Fachpraktikum Parallele Programmierung, Wintersemester 2024/25

3. Aufgabe: Histogramm

*Johannes Becker, Norbert Baumstark, Stefan Butz*

**Aufgabenstellung**

Es sollten Routinen zur Erstellung eines Histogramms auf einer GPU implementiert und hinsichtlich ihrer Laufzeit untersucht werden. Dabei sollte gezählt werden, wie oft einzelne Bytewerte in einem Array von Bytes vorkommen. Das Hochzählen der Anzahlen in den Bins des Histogramms sollte durch atomare Operationen erfolgen. Es waren zwei Varianten zu untersuchen: zum einen das Hochzählen direkt im globalen Speicher; zum anderen sollten zunächst mehrere Instanzen der Bins im Shared Memory angelegt und diese erst abschließend zu einem einzigen Histogramm akkumuliert werden.

Die Eingabedaten waren nach Annahme Bytewerte zwischen 1 und 128. Es waren zwei Varianten des Histogramms zu erstellen, eine mit 128 Bins für die einzelnen Bytewerte sowie eine weitere mit jeweils einem Bin für die Buchstaben ‚A‘ bzw. ‚a‘ bis ‚Z‘ bzw. ‚z‘ und einem weiteren Bin für alle anderen Zeichen, d.h. insgesamt 27 Bins.

**Verzeichnisstruktur**

Auf der obersten Verzeichnisebene befinden sich zwei Dateien, nämlich ein Makefile sowie ein Python-Skript perform\_measurements.py, welches alle Messungen ausführt.

Es gibt folgende Unterverzeichnisse:

* src/ enthält den Quellcode.
* analysis/: enthält ein Jupyter-Notebook zur Auswertung der Messergebnisse.
* doc/ enthält diese Dokumentation.
* input\_data/ enthält die Testdaten, die der Aufgabenstellung beigefügt waren.
* nvidia/ enthält das mit den CUDA-Entwicklungstools mitgelieferte Beispielprogramm deviceQuery, welches Geräteinformationen über die GPU ausgibt.
* bin/ wird ggf. von make angelegt und dient als Zielverzeichnis für die Objektdateien sowie die ausführbaren Dateien deviceQuery und histogram.
* measurements/ wird ggf. von perform\_measurements.py angelegt und dient der Ablage der Geräteinformationen und Messergebnisse.

In den abgegebenen Dateien sind die Messergebnisse enthalten, die dieser Dokumentation zugrunde liegen.

**Aufruf**

Nach Erstellen der ausführbaren Dateien durch

make

startet man durch

./perform\_measurements.py

die Messungen. Gegebenfalls sind die Parameter im Makefile auf die verwendete Architektur und im Python-Skript der Pfad zum Python-Interpreter anzupassen.

Das Python-Skript ermittelt zunächst die Geräteinformationen durch Aufruf von bin/deviceQuery und ruft dann bin/histogram mit verschiedenen Kommandozeilenargumenten auf, um verschiedene Szenarien zu messen. Insbesondere wird bin/histogram angewiesen, pro Szenario und Kernel 100 Messungen vorzunehmen. Neben den der Aufgabenshhistellung beiliegenden Testdaten umfassen die gemessenen Datengrössen den Bereich von Bytes bis GiB in Zweierpotenzen. Die Ausführungsdauer des Python-Skripts ist also erheblich. Sollen aus Gründen der Zeitersparnis weniger als 100 Durchläufe je Szenario erfolgen, kann die Konstante N\_RUNS im Python-Skript entsprechend angepasst werden.

Die Geräteinformationen werden als Textdatei, die Messergebnisse im JSON-Format in einem Unterverzeichnis von measurements/ gespeichert. Das angelegte Unterverzeichnis trägt einen Zeitstempel als Namen.

Einzelne (wiederholte) Messungen können direkt durch Aufruf des Binarys bin/histogram mit entsprechenden Kommandozeilenargumenten vorgenommen werden.

Beispiele:

bin/histogram ./input\_data/test.txt

führt mit jedem der vier Kernels eine Messung für die Beispieldatei aus, wobei 128 Bins verwendet werden.

bin/histogram -- 10500 asl 7

führt für pseudo-zufällige Daten der Grösse 10500 Bytes mit den Kernels histogram\_atomic\_private (‘a’) und histogram\_atomic\_private\_stride (‘s’) jeweils 7 Messungen aus, wobei 27 Bins (‘l’) verwendet werden.

Die Usage-Message, die nach Aufruf von bin/histogram ohne Argumente erscheint (sie findet sich auch am Anfang von src/histogram.cu), erklärt die Bedeutung der möglichen Kommandozeilenargumente.

**Der Programmcode – histogram.cu**

Kernels

Die Kernels sind als Template-Funktionen ausgestaltet. Für den Template-Parameter Mapping kann eine struct übergeben werden, welche die Zuordnung von Zeichencodes zu Bins definiert. Wir verwenden zwei Mappings, entsprechend den beiden Aufgabenteilen.

Für Aufgabenteil a werden die Zeichencodes 1 bis 128 den Bins 0 bis 127 zugeordnet:

struct Mapping128 {

constexpr static size\_t numBins = 128;

constexpr static \_\_host\_\_ \_\_device\_\_ unsigned char map(

unsigned char c

) {

return (c - 1u) & 0x7f;

}

};

Für Aufgabenteil b werden die Buchstaben ‘A’/’a’ bis ‘Z’/’z’ (Zeichencodes 65 bzw. 90 bis 97 bzw. 122) den Bins 1 bis 26 zugeordnet; alle übrigen Zeichencodes dem Bin 0:

struct MappingLetter {

constexpr static size\_t numBins = 27;

constexpr static \_\_host\_\_ \_\_device\_\_ unsigned char map(

unsigned char c

) {

c = (c & 0xdf) - 64u;

return c & (0u - (c <= 26u));

}

};

Es werden vier Kernels untersucht.

atomic\_global

«Baseline» ist ein einfacher Kernel, bei dem jeder Thread genau ein Zeichen bearbeitet und dann den entsprechenden Bin im globalen Speicher hochzählt:

template<typename Mapping>

\_\_global\_\_ void histogram\_kernel\_atomic\_global(

unsigned char \* input, BinType \* bins, size\_t numElements

) {

unsigned int idx = blockIdx.x \* blockDim.x + threadIdx.x;

if (idx >= numElements) return;

unsigned char c = input[idx];

atomicAdd(&bins[Mapping::map(c)], 1);

}

Beim Aufruf des Kernels wird berechnet, wie viele Blöcke benötigt werden, in Abhängigkeit von der Eingabegrösse:

constexpr size\_t nThreadsPerBlock = 128;

// …

dim3 dimGrid(

(numElements + nThreadsPerBlock - 1) / nThreadsPerBlock, 1, 1

);

dim3 dimBlock(nThreadsPerBlock, 1, 1);

histogram\_kernel\_atomic\_global<Mapping> <<<dimGrid, dimBlock>>> (

input, bins, numElements

);

atomic\_private

Der zweite Kernel unterscheidet sich vom ersten nun in der Verwendung von Shared Memory. Jeder Block erhält eine private Instanz des Bins-Arrays im Shared Memory. Diese Instanzen werden abschliessend im globalen Speicher aggregiert.

template <typename Mapping>

\_\_global\_\_ void histogram\_kernel\_atomic\_private(

unsigned char \* input, BinType \* bins, size\_t numElements

) {

unsigned int idx = blockIdx.x \* blockDim.x + threadIdx.x;

\_\_shared\_\_ BinType sBins[Mapping::numBins \* sizeof(BinType)];

for (

unsigned int t = threadIdx.x; t < Mapping::numBins;

t += blockDim.x

) {

sBins[t] = 0;

}

\_\_syncthreads();

if (idx < numElements) {

unsigned char c = input[idx];

atomicAdd(&sBins[Mapping::map(c)], 1);

}

\_\_syncthreads();

for (

unsigned int t = threadIdx.x; t < Mapping::numBins;

t += blockDim.x

) {

atomicAdd(&bins[t], sBins[t]);

}

}

Der Aufruf erfolgt entsprechend wie beim ersten Kernel – es werden so viele Blöcke angefordert, wie für die Eingabe benötigt werden.

atomic\_private\_stride

Beim dritten Kernel ist die Anzahl der Threads fest, unabhängig von der Eingabegrösse. Pro Schritt («stride») wird ein Block von nebeneinanderliegenden Zeichen bearbeitet, bis die Eingabe abgearbeitet ist. Es wird wieder mit privaten Instanzen des Bins-Arrays für die einzelnen Blöcke gearbeitet.

template <typename Mapping>

\_\_global\_\_ void histogram\_kernel\_atomic\_private\_stride(

unsigned char \* input, BinType \* bins, size\_t numElements

) {

unsigned int idx = blockIdx.x \* blockDim.x + threadIdx.x;

// initialisiere Array im shared memory mit 0

\_\_shared\_\_ BinType sBins[Mapping::numBins \* sizeof(BinType)];

for (

unsigned int t = threadIdx.x; t < Mapping::numBins;

t += blockDim.x

) {

sBins[t] = 0;

}

\_\_syncthreads();

{

// baseLimit ist die kleinste Zeichenposition, ab der ein

// ab baseLimit beginnender Stride genau am letzten Zeichen

// des Inputs endet oder über den Input hinausragt.

// Im Prinzip wäre das die letzte Iteration der Schleife. Da

// allerdings hier durch eine if-Abfrage geprüft werden

// müsste, ob idx noch innerhalb des Inputs liegt, spart es

// etwas Zeit, den letzten Stride separat zu behandeln.

int stride = blockDim.x \* gridDim.x;

size\_t baseLimit = numElements >= stride ?

numElements - stride : 0;

size\_t base = 0;

for (; base < baseLimit; base += stride) {

unsigned char c = input[base + idx];

atomicAdd(&sBins[Mapping::map(c)], 1);

}

if (base + idx < numElements) {

unsigned char c = input[base + idx];

atomicAdd(&sBins[Mapping::map(c)], 1);

}

}

\_\_syncthreads();

for (

unsigned int t = threadIdx.x; t < Mapping::numBins;

t += blockDim.x

) {

atomicAdd(&bins[t], sBins[t]);

}

}

Die Anzahl Threads ist nun unabhängig von der Eingabegrösse:

constexpr size\_t nThreadsPerBlock = 256;

// …

dim3 dimGrid(1024, 1, 1);

dim3 dimBlock(nThreadsPerBlock, 1, 1);

histogram\_kernel\_atomic\_private\_stride<Mapping> <<<dimGrid, dimBlock>>> (

input, bins, numElements

);

Anmerkung: Die Anzahl Blöcke wurde mittels «Trial and Error» und aufgrund von allgemeinen Empfehlungen aus dem Web als 1024 festgelegt. Die gemessenen Ausführungszeiten scheinen nicht wesentlich von der Anzahl der Blöcke abzuhängen, sofern diese nicht zu klein wird. Bei einer grossen Anzahl Blöcken wird der Kernel wohl äquivalent zum Kernel ohne Stride. Die optimalen Parameterwerte sind wahrscheinlich geräteabhängig und nur mittels eines Skripts sinnvoll zu ermitteln.

atomic\_global\_stride

Der vierte Kernel behält das Stride-Konzept bei, verzichtet jedoch auf die Verwendung von Shared Memory – die Bins werden direkt im globalen Speicher hochgezählt:

template <typename Mapping>

\_\_global\_\_ void histogram\_kernel\_atomic\_global\_stride(

unsigned char \* input, BinType \* bins, size\_t numElements

) {

unsigned int idx = blockIdx.x \* blockDim.x + threadIdx.x;

{

int stride = blockDim.x \* gridDim.x;

size\_t baseLimit = numElements >= stride ?

numElements - stride : 0;

size\_t base = 0;

for (; base < baseLimit; base += stride) {

unsigned char c = input[base + idx];

atomicAdd(&bins[Mapping::map(c)], 1);

}

if (base + idx < numElements) {

unsigned char c = input[base + idx];

atomicAdd(&bins[Mapping::map(c)], 1);

}

}

}

Der Aufruf erfolgt wie beim dritten Kernel – die Anzahl der Threads ist unabhängig von der Eingabegrösse.

Vorgehen bei den Messungen

Das Programm allokiert zunächst (einmalig) Host- und Device-Speicher. Danach wird für *jeden* der zu messenden Kernels Folgendes durchgeführt:

* Um Verfälschungen der Messungen durch niedrigeren Takt der GPU im Idle-Zustand auszuschliessen, werden zunächst 200 Warmup-Runs mit einer Datengrösse von 100 MiB durchgeführt, d.h. der jeweilige Kernel wird 200mal aufgerufen.
* Danach werden die mittels Kommandozeile spezifizierte Anzahl von Malen
  + die Eingabedaten vom Host- zum Device-Speicher transferiert,
  + der Kernel ausgeführt, d.h. das Histogramm erstellt,
  + das Histogramm vom Device- in den Host-Speicher übertragen.

Die Zeiten für jeden dieser drei Schritte werden mittels CUDA-Events gemessen.

* Das Ergebnis der letzten Wiederholung wird durch Vergleich mit einem auf der CPU erstellten Histogramm auf Korrektheit geprüft.

Das Programm gibt die Aufrufparameter sowie die Messergebnisse im JSON-Format auf stdout aus.

Eingabedaten

Das Programm erlaubt sowohl das Einlesen einer Textdatei als auch die Generierung synthetischer Daten beliebiger Grösse. Da zu erwarten ist, dass die Laufzeit von der Anzahl auftretender Konflikte beim atomaren Zugriff auf den Speicher abhängt (was sich durch die Messwerte bestätigt), verwenden wir zweierlei synthetische Daten:

* Pseudozufällige Daten (Werte zwischen 1 und 128), die durch einen Lehmer-Zufallszahlengenerator erzeugt werden. Der Seed des Zufallszahlengenerators ist fest, d.h. jeder Durchlauf arbeitet mit den gleichen Daten. Die Implementierung des Zufallszahlengenerators entstammt der Wikipedia. Es ist nicht anzunehmen, dass die Zufallszahlen von guter Qualität sind – allerdings sind sie für den vorliegenden Zweck wohl ausreichend, da es nur darum geht, sehr kurze, systematische Muster zu vermeiden. Die Zahlenwerte von 1 bis 128 sind in den Daten annähernd gleichverteilt. Bei generierten Zahlenwerten hat jeder der 128 Zahlenwerte eine relative Häufigkeit von 0.7817 %, und der Unterschied in den relativen Häufigkeiten des seltensten und des häufigsten Zahlenwerts beträgt Prozentpunkte.
* Gleichförmige, d.h. konstante Daten, die nur aus dem Zeichen ‘a’ bestehen.

Hardware

Die Messungen wurden auf folgenden Geräten durchgeführt:

* NVIDIA Jetson Xavier NX 16 GB in einem Seeed Studio reComputer J2022;  
  GPU gemäss Datenblatt: 384-core NVIDIA Volta GPU with 48 Tensor Cores,  
  CPU gemäss Datenblatt: 6-core NVIDIA Carmel ARM v8.2 64-bit CPU 6MB L2 + 4MB L3.
* NVIDIA Tesla V100-SXM2-32GB,

funkel.fernuni-hagen.de.

Da es einfach durchzuführen war, wurde der CUDA-Code mit dem Tool hipify-perl von AMD nach HIP konvertiert und mittels hipcc für AMD-GPUs kompiliert. Es waren keine wesentlichen Anpassungen am konvertierten Code erforderlich. Zum Vergleich mit den GPUs von NVIDIA wurden die Messungen dann auch auf folgender Grafikkarte durchgeführt:

* AMD Radeon 6800 XT, 16 GB, auf einem AMD Ryzen 9 7950X mit 64 GB Hauptspeicher.

Der Code für AMD ist der Abgabe *nicht* beigefügt, findet sich aber auf Github unter

Die Ausgaben von deviceQuery für die einzelnen Geräte finden sich im Anhang.

Messergebnisse für die Beispieldatei