Performanceprobleme. Dadurch sind objektorientiert designte Programme auch schwerer zu parallelisieren. Auch der Aufwand, um das Outputformat zu erzeugen, ist größer.
Beim *data-oriented Design* hingegen fokussiert man sich darauf, den Output so schnell und effizient wie möglich zu erzeugen. Man überlegt also, was für einen Output man erreichen will und wie man diesen mit dem geringst möglichen Aufwand generiert. Dazu erstellt man auch den Input direkt so, dass man dieses Ziel leicht erreichen kann. Statt als *Array of Structures* werden die Daten oft als *Structures of Arrays* angelegt. Das begünstigt auch das Iterieren über die Daten. Funktionen sind meist *general purpose* und werden auf großen Datenchunks angewandt. All das macht es leichter, den Code leichter zu parallelisieren und sorgt für eine gute CPU-Ausnutzung, wodurch diese Herangehensweise deutlich performanceorientierter als *object-oriented design* ist.

4. What are streaming stores?

Streaming Stores ermöglichen das Speichern (*store*) von Werten direkt im RAM, ohne sie zuvor im Cache abzulegen. Dadurch kann der Cache für wichtigere Daten freigehalten werden. *Streaming Stores* eignen sich daher gut, wenn der Code *memory-bound* und der Cache damit generell sehr stark ausgelastet ist. Außerdem sollten die zu speichernden Daten nicht direkt wieder benötigt werden. Wenn doch, ist es effizienter, sie direkt im Cache zu speichern, statt sie erst aus dem RAM laden zu müssen.Zu guter letzt dürfen die zu speichernden Daten vorher nicht gelesen worden sein, da sie in dem Fall bereits im Cache liegen würden (außer sie wurden mit *Streaming Loads* geladen). Nutzt man *Streaming Stores* ohne diese Aspekte zu berücksichtigen, kann sich die Performance deutlich verschlechtern, da das Lesen von Daten aus dem RAM deutlich teurer ist, als aus dem Cache.

5. Describe a typical cache hierarchy used in Intel CPUs.

Jeder Core der CPU besitzt einen eigenen L2-Cache und je einen L1-Cache für Daten und einen für Instruktionen. Außerdem hat die CPU einen coreübergreifenden L3-Cache. Im Gegensatz zum L1-Cache sind L2- und L3-Cache *unified* und speichern folglich sowohl Daten als auch Instruktionen.
Mit nur vier Taktzyklen ist die Latenz beim Laden aus dem L1-Cache am geringsten. Beim L2-Cache werden dafür bereits zehn Taktzyklen und beim L3-Cache ganze 40 bis 75 Zyklen gebraucht. Das Laden aus dem RAM dauert ca. 100 bis 200 Taktzyklen und ist damit deutlich langsamer.
Der L1-Cache besteht aus 64 *Hash-Buckets*, welche je maximal 8 Hashtabellen bzw. Cache Lines beinhalten. Jede davon hat eine Größe von 64 Byte, womit man auf eine Gesamtgröße von 32 KB für den L1-Cache kommt. Mit bis zu 8 Cache Lines auf 512 *Buckets* und einer Gesamtgröße von 256 KB ist der L2-Cache achtmal so groß. Der L3-Cache hat als einziger bis zu 16 Cache Lines pro *Bucket*. Mit 8192 *Buckets* und einer Gesamtgröße von 8 MB ist er der mit Abstand größte Cache. Nur die Blockgröße und damit die Größe der einzelnen Cache Lines ist aktuell mit 64 Byte überall gleich.

6. What are cache conflicts?

Da es sich bei Caches um Hardware Hashtabellen handelt, können Hashtabellenkonflikte auftreten, wenn mehrere Elemente auf dieselbe Position hashen. Beim Speichern einer neuen Cache Line wird diese meist per Modulo Operation auf eine Position im Cache gehasht. Gibt es dort keinen freien Platz, wird eine Cache Line gelöscht und mit der neuen ersetzt. Dies birgt jedoch einige Probleme mit sich. Einerseits können dadurch Daten entfernt werden, die im Anschluss wieder benötigt und dann neu geladen werden müssen. Außerdem kann es passieren, dass Cache Lines ungünstig auf die einzelnen *Buckets* gehasht werden. Wenn beispielsweise wiederholt Zweierpotenzen zur Hasherzeugung verwendet werden, kann es passieren, dass jedes Mal auf dasselbe *Bucket* gehasht wird. Selbst wenn es eventuell freie Plätze in anderen *Buckets* gäbe, würden diese also nicht genutzt werden. Stattdessen wird bei jedem Zugriff eine Cache Line weggeschmissen und ersetzt. Das kann die Performance stark negativ beeinflussen.

10. Vorlesung

1. Name and explain some useful compiler flags during development.

Ein praktisches Flag zum Debuggen ist `-Wall`. Mit diesem Flag werden alle Compiler-Warnungen aktiviert, die normalerweise nicht angezeigt werden würden. Darunter unter anderem implizite Typkonversionen, toter Code oder nicht-initialisierte Variablen.
Ein weiteres praktisches Flag ist `-g`, welches Debugging-Informationen direkt ins Programm inkludiert. Dadurch kann unter anderem der Code schrittweise ausgeführt oder Variablen überwacht, eingestellt und ausgewertet werden. Auf diese Weise kann man Programme effizienter debuggen.
Mit dem Flag `-fsanitize=address` können viele Speicherfehler gefunden werden, die sonst nur schwer zu finden sind und oft lange unerkannt bleiben. Dazu gehören unter anderem *out-of-bound*-Zugriffe, *use-after-free*, wo auf eine bereits freigegebene oder gelöschte Variable zugegriffen wird, oder *memory leaks*. Dieses Flag gibt es allerdings nur auf dem Clang- und GCC-Compiler.
Zu guter Letzt gibt es noch das `-fsanitize=undefined` Flag, welches undefiniertes Verhalten anzeigt. Wie auch das vorherige gibt es auch dieses Flag nur für Clang und GCC. Es findet unter anderem Integer Overflows, Division durch Null oder Typinkonsistenzen bei Funktionsaufrufen.

2. How could Intel oneAPI help you write better programs?

Intel oneAPI ist eine umfassende und einheitliches Entwicklungsplattform, die dabei hilft Programme zu schreiben, die die Intel-Hardwarearchitektur voll ausnutzen. Sie ermöglicht das Testen und Optimieren der Performance auf verschiedenen Hardwaretypen, ohne dass dazu der Code geändert werden muss. Dazu bietet oneAPI eine einheitliche Programmiersprache namens *Data Parallel C++* an. In dieser geschriebene Programme sind automatisch für verschiedene Hardwaretypen optimiert, was den Code vereinfacht und besser wartbar macht. Des Weiteren werden auch andere Programmiersprachen und Tools, wie beispielsweise C++, Python oder OpenCL unterstützt. Um das Optimieren zu vereinfachen, stellt oneAPI außerdem eine Reihe von Tools und Bibliotheken bereit. Darunter Debugging-Tools, Leistungsanalysetools und Bibliotheken zur Optimierung von Bild- und Signalverarbeitung.
Intel Inspector ist eine der vielen in Intel oneAPI integrierten Intel-Technologien. Es ist ein Tool zum Prüfen von *Threading* und *Memory Correctness*. Man kann es nutzen, um bspw. Deadlocks oder Data Races zu finden.
Eine weitere integrierten Intel-Technologie ist *VTune*. Dabei handelt es sich um einen Profiler, der einem unter anderem Informationen über die Zyklen-pro-Instruktion-Rate (CPI), die Nutzung von Threads im Programm, die *Cache-Misses*-Rate, die genutzte *memory bandwidth* und viele andere Aspekte gibt. Des Weiteren ermittelt er die *Hotspots* im Programm, also die Teile, die die meiste Zeit brauchen. Das können ganze Funktionen oder sogar einzelne Zeilen oder Assembler Instruktionen sein. Zu den *Hotspots* wird

zusätzlich angegeben, wie viel Zeit dort je gebraucht wird. Weiterhin kann man bspw. in der *Microarchitecture Analyse* prüfen, wie effizient die zugrundeliegende Hardware ausgenutzt wird. Die *Memory Access Analyse* zeigt einem an, wie viel der verfügbaren Bandbreite tatsächlich vom Programm genutzt wird. Anhand all dieser Informationen kann man herausfinden, an welchen Stellen ein Programm *Bottlenecks* bzw. Optimierungsbedarf hat und kann entsprechend handeln.

3. What can we learn from the following quote? "*Premature optimization is the root of all evil"* (Donald Knuth).

Das Zitat sagt aus, dass *Premature Optimization,* also Optimierungen, die man ohne weitere Informationen über die tatsächliche Performance des Codes macht, sich in vielen Fällen sogar negativ auf die Performance auswirkt. Nur weil man denkt, dass bspw. eine bestimmte Funktion langsam ist, sollte man nicht anfangen, sie auf alle möglichen Arten zu optimieren. Wenn die tatsächlichen Bottlenecks und andere Probleme nicht bekannt sind, ist die Wahrscheinlichkeit groß, dass man sie auf diese Art nicht fixen wird und eventuell sogar verschlimmert. Durch blindes "Optimieren" kann es außerdem passieren, dass man zu stark in das Programm eingreift und dadurch Abhängigkeiten oder die Korrektheit beeinträchtigt.
Beim Optimieren sollte man daher stets so vorgehen, dass man zuerst durch Messungen und *Profiling* die tatsächlichen Bottlenecks findet und analysiert. Erst dann sollte man mit dem Optimieren dieser beginnen. Der Grundsatz dabei lautet "*Keep it short and simple*". Man sollte also versuchen, so wenig wie möglich in das Programm einzugreifen und die Optimierungen auf einfache Dinge zu beschränken.

11. Vorlesung

1. What is Cython?

Cython ist eine Programmiersprache, die sowohl Python beinhaltet als auch Schnittstellen zu C und C++ bietet. Sie ist dabei ein Superset von Python. Das heißt, dass jeder valide Python-Code auch valider Cython-Code ist. Durch die Schnittstellen zu C/C++ hat man außerdem Zugang zu allen Features der beiden Sprachen, einschließlich der *Standard Template Library*.
Cython kann verwendet werden, um langsamen, rechenintensiven Python-Code performanter zu machen. Dadurch kann in einem Python-Programm fast C- oder C++-Geschwindigkeit erreicht werden. Dazu wird der in Cython geschriebene Code intern in C-Code übersetzt. Dieser wird anschließend in ein *Native Extension Modul* kompiliert, welches man dann wie ein normales Python-Modul importieren und im Python-Code verwenden kann. Da Cython Python sehr ähnlich und damit leichter zu lernen ist, kann man sich auf diese Weise das Schreiben der Module im komplizierten C ersparen. Man kann mit Cython allerdings auch direkt C- oder C++-Code wrappen, um diesen in Python zu verwenden.
Ein Nachteil von Cython ist jedoch, dass es nicht sehr portabel ist. Abhängig vom Betriebssystem wird mit Cython entweder eine .so- (Linux) oder eine .pyd-Datei (Windows) erstellt, welche je nur auf dem jeweiligen Betriebssystem laufen. Des Weiteren funktioniert Cython-Code meist nur auf der spezifischen Pythonversion, mit der er erstellt wurde. Diese Probleme fallen allerdings durch die Nutzung von *Anaconda* weg, da die Cython-Datei dort beim ersten Dateiaufruf entsprechend kompiliert wird.

2. Describe an approach how Python programs can be accelerated with the help of Cython.

Zunächst lokalisiert man mit einem Profiler, bspw. mit *cProfiler* oder *pyinstrument*, die Hotspots im Python-Code. Bei letzterem muss man lediglich in der Kommandozeile `pyinstrument filename.py` ausführen, um die Hotspots angezeigt zu bekommen.
Hat man die Bottlenecks gefunden, erstellt man eine neue Cython-Datei, in welche man die zu beschleunigenden Funktionen oder Codeabschnitte kopiert. Anschließend kann man diese bereits über die Kommandozeile mit `cyphonize -i -a filename.pyx` kompilieren. Dabei steht das `-i` für '*in place*' und sorgt dafür, dass die beim Kompilieren erstellten Dateien direkt neben der *Source*-Datei abgelegt werden. Mit dem `-a` wird zusätzlich eine Annotationsdatei erzeugt. An dieser kann man ablesen, wie viel des Cython-Codes noch in Python ist. Das gibt Hinweise darauf, an welchen Stellen man den Code überarbeiten sollte, um das Programm effizienter zu machen. Zu guter Letzt gibt man bei `filename.pyx` die zu kompilierende Datei an. Führt man das Kommando aus, werden eine C-Datei mit dem übersetzten Cython-Code, eine .so- bzw. .pyd-Datei mit der *Native Extension* und die durch das `-a` angegebene Reportdatei erstellt. Nun kann die neue Datei bereits in die ursprüngliche Python-Datei importiert und die kopierten

Funktionen aufgerufen werden. Das Programm sollte bereits etwas schneller sein, allerdings ist diese Verbesserung meist noch nicht ausreichend.
Im nächsten Schritt sollte man nun also den kopierten Code überarbeiten. Ein erster Schritt ist dabei das Ergänzen von Typannotationen zu allen Variablen. Das geht durch das Voranstellen von bspw. `cdef double` vor der Variablendefinition. Dabei muss man beachten, dass alle Variablen zum Beginn der Funktion definiert werden müssen. Bei späteren Variablendefinition mit `cdef` wird es beim Kompilieren Fehler geben. Hat man alle Typannotationen hinzugefügt und die Cython-Datei erneut kompiliert, sollte man bereits eine große Performanceverbesserung feststellen können.
Nun kann man mit dem Profiler erneut die Performance prüfen und erneut in der Annotationsdatei nachschauen, welche Zeilen noch hinterlegt sind und Optimierungsbedarf haben und diese iterativ beheben. Dabei kann man bei Bedarf auch C-Bibliotheken verwenden. Auch durch das Setzen von Cython-Compiler-Direktiven wie *language_level*, *boundscheck* oder *cdivision* zu Beginn des Cython-Codes kann dieser etwas effizienter werden. Allerdings fallen dadurch auch bestimmte Sicherheitschecks weg, wodurch er fehleranfälliger wird.

3. Describe two ways for compiling a .pyx Cython module.

Eine Möglichkeit ist das Kompilieren über die Kommandozeile mit `cyphonize -i -a filename.pyx`. Diese habe ich bereits oben in Aufgabe 2) ausführlich beschrieben, weshalb ich hier nicht weiter darauf eingehe.
Eine weitere Möglichkeit ist das Importieren und Installieren von *pyximport* durch das Einfügen der Zeilen `import pyximport` und `pyximport.install()` zu Beginn der Python-Datei. Dadurch wird das Programm bei Änderungen an importierten .pyx-Modulen automatisch neu kompiliert. Das ist vor allem praktisch, um das Programm transparenter zu machen und erspart den Umweg über die Kommandozeile.

4. Name and describe two compiler directives in Cython.

Mit dem Compiler-Direktive `language_level` gibt man an, welche Version von Python im Cython-Code verwendet wird. Das wäre bspw. Python 2.x bei `language_level = 2` oder Python 3.x bei `language_level = 3`. Die im Code verwendete Python-Syntax muss dem gewählten Level entsprechen.
Mit dem *boundscheck*-Flag kann man festlegen, ob *Boundschecks* durchgeführt werden sollen oder nicht. Ist man sich sicher, dass nirgends ein *Out-of-Bound*-Fehler auftreten kann, kann man `boundscheck = False` setzen.
cdivision ist ein weiteres Compiler-Direktive. Es bestimmt, ob bei Division statt dem Python-Divisionsverhalten das von C genutzt wird oder nicht. Wenn es auf True gesetzt wird, wird bei Division nicht auf mögliche Fehler geprüft. Null-Division resultiert dann in *undefined Behaviour*.

5. What is the difference between def, cdef and cpdef when declaring a Cython function?

Mit `def` definiert man normale Python-Funktionen. Man kann dabei keinen Rückgabetypen angeben, da stets ein Python-Objekt zurückgegeben wird.
`cdef`-Funktionen sind C-Funktionen und entsprechend schneller als `def`-Funktionen. Zusätzlich kann für diese Funktionen auch ein Rückgabetypen angegeben werden. Sie können allerdings nur in Cython verwendet und nicht in Python aufgerufen werden. Daher eignen sie sich vor allem für Hilfsfunktionen innerhalb des Cython-Programms.
Zu guter Letzt gibt es noch `cpdef`-Funktionen. Bei diesen werden im Hintergrund zwei Varianten der Funktion angelegt, eine Python- und eine Cython-Variante. Je nachdem, von wo aus die `cpdef`-Funktion aufgerufen wird, wird dann eine den beiden Varianten ausgeführt. Wie bei `cdef`-Funktionen kann man auch hier einen Rückgabedatentypen angeben, der bei Aufrufen aus Cython heraus verwendet wird.
Da `cdef`-Funktionen etwas effizienter sind als `cpdef`-Funktion, sollte man für Funktionen, die tatsächlich nur in Cython aufgerufen werden, lieber `cdef` verwenden.

6. What are typed memoryviews especially useful for in Cython?

Mit *Typed Memoryviews* kann man Python- oder *Numpy*-Arrays aus Python nach Cython übergeben. Das ist praktisch, da Berechnungen auf großen Arrays oder Matrizen häufig ein Bottleneck in Python-Programmen sind. *Typed Memoryviews* helfen daher, solche Berechnungen und die dabei verwendeten Daten nach Cython zu verschieben. Man verwendet sie, indem man den Funktionsparameter auf diese Art definiert:

```
def f(double[:,:] array):
```

In diesem Beispiel würde also ein 2-dimensionaler Array vom Datentyp double übergeben werden.

12. Vorlesung

1. What are extension types in the context of Python?

Extension Types sind in Cython geschriebene und kompilierte Klassen. Sie sind sehr ähnlich zu Python-Klassen, haben aber im Gegensatz zu diesen Typannotierungen und eventuell zusätzliche Keywords. Außerdem sind sie bei korrekter Implementierung deutlich schneller als die Python Alternativen. Nach dem Import können sie wie normale Python-Klassen im Programm verwendet werden. Man kann sie außerdem verwenden, um externen C- oder C++-Code in ihnen zu wrappen und das Programm so zu beschleunigen. Es ist auch möglich, Klassen direkt in C zu schreiben statt in Cython. Allerdings ist das deutlich komplizierter. Beispiele dafür sind die *built-in* Objekte *String*, *List* oder *Dictionary*.
Extension Types definiert man in Cython durch `cdef class MyClass`. Innerhalb dieser Klassen kann man dann Klassenattribute setzen. Diese müssen zur Verwendung auf klassenebene namentlich gesetzt werden. Des Weiteren muss man eins der Keywords `readonly` und `public` bei ihrer Definition angeben. Mit `readonly` gekennzeichnete Attribute können dabei von Python aus nur gelesen und nicht überschrieben werden. `public`-Attribute hingegen können von überall gelesen und geschrieben werden. Setzt man keines der beiden Keywords, kann nur innerhalb von Cython auf das Attribut zugegriffen werden. Ein Beispiel für einen *Extension Type* könnte so aussehen:

```
cdef class MyClass:
    cdef:
        readonly str myAttribut1
        public float myAttribut2
    def __init__(self, myAttribut1, myAttribut2):
        self.myAttribut1 = myAttribut1
        self.myAttribut2 = myAttribut2
```

2. How do extension types data fields in Cython differ from data fields in Python classes?

Wie in Aufgabe 1) bereits erwähnt, müssen Attribute bei *Extension Types* direkt nach der Klassendefinition explizit definiert werden. Es reicht also nicht, sie wie in Python nur indirekt im *Constructor* anzugeben. Außerdem werden den Attributen explizit Datentypen wie *char*, *int, float* oder auch *Extension Types* zugewiesen. Zusätzlich kann durch `readonly`, `public` oder das Weglassen einer solchen Kennzeichnung bestimmt werden, von wo und wie auf ein Attribut zugegriffen werden kann. In Python geht das nicht, stattdessen kann man jedes Attribut von überall aus lesen und schreiben. Ein weitere Unterschied ist, dass Attribute von *Extension Types* nicht wie in Python in nem *Dictionary* gehalten werden, sondern normal im Speicher abgelegt sind. Damit fällt der *Dictionary Lookup* weg, was die Performance deutlich verbessert. Der Feldzugriff ist damit bei *Extension Types* genauso effizient wie in nativem Code. Insgesamt sind *Extension Types* dadurch so effizient wie `struct` in C.

3. Give a simple description of how to wrap C / C++ code in Cython.

Zunächst schreibt man wie gewohnt seinen C- oder C++-Code, indem man Funktionsdeklarationen in eine Header-Datei und die Implementierungen selbst in eine -c bzw. .cpp-Datei packt. Als nächstes bindet man diese in eine Cython-Datei ein, welche das Interface bildet. Dabei definiert man zunächst, welche Funktionen und aus welcher Datei man verwenden möchte. Das sieht dann ungefähr wie folgt aus:

```
cdef extern from "myHeader.h":
      float myFunction(int n)
      ...
```

Die Funktionsliste kann dabei beliebig lang sein. Anschließend kommt das Interface, das man von Python aus benutzen wird. Das könnte bei diesem Beispiel wie folgt aussehen:

```
def myFunktionPy(int n):
      return myFunction(n)
```

Innerhalb der Cython-Funktion `myFunktionPy` wird die gewrappte C- bzw C++-Funktion `myFunction` aufgerufen und deren Ergebnis zurückgegeben.
Damit das Ganze funktioniert, muss man allerdings zusätzlich noch Compiler-Direktiven setzen. Die wichtigsten sind dabei `distutils: language = c++` und `distutils: sources = mySource.cpp`. Die erste sagt dem Compiler, in welcher Sprache die gewrappte Datei geschrieben ist. Das kann wie im Beispiel C++ oder auch C sein. Die zweite Direktive gibt den Namen der *Source*-Datei an, in welcher der zu verwendende Code implementiert ist. Zusätzlich kann man noch weitere Compiler-Direktiven angeben. Mit `distutils: extra_compile_args = -fopenmp -ffast-math` kann man beispielsweise zusätzliche Compilerflags setzen und mit `distutils: extra_link_args = -fopenmp` auf Libraries linken. Man muss jedoch beachten, dass diese Argumente vom Betriebssystem abhängig sind. Um das zu umgehen, kann man in einer gesonderten Setup-Datei über Alias die für das verwendete Betriebssystem benötigten Argumente setzen und übergeben.
Nachdem man das alles getan hat, kann man die Datei wie gewohnt in Python importieren und die implementierten Funktionen aufrufen.

13. Vorlesung

1. Delimit from each other the following SSD parts: Cells, Pages and Blocks.

Zellen sind die kleinsten Komponenten der SSD und speichern die einzelnen Bits. Dazu wird in ihnen eine bestimmte Anzahl an Elektronen gespeichert. Je nachdem, ob die Anzahl dieser einen bestimmten Schwellenwert über- oder unterschreitet, entscheidet sich der Wert der Zelle. Es gibt verschiedene Arten von Zellen, die je unterschiedlich viele Bits darstellen können. Die verlässlichsten, aber auch teuersten Zellen sind *Single Level Cells* (SLC). Diese haben nur einen Schwellenwert und können ein Bit, also 0 oder 1 speichern. Dadurch sind sie relativ robust gegenüber Schwankungen in der Zahl der Elektronen. Die nächste Variante sind *Multiple Level Cells* (MLC) mit 2 Bits und 4 Werten. Dann kommen *Triple Level Cells* (TLC) mit 3 Bits und 8 Werten und zum Schluss *Quad Level Cells* (QLC) mit 4 Bits und 16 Werten. Die QLCs können zwar am meisten Werte speichern, sind allerdings durch die vielen Schwellenwerte auch am unzuverlässigsten und gehen schnell kaputt. Auch die Performance ist deutlich schlechter, als bei SLCs oder MLCs. In der Regel basieren die Zellen auf *NAND*-Speicher.
Pages sind die Komponenten, in denen die Zellen organisiert sind. Sie sind meist 4 KB groß und sind die kleinsten Einheiten, auf denen bei Zellzugriffen operiert wird. Das heißt, zum Lesen einer einzelnen Zelle muss immer die ganze Page geladen werden, auf der sich diese befindet. Da Pages keine Möglichkeit zum Überschreiben einzelner Werte haben, muss zum Schreiben einer Zelle die gesamte Page, auf der sich diese befindet, auf eine neue Page kopiert und dabei der neue Wert gespeichert werden. Aus diesem Grund ist Lesezugriff auch schneller als Schreibzugriff. Die invalid gewordene alte Page wird anschließend als *stale* markiert und zu einem späteren Zeitpunkt *garbage collected*.
Pages wiederum sind in Blöcke (*Blocks*) organisiert, die meist 512 KB oder 1 MB groß sind. Sie bestehen also aus 128 bzw. 256 Pages. Nicht auf den einzelnen Pages, sondern nur hier können Löschoperationen durchgeführt werden. Hat man beispielsweise Schreiboperationen auf n Pages eines Blockes durchgeführt, so enthält dieser nun n als *stale* markierte, also invalide Pages. Um diesen Speicher wieder freizugeben und die n Pages zu löschen, werden die nicht-invaliden Pages in einen neuen Block kopiert. Anschließend wird der gesamte Block mit den n invaliden Pages gelöscht. Diese Operation nennt sich *Garbage Collection*. Es ist dabei anzumerken, dass das Löschen sehr teuer ist und eine große Latenz hat.

2. What is the purpose of garbage collection in SSDs?

Die *Garbage Collection* wird vom Controller gesteuert und bestimmt anhand eines Schwellenwerts, wann ein Block gelöscht wird. Dazu wird die Anzahl an invaliden Pages innerhalb der Blöcke überwacht. Überschreitet sie den Schwellenwert, wird der Block wie in Aufgabe 1) beschreiben gelöscht, um Platz für neue Pages zu machen. Je voller die SSD ist, desto häufiger muss dieser Prozess durchgeführt werden. Wird der Schwellenwert während einer Schreiboperationen überschritten, springt die *Garbage*

Collection an. Statt nur einer Page muss dann ein ganzer Block geschrieben werden. Dadurch steigt die *Write Amplification*, also der Faktor des Verhältnisses der tatsächlich geschriebenen Pages und der Anzahl an ursprünglich geplanten Schreiboperationen, bis auf sehr hohe Werte an. Da sich ein hoher *Write Amplification* Faktor negativ auf die Performance und die Lebenszeit der SSD auswirkt, wird versucht, möglichst effiziente *Garbage Collection* Algorithmen zu schreiben. Durch das Vermeiden unnötig vieler Schreiboperationen soll die *Write Amplification* möglichst gering gehalten werden. Um die Kosten der *Garbage Collection* zu verringern, haben SSDs oft mehr Speicher als ausgewiesen. Dieser Extraspeicher kann dann als Puffer benutzt werden. Außerdem versucht der Algorithmus, verwandte Daten möglichst in gleiche Blöcke zu legen, damit sie nah beieinander sind und möglichst schnell geladen werden können.

3. What is the purpose of wear leveling in SSDs?

Beim Löschen und Schreiben eines Blockes bleiben manchmal ein paar Elektronen in den Zellen übrig. Mit der Zeit sammeln sich diese an, wodurch die Zellen die Daten nicht mehr korrekt speichern können und kaputt gehen. Da SLC nur einen einzigen Schwellenwert haben, sind sie relativ robust gegen dieses Problem. QLC jedoch haben 15 Schwellenwerte, die relativ nah beieinander liegen. Aus diesem Grund haben diese Zellen nur eine geringe Lebenszeit.
Damit die Zellen trotzdem eine möglichst lange Lebensdauer haben, existiert *Wear Leveling*. Das ist ein vom Controller gesteuerter Prozess, der sicherstellt, dass alle Zellen gleichmäßig viel benutzt werden. Dazu werden unter anderem Daten, die nie oder kaum überschrieben werden, regelmäßig gelesen und auf Zellen überschrieben, die häufiger benutzt werden. Auch dieser Prozess erhöht jedoch die *Write Amplification* und wirkt sich dadurch negativ auf die Performance aus. Hier findet also ein Trade-of zwischen Performance und Lebensdauer statt.

4. Tell some interesting things about SSDs with an M.2 form factor.

SSDs mit M.2 Formfaktor gibt es in verschiedenen Ausführungen. Sie kommen mit *B-Key*, *M-Key* und *B-&-M-Key* vor. Man muss also genau darauf achten, welcher Key mit dem eigenen System kompatibel ist. Außerdem gibt es sowohl M.2 SSDs, die PCIe unterstützen, als auch welche, die nur SATA unterstützen. Während erstere über NVMe sehr schnell kommunizieren können, können letztere nur AHCI-Protokolle nutzen und sind daher deutlich langsamer. Man muss beim Kauf also genau darauf achten, was für Spezifikationen die SSD hat.

5. What influence do garbage collection and wear leveling have on write amplification of an SSD?

Beide Prozesse erhöhen die *Write Amplification*. Wird durch eine Schreiboperation die *Garbage Collection* aktiviert, so werden alle nicht-invaliden Pages eines Blockes in einen anderen Block umgeschrieben. Vor allem wenn der Speicher bereits sehr voll ist,

geschieht das sehr oft. Entsprechend hoch wird dadurch die *Write Amplification.* Ähnlich ist es bei *Wear Leveling*, das regelmäßig die Inhalte ganzer Blöcke verschiebt. Auch dafür werden sehr viele Schreiboperationen benötigt, was die *Write Amplification* erhöht.

6. Discuss three different recommendations for writing code for SSDs.

Der erste Tipp ist, zu vermeiden, viele kleine Dateien zu erzeugen. Der Grund ist, dass beim Lesen solcher Dateien eine ganze Page bzw. sehr viele Pages geladen werden müssen, nur um an wenige Informationen innerhalb dieser zu kommen. Das wirkt sich negativ auf die Performance aus, da unnötig viele Pages geladen werden müssen. Durch die hohe benötigte Bandbreite können außerdem andere Applikationen behindert werden. Obendrauf verkürzen die vielen Lese- und Schreibzugriffe die Lebenszeit der SSD. Man sollte daher versuchen, Datenstrukturen möglichst kompakt zu gestalten. Die Daten sollten möglichst nah beieinander liegen und im besten Fall alle in einer oder wenigen großen Dateien sein. Dadurch kann man durch einmal laden direkt alle Informationen bekommen, ohne die eben aufgezählten negativen Effekte.
Weiterhin sollte man darauf achten, die SSD nicht volllaufen zu lassen. Je weniger freier Speicher auf der SSD verfügbar ist, desto häufiger muss die *Garbage Collection* laufen und desto mehr Blöcke muss sie bewegen, um einen Block frei zu machen. Da die *Garbage Collection* bei vollem Speicher bei fast jeder Lese- oder Schreiboperation ausgelöst wird, läuft sie dann im Vordergrund statt im Hintergrund. Das wirkt sich sehr negativ auf die Performance aus und sorgt für eine sehr hohe *Write Amplification*. Ein Tipp, um das zu vermeiden, ist *Overprovisioning* durch manuelle Partitionierung des Speicher. So stellt man sicher, dass immer genug Speicher verfügbar ist, den die *Garbage Collection* nutzen kann.
Zu guter Letzt sollte man, wenn man viele kleine Lese- oder Schreiboperationen macht, mehrere Threads verwenden, statt nur einem. Die tatsächliche optimale Menge hängt von vielen Faktoren ab, aber 16 bis 64 Threads haben sich als eine gute Anzahl erwiesen. Die Operationen werden dann von den Threads gleichzeitig in die Queue der SSD gepackt. Auf diese Weise kann der interne Parallelismus der SSD voll ausgenutzt werden. Für Operationen auf großen Dateien hingegen reichen schon wenige Threads. Je nach äußeren Faktoren kann mit 2 bis 5 Threads bereits die optimale Leistung erreicht werden.

7. How could the CPU load for IO be reduced?

Eine Möglichkeit ist das Nutzen von asynchronen Routinen, statt blockierenden. Durch diese kann die CPU normal weiterarbeiten, während auf die Ergebnisse des IO-Calls gewartet wird. Sobald die angeforderten Daten dann vorliegen, werden sie weitergearbeitet.
Des Weiteren kann man bestimmte Flags setzen, um einen ähnlichen Effekt zu *Streaming Stores* bzw. *Streaming Loads* zu erzielen. So kann man das *OS Buffering* deaktivieren, wodurch die Applikation selbst das Cachen übernimmt und der RAM nicht unnötig belastet wird.

8. How could you solve problems that do not fit in DRAM without major code adjustments?

Wenn man es richtig angeht, kann man eine SSD als “RAM-Ersatz” verwenden und dadurch riesige Datensätze auf einer einzigen Workstation mit begrenztem RAM lösen. Die Latenzen, die man dabei normalerweise hat, können durch effizientes Pipelining versteckt werden. Dazu lädt man bereits den nächsten Datenchunk, während man auf dem aktuellen rechnet. So kann man direkt auf dem nächsten Datenchunk weiterrechnen, wenn die Berechnungen auf dem aktuellen abgeschlossen sind.
Wenn es einem nicht auf die Performance ankommt, kann man das auch mit Numpys *Memory Mapped Files* umsetzen. Diese können wie normale Arrays im Speicher benutzt werden. Die Komplexität dahinter wird versteckt und automatisch vom Betriebssystem gemacht. In Python könnte man einen solchen Array auf diese Art erstellen:

```
A = np.memmap('A.memmap', dtype='float64', mode='w+', shape=(200000, 10000))
```

Diese Zeile erstellt eine 200.000 mal 10.000 große float Matrix, die anschließend befüllt und für Rechnungen verwendet werden kann.