Praktikum 3 zu Parallele Programmierung

Fabian Czappa

TECHNISCHE UNIVERSITÄT DARMSTADT

Wintersemester 2023/2024
16. Januar 2024

Allgemeines

- Die Standardsprache im Quellcode ist Englisch, weswegen auch die Typen/Methoden so benannt sind.
- Es gibt Fußnoten, wenn Konzepte das erste Mal verwendet werden; dort können Sie sich weiter informieren.
- *Es gibt Hinweise (zu gutem Programmierstil oder anderem) in kursiver Schrift.*
- In diesem Praktikum wird auch CUDA benötigt. Stellen Sie sicher, dass Sie eine aktuelle Version davon lauffähig installiert bzw. Zugriff auf den Hochleistungsrechner der TU Darmstadt haben.

Bepunktung der Aufgaben

- Bei jeder Teilaufgabe stehen die erreichbaren Punkte dabei, welche die Anzahl an funktionalen Tests für die jeweige Teilaufgabe angibt.
- Für jeden funktionalen Test gilt: Sollte der Test fehlschlagen, nicht kompilieren oder länger als **eine Minute** brauchen, erhalten Sie keinen Punkt dafür. Sollte der Test innerhalb von **einer Minute** korrekt durchlaufen, erhalten Sie dafür einen Punkt.

Zusätzliche Hinweise

- Nennen sie **keine** Ihrer Dinge (z.B. Funktionen, Dateien, Klassen) mit `fabian` als Bestandteil vom Namen.
- Ihnen steht frei weitere Funktionalität zu implementieren, solange nicht explizit anders geregelt.
- Dieses Praktikum baut auf dem zweiten Praktikum auf. Wir haben einen Lösungsvorschlag in Moodle hochgeladen – passen Sie dahingehend Ihre Lösung an oder übernehmen Sie diesen (insbesondere neue Funktionalitäten).
- Achen Sie darauf, dass Sie nur standardkonformes C++/OpenMP/CUDA benutzen. Das Benutzen von compiler-/laufzeit-/betriebssystemspezifischen Erweiterungen und Funktionalitäten abseits der von uns vorgegebenen Dinge (Frameworks für die funktionalen Tests und die Laufzeittests, Benutzung von Typen wie `std::uint32_t` etc.) können zu Punktabzug führen.
- Benutzen Sie nicht mehr Speicher als gegeben für die CUDA-Kernel, da die Tests keine Möglichkeit haben, Ihnen diesen korrekt zur Verfügung zu stellen.
- Bitte beachten Sie, dass die Tutor:innen keine bindenden Aussagen treffen.

Nachname, Vorname: _____ Matrikelnummer: ⌴⌴⌴⌴⌴⌴⌴

Aufgabe 1: Verschlüsselung anhand von Schemata (Gesamt: 5 Punkte)

Die hier geforderten Funktionen werden in genau dieser Form getestet; eine solche ist durch die `Teletype` Schrift gekennzeichnet.

1a) Schritt und Schema der Verschlüsselung (1 Punkt)

Stellen Sie in der Datei `Algorithm.h` das `enum`[1] `EncryptionStep` bereit, welches genau die Werte `E`, `D`, `K` und `T` annehmen kann. Definieren Sie außerdem den Typ `EncryptionScheme`, welcher gleich `std::array<EncryptionStep, 16>` ist.

1b) Kodierung eines Schemas (1 Punkt)

Stellen Sie in der Datei `Algorithm.h` die Funktion `encode` bereit, die ein `EncryptionScheme` als Parameter nimmt und ein `std::uint64_t` zurück gibt. Die Kodierung erfolgt so:
Übersetzen Sie die `EncryptionStep` zuerst in Bits: E $\mapsto$ 00, D $\mapsto$ 01, K $\mapsto$ 10, T $\mapsto$ 11.
Ordnen Sie die Bits derart an, dass diese geordnet im `std::uint64_t` stehen; der erste Schritt/der niedrigste Wert im Schema als niedrigwertigste Gruppe von zwei Bits, etc. (dies verbraucht die unteren 32 Bit). Replizieren Sie die 32 unteren Bits in die 32 oberen Bits. In einem kleineren Beispiel sieht es so aus:
`<scheme>[0] = E`, `<scheme>[1] = D`, `<scheme>[2] = E`, `<scheme>[3] = T`, ...
`<code> = X - X - X - 11000100 - X - X - X - 11000100`

1c) Dekodierung eines Schemas (2 Punkte)

Stellen Sie in der Datei `Algorithm.h` die Funktion `decode` bereit, die ein `std::uint64_t` als Parameter nimmt und ein `EncryptionScheme` zurück gibt. Diese soll das Inverse zu `encode` sein, d.h., für alle möglichen `EncryptionScheme e` soll gelten: `e == decode(encode(e))`.
Implementieren Sie die Funktionalität. Außerdem: Sollte der zu dekodierende Wert nicht der Form von `encode` entsprechen (niedrigwertige 32 Bit nicht gleich den hochwertigen 32 Bit), werfen Sie eine `std::exception`.

1d) Verarbeitung eines Schemas (1 Punkt)

Stellen Sie in der Datei `Algorithm.h` die Funktion `perform_scheme` bereit, welche drei Parameter nimmt: ein `BitmapImage`, ein `Key::key_type` und ein `EncryptionScheme`. Sie gibt ein `BitmapImage` zurück.
In der Methode iterieren Sie über das gegebene Schema (von Stelle 0 bis Stelle 15) und machen je nach Schritt eins der folgenden Dinge:

- Schritt ist `E`: Verschlüsseln Sie die aktuelle Version des Bildes mit `FES::encrypt`, nutzen Sie dafür den aktuellen Schlüssel.
- Schritt ist `D`: Entschlüsseln Sie die aktuelle Version des Bildes mit `FES::decrypt`, nutzen Sie dafür den aktuellen Schlüssel.
- Schritt ist `T`: Transponieren Sie das aktuelle Bild.
- Schritt ist `K`: Berechnen Sie den aktuellen Schlüssel mit `Key::produce_new_key` neu.

Sie ersetzen also pro Schritt entweder das aktuelle Bild oder den aktuellen Schlüssel.
Einige Schemata sind "unnütz", z.B. K–K–E–E–D–D–T–T–K–K–E–E–D–D–T–T. Nutzen Sie diese zur Korrektheitsprüfung.
Sie können annehmen, dass die Bilder (auch transponiert) die Anforderungen für Ver- und Entschlüsselung erfüllen, also Breite und Höhe Vielfache von 48 sind.

[1] `https://en.cppreference.com/w/cpp/language/enum`

Aufgabe 2: Portierung Entfärben (Gesamt: 5 Punkte)

Die hier geforderten Funktionen werden in genau dieser Form getestet; eine solche ist durch die `Teletype` Schrift gekennzeichnet.
Wir folgen hier der Konvention, dass `.cuh` Dateien als Header für CUDA gelten und `.cu` Dateien als Objektdateien für CUDA.

2a) Berechnung der Blockgrößen (1 Punkt)

Stellen Sie in der Datei `common.cuh` die Funktion `divup` bereit, welche zwei Parameter vom Typ `unsigned int` akzeptiert und ein `unsigned int` zurück gibt. Sie dürfen annehmen, dass die übergebenen Werte immer größer als 0 sind. Dabei soll gelten: wenn `r = divup(n, d)`, dann gilt $r \times d \geq n > (r-1) \times d$
Sie können die Funktion benutzen, um die Blockgrößen für CUDA-Kernel zu berechnen. Als ersten Parameter übergeben Sie die Anzahl Threads, als zweiten Parameter die Anzahl Threads pro Block (ggf. pro Dimension).
Die Tests benutzen `divup` an diversen Stellen. Stellen Sie sicher, dass Sie die Funktion korrekt implementiert haben, um Fehler in anderen Tests zu vermeiden.

2b) CUDA-Kernel für Entfärben (2 Punkte)

In der Datei `image.cuh` haben wir die Deklaration des CUDA-Kernels `grayscale_kernel` gestellt, ändern Sie diese nicht. Implementieren Sie den Kernel in der Datei `image.cu`. Dieser soll die Funktionalität von `BitmapImage::get_grayscale` auf der Grafikkarte nachbauen. Die Parameter haben folgende Bedeutung:

- `width`: Die Breite des Bildes, größer als 0.
- `height`: Die Höhe des Bildes, größer als 0.
- `input`: Die Pixel des (bunten) Eingabebildes (dicht gepackt). Sie können mindestens `input[0]` bis `input[width * height - 1]` lesen.
- `output`: Der Speicherbereich des entfärbten Ausgabebildes (dicht gepackt). Sie können mindestens `output[0]` bis `output[width * height - 1]` lesen und schreiben. Schreiben Sie den berechneten Wert von `input[k]` nach `output[k]`.

Gehen Sie davon aus, dass der Kernel mit $(b_x, b_y, 1)$ Threadblöcken der Größe $(t_x, t_y, 1)$ aufgerufen wird, sodass gilt: $b_x \times t_x \geq$ `width` und $b_y \times t_y \geq$ `height`.
Sollte Ihre Implementierung *shared memory* brauchen, ändern Sie die Definition von `GRAYSCALE_SHARED_MEM`.
Da die Pixel nicht voneinander abhängen, ist es egal, in welcher Richtung die Werte des rechteckigen Bildes linearisiert sind. CPUs und GPUs rechnen Fließkommaoperationen unterschiedlich, sodass Sie nicht erwarten müssen, dass genau das gleiche Ergebnis bei beiden Varianten raus kommt.
In unserer Referenzimplementierung unterscheiden sich die berechneten Grauwerte zwischen der CPU- und der GPU-Variante jeweils um maximal 1 pro Pixel. Dies postulieren wir auch für die Tests.

2c) Aufruf des CUDA-Kernels für Entfärben (2 Punkte)

In der Datei `image.cuh` haben wir die Deklaration von `get_grayscale_cuda` gestellt, ändern Sie diese nicht. Implementieren Sie die Funktion in der Datei `image.cu`. Diese soll mit dem übergebenen Bild den CUDA-Kernel `grayscale_kernel` aufrufen und das berechnete, entfärbte Bild zurück geben.
Sie können dafür in der Vorlage `BitmapImage::get_data()` benutzen; falls Sie Ihre eigene Klasse benutzen, müssen Sie ggf. mehrere Stellen anpassen.

Aufgabe 3: Hashen (Gesamt: 10 Punkte)

Die hier geforderten Funktionen werden in genau dieser Form getestet; eine solche ist durch die `Teletype` Schrift gekennzeichnet.
Wir folgen hier der Konvention, dass `.cuh` Dateien als Header für CUDA gelten und `.cu` Dateien als Objektdateien für CUDA.

3a) CUDA-Kernel für Hashen (2 Punkte)

In der Datei `encryption.cuh` haben wir die Deklaration des CUDA-Kernels `hash` gestellt, ändern Sie diese nicht. Implementieren Sie den Kernel in der Datei `encryption.cu`. Dieser soll die Funktionalität von `Hash::hash(std::uint64_t)` für mehrere Werte nachbauen. Die Parameter haben folgende Bedeutung:

- `length`: Die Anzahl an Werten, größer als 0.
- `values`: Die Werte, welche gehasht werden sollen (dicht gepackt). Sie können mindestens `values[0]` bis `values[length - 1]` lesen.
- `hashes`: Der Speicherbereich der berechneten Hashes (dicht gepackt). Sie können mindestens `hashes[0]` bis `hashes[length - 1]` lesen und schreiben. Schreiben Sie den berechneten Wert von `values[k]` nach `hashes[k]`.

Gehen Sie davon aus, dass der Kernel mit $(b_x, 1, 1)$ Threadblöcken der Größe $(t_x, 1, 1)$ aufgerufen wird, sodass gilt: $b_x \times t_x \geq$ `length`. Sollte Ihre Implementierung *shared memory* brauchen, ändern Sie die Definition von `HASH_SHARED_MEM`.

3b) Flacher CUDA-Kernel für Hashen (2 Punkte)

In der Datei `encryption.cuh` haben wir die Deklaration des CUDA-Kernels `flat_hash` gestellt, ändern Sie diese nicht. Implementieren Sie den Kernel in der Datei `encryption.cu`. Dieser soll die Funktionalität von `Hash::hash(std::uint64_t)` für mehrere Werte nachbauen. Die Parameter haben folgende Bedeutung:

- `length`: Die Anzahl an Werten, größer als 0.
- `values`: Die Werte, welche gehasht werden sollen (dicht gepackt). Sie können mindestens `values[0]` bis `values[length - 1]` lesen.
- `hashes`: Der Speicherbereich der berechneten Hashes (dicht gepackt). Sie können mindestens `hashes[0]` bis `hashes[length - 1]` lesen und schreiben. Schreiben Sie den berechneten Wert von `values[k]` nach `hashes[k]`.

Gehen Sie davon aus, dass der Kernel mit $(1, 1, 1)$ Threadblöcken der Größe $(t_x, 1, 1)$ aufgerufen wird, wobei t_x und `length` beliebig, aber größer als 0, sind. Sollte Ihre Implementierung *shared memory* brauchen, ändern Sie die Definition von `FLAT_HASH_SHARED_MEM`.

3c) Finden von Hashs (2 Punkte)

In der Datei `encryption.cuh` haben wir die Deklaration des CUDA-Kernels `find_hash` gestellt, ändern Sie diese nicht. Implementieren Sie den Kernel in der Datei `encryption.cu`. Dieser soll in einer übergebenen Liste von Hashs einen bestimmten suchen, und die Indizes zurück geben, an welchen der gesuchte Hash steht. Die Parameter haben folgende Bedeutung:

- `length`: Die Anzahl an Hashs, größer als 0.
- `searched_hash`: Der gesuchte Hash.
- `hashes`: Die Hashs, in welches der übergebene Wert gesucht werden soll. Sie können mindestens `hashes[0]` bis `hashes[length - 1]` lesen.

- `indices`: Der Speicherbereich für die Indizes, an welchen (bzgl. `hashes`) der übergebene Wert steht. Sie können mindestens `indices[0]` bis `indices[length - 1]` lesen und schreiben. Ist komplett mit 0 initialisiert.
- `ptr`: Ein Speicherbereich, welchen Sie möglicherweise benutzen möchten. Sie können mindestens `ptr[0]` lesen und schreiben. Ist komplett mit 0 initialisiert.

`indices` soll nach Ausführung des Kernels die entsprechenden Indizes enthalten. Diese sollen alle am Anfang des Speicherbereichs sein, alle nicht gebrauchten Werte sollen 0 sein. Die Reihenfolge der gefundenen Indizes ist egal. Es ist möglich, dass keiner/alle/mehrere der übergebenen Hashs gleich dem gesuchten sind.
Gehen Sie davon aus, dass der Kernel mit $(b_x, 1, 1)$ Threadblöcken der Größe $(t_x, 1, 1)$ aufgerufen wird, sodass gilt: $b_x \times t_x \geq$ `length`. Sollte Ihre Implementierung *shared memory* brauchen, ändern Sie die Definition von `FIND_HASH_SHARED_MEM`.

3d) Hashen von kodierten Schemata (2 Punkte)

In der Datei `encryption.cuh` haben wir die Deklaration des CUDA-Kernels `hash_schemes` gestellt, ändern Sie diese nicht. Implementieren Sie den Kernel in der Datei `encryption.cu`. Dieser soll für eine gegebene Anzahl an kodierten Schemata (vgl. 1b) deren Hashs berechnen. Die Parameter haben folgende Bedeutung:

- `length`: Die Anzahl an (kodierten) Schemata, welche gehasht werden sollen, größer als 0.
- `hashes`: Der Speicherbereich für die Hashs der kodierten Schemata. Sie können mindestens `hashes[0]` bis `hashes[length - 1]` lesen und schreiben.

Enumerieren Sie die Schemata dabei wie die Ordnung der Zahlen es vorgibt, also:
`hashes[0] = hash(encode(...-E-E)) = hash(...-00-00)`,
`hashes[1] = hash(encode(...-E-D)) = hash(...-00-01)`,
`hashes[2] = hash(encode(...-E-K)) = hash(...-00-10)`,
`hashes[3] = hash(encode(...-E-T)) = hash(...-00-11)`,
`hashes[4] = hash(encode(...-D-E)) = hash(...-01-00)`,
...
Gehen Sie davon aus, dass der Kernel mit $(b_x, 1, 1)$ Threadblöcken der Größe $(t_x, 1, 1)$ aufgerufen wird, wobei aber b_x, t_x und `length` beliebig, aber größer als 0, sind. Sollte Ihre Implementierung *shared memory* brauchen, ändern Sie die Definition von `HASH_SCHEMES_SHARED_MEM`.

3e) Wiederfinden des Bildes (2 Punkte)

Da FES alleine relativ schwach ist, haben wir alle Urlaubsbilder nach einem bestimmten Schema verschlüsselt. Dieses haben wir leider verloren, wissen aber noch folgende Dinge darüber:

- Es waren 10 zufällig ausgewählte Schritte (also E, D, K oder T), gefolgt von 6-maligem Verschlüsseln.
- Der Hash des (kodierten) Schemas ist: `9 810 482 633 726 283 944` (dieser ist eindeutig).
- Der erste Schlüssel war `Key::get_standard_key()`.

Stellen Sie in der Datei `Algorithm.h` die Funktion `retrieve_scheme` bereit, welche ein `EncryptionScheme` zurück gibt (falls mehrere möglich sind, suchen Sie davon eins aus) und ein `std::uint64_t` als Parameter nimmt (den Hash der Kodierung eines Schemas, hat an Stellen 10 bis 15 ein `E`). Dabei soll für
`s = retrieve_scheme(c)` gelten: `hash(encode(s)) = c`.
Es sind also $4^{10} = 2^{20} = 1024 \times 1024$ *Schemata möglich.*
Im Ordner `output/` *liegt eins der 'verlorenen' Bilder. Wenn Sie das Schema gefunden haben, können Sie dieses 'retten', indem Sie das Schema rückwärts ablaufen lassen, dabei aber auf den Schlüssel achten. Dies hat nichts mit der Bewertung zu tun und dient nur Ihrer Kontrolle.*

Nachname, Vorname: _____ Matrikelnummer: ␣␣␣␣␣␣␣␣

Hinweise zur Abgabe

Hier ein paar zusätzliche Formalitäten zu Ihrer Abgabe. Falls Sie diese nicht beachten, ist es möglich, dass Sie keine Punkte für eine oder alle Aufgaben erhalten.

Angabe der Autorschaft

Geben Sie in der Datei /source/authors.h an, wer von Ihnen an welcher Teilaufgabe mitgearbeitet hat. Mehrere Namen pro Teilaufgabe sind ok, trennen Sie diese z.B. mit Komma. Wir fordern eine Teilnahme von allen! Sollten Sie nicht an dem Praktikum mitgearbeitet haben (basierend auf den Angaben), erhalten Sie 0 Punkte auf das ganze Praktikum.

Hochladen des Quellcodes

Zippen Sie folgende Dateien und laden Sie das zip-Archiv in Moodle hoch:

- Den Ordner /cmake/
- Den Ordner /source/
- Die Datei CMakeLists.txt

Sie sind alle für die korrekte Abgabe verantwortlich. Zu spät eingereichte Dateien, Dateien mit fehlenden Lösungen, etc., liegen in allein Ihrer Verantwortung.

Hinweise zum Lichtenberg

Der Lichtenberg-Hochleistungsrechner ist aufgeteilt in sogenannte Loginknoten und Rechenknoten, wobei sich alle das gleiche Dateisystem teilen. Erstere sind mit **ssh** und **scp** zu erreichen und werden benutzt um Rechenaufgaben für letztere zu kreieren. Wenn Sie eine sinnvolle Linux-Distribution verwenden, kann /bin/bash nativ **ssh** und Sie können das Dateisystem des Lichtenberg direkt in Ihrem Explorer (z.B. Nemo) einbinden. Wenn Sie Windows benutzen, empfiehlt sich das Terminal bzw. PuTTy für **ssh** und WinSCP für **scp**.

Sobald Sie eingeloggt sind, sollten Sie zusätzliche Softwarepakete laden. Das erreichen Sie beispielsweise mit **module load git/2.40.0 cmake/3.26.1 gcc/11.2.0 cuda/11.8 boost/1.81.0**. Kopieren Sie die Dateien auf den Hochleistungsrechner und kompilieren Sie dort das Programm.

Wenn Sie eine Rechenaufgabe lösen möchten, müssen Sie dafür SLURM benutzen. Effektiv führen Sie den Befehl **sbatch <filename>** aus, wobei <filename> hier eine Datei ist, die, sobald Ihr Programm Zeit und Resourcen zugeteilt bekommen hat, auf den Rechenknoten ausgeführt wird. Ein beispielhaftes Skript sieht so aus:

```
#!/bin/bash

#SBATCH -J parprog
#SBATCH -e /home/kurse/kurs00072/<TUID>/stderr/stderr.parprog.%j.txt
#SBATCH -o /home/kurse/kurs00072/<TUID>/stdout/stdout.parprog.%j.txt
#SBATCH -C avx512
#SBATCH -n 1
#SBATCH --mem-per-cpu=1024
#SBATCH --time=5
#SBATCH --cpus-per-task=1
#SBATCH -A kurs00072
#SBATCH -p kurs00072
#SBATCH --reservation=kurs00072

module load git/2.40.0 cmake/3.26.1 gcc/11.2.0 cuda/11.8 boost/1.81.0

echo "This is job $SLURM_JOB_ID"
```

Der Kurs ist ab dem 01. November 2023 freigeschaltet. Alle Informationen erhalten Sie auch hier: `https://www.hrz.tu-darmstadt.de/hlr/nutzung_hlr/zugang_hlr/loginknoten_hlr/index.de.jsp`