Projektarbeit des Studienganges "Ingenieurinformatik" im Rahmen des Softwarepraktikums

Beschleunigung von AES-Verschlüsselungen unter Zuhilfenahme einer GPU

Simon Waloschek, Benedikt Krüger, Ibrahim Alptekin, Daniel Nickchen

30. November 2010

Inhaltsverzeichnis

1 Einführung und Grundlagen

1.1 Einleitung

Moderne Verschlüsselungsalgorithmen sind im Allgemeinen sehr rechenintensiv und werden oft als Bestandteil des Betriebssystems ausgeführt. Da gewöhnliche CPUs für diese Art von Operationen nicht ausgelegt sind, wird das gesamte System durch die entstehende Auslastung gebremst. Es liegt also nahe, eine geeignetere Plattform für die Berechnung von Verschlüsselungen zu nutzen.

Im Rahmen dieses Praktikums wird daher evaluiert, inwiefern sich moderne Grafikkarten bzw. FPGAs zur optimierteren Ausführung nutzen lassen. Die Verschlüsselungen sollen transparent in den Linux-Kernel eingebunden und systemweit zur Verfügung gestellt werden.

Im Folgenden wird mithilfe des CUDA-Frameworks von NVIDIA der AES-Algorithmus auf einer GPU vom Typ "8800 GTS" aus dem Hause NVIDIA implementiert und durch eine Reihe von Tests auf eine eventuelle Verbesserung der Datendurchsatzrate hin überprüft.

1.2 Grundlagen

1.2.1 AES im Kurzüberblick

Der *Advanced Encryption Standard* (AES) ist ein symmetrisches Kryptosystem. Es wurde von Joan Daemen und Vincent Rijmen im Rahmen eines international ausgeschriebenen Wettbewerbes des *National Institute of Standards and Technology* (NIST) entwickelt. Als Nachfolger von DES und 3DES, gilt AES seit 2000 als De-facto Verschlüsselungsstandard, welcher Dank seiner starken Verschlüsselung selbst höchsten Sicherheitsansprüchen genügt.

Bei AES handelt es sich um ein Blockverschlüsselungssystem, auch Blockchiffre genannt. Es ist also ein Verschlüsselungsverfahren, bei dem der Klartext in eine Folge gleichgroßer Blöcke zerlegt wird. Diese Blöcke werden anschließend unabhängig voneinander mit einem aus einem Schlüsselwort berechneten Blockschlüssel chiffriert. Somit werden auch Chiffretextblöcke mit einer festen Länge erzeugt und letzendlich zum endgültigen Chiffretext aneinandergereiht.

AES schränkt die Blocklänge auf 128 Bit ein. Die Schlüssellänge kann jedoch zwischen 128, 192 und 256 Bit gewählt werden, weshalb zwischen den drei AES-Varianten AES-128, AES-192 und AES-256 unterschieden wird. AES bietet ein sehr hohes Maß an Sicherheit und ist in den USA sogar für staatliche Dokumente mit höchster Geheimhaltungsstufe zugelassen. Der Algorithmus ist frei verfügbar und darf ohne Lizenzgebühren eingesetzt, sowie in Software bzw. Hardware implementiert werden.

1.2.2 CUDA Framework

Das *Compute Unified Device Architecture Software Developer Kit* (CUDA SDK) wurde von NVIDA am 15. Februar 2007 erstmals der Öffentlichkeit vorgestellt. Intention dieses SDKs ist, die Programmierung aktueller Grafikkarten unter einer einheitlichen und standardisierten Schnittstelle zu ermöglichen.

Die Architektur moderner GPUs ist, aufgrund ihrer Geschichte als reine Berechnungseinheit für Bildschirmausgaben, für den Zweck ausgelegt, Operationen parallel auszuführen. Als Co-Prozessor genutzt können GPUs somit Dank der CUDA-API dazu verwendet werden, bestimmte Programmteile signifikat schneller abzuarbeiten.

CUDA basiert auf einer optimierten Variante von C ("C for CUDA") und ist damit weitestgehend plattformunabhängig. So ist es möglich, entsprechend programmierte CUDA-Anwendungen unter Windows, Linux und Mac OS auszuführen - eine kompatible Grafikkarte vorausgesetzt.

2 GPU Architektur

Moderne Grafikkarten bestehen aus einer speziellen Architektur, welche es erlaubt, viele Rechenoperationen parallel auszuführen. Je nach Grafikkarte und Hersteller ist die Architektur verschieden. Im Folgenden wird auf die Architektur der „GeForce 8800 GTS" von NVIDIA eingegangen, da Diese hier verwendet wurde.

2.1 Begriffsklärung

Um den Aufbau der Architektur und des Programmes besser verstehen zu können, wird an dieser Stelle auf einige Begrifflichkeiten eingegangen.

So bald ein CUDA-Programmfluss angestoßen wird, wird ein sogenanntes *Grid* erstellt, welches mehrere *Threads* enthält, welche wiederum alle die gleiche *Kernel-Funktion* ausführen.

Unter einer Kernel-Funktion wird die Funktion verstanden, welche parallel auf der Grafikkarte ausgeführt werden soll. In diesem Fall sind dies *AES_decrypt* und *AES_encrypt*. Da alle Kernel-Funktionen den gleichen Code ausführen, benötigen sie feste und einzigartige Variablen, über die sie sich identifizieren können - schließlich soll Jede etwas Anderes und ihr zugewiesenes berechnen.

Hierzu werden die Threads in einer zweistufigen Hierarchie angeordnet.

Auf der untersten Stufe werden die Threads in Blöcken angeordnet. Die Variablen über welche sich die Threads identifizieren können sind *blockId* und *threadId* – Also die Nummer des Blockes und Position des Threads im Block. Diese werden zur Laufzeit des Threads von der CUDA-Runtime angelegt. Die Threads eines Blockes werden in einer dreidimensionalen Struktur angelegt. So besteht jede *threadId* aus 3 Koordinaten: *x, y*

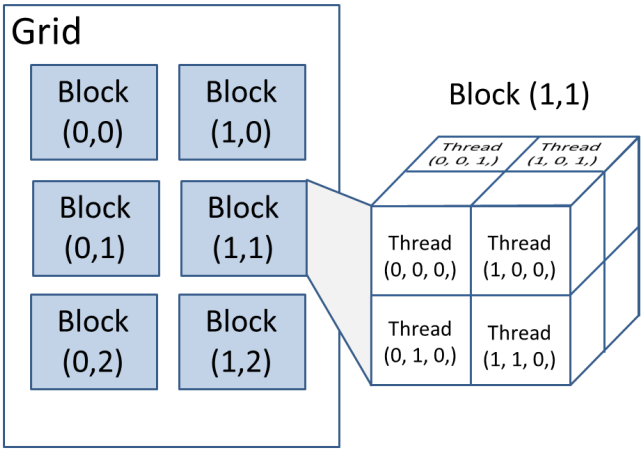

Abbildung 1: Thread-Hierarchie

und *z*. Wie viele Threads in jeder Richtung angelegt werden, wird bei dem sogenannten *Kernel-Launch* bestimmt, hierzu später mehr. Maximal können jedoch 512 Threads einem Block zugewiesen werden. Wie sie in dem Block angelegt werden (z.B. 512x1x1, 16x16x2 oder 8x16x2) ist dem Entwickler überlassen. Da die Threads in einer dreidimensionalen Struktur abgelegt werden, ist auch *threadId* eine Struktur, die diese 3 Koordinaten hält. Es kann die genaue Position des Threads abgefragt werden, indem *threadId.x*, *threadId.y* und *threadId.z* ausgewertet werden.

Auf der oberen Stufe der Hierarchie werden die Blöcke zu einem Grid zusammengefasst. Die Blöcke (mit den darin abgelegten Threads) werden in einem zweidimensionalen Koordinatensystem abgelegt. Der Programmierer kann bei dem Aufruf der Kernel-Funktion in CUDA bestimmen, wie viele Blöcke in x- und y-Richtung angelegt werden sollen. Hierbei ist zu beachten, dass die Dimensionen des Grids einmalig angelegt werden - beim *Kernel-Launch*. Später kann hierauf kein Einfluss mehr genommen werden! Auch die *blockId* muss also eine zweidimensionale Struktur beinhalten: *blockId.x* und *blockId.y*.

2.2 Der Kernel-Launch

Der Kernel-Launch bestimmt, in welchen Dimensionen die Blöcke und das Grid angelegt werden sollen. Damit wird auch bestimmt, wie viele Threads insgesamt ausgeführt werden sollen.

Zuerst wird bestimmt, wie die Dimensionen der Blöcke aussehen sollen:

```
dim3 dimBlock(2,2,2);
```

Anschließend werden die Dimensionen des Grids festgelegt:

```
dim3 dimGrid(2,3,1);
```

Bei den Dimensionen des Grids werden lediglich die x- und y-Dimensionen berücksichtigt, da die Grid-Struktur nur zweidimensional ist. Es ist üblich, als dritte Variable den Wert 1 zu setzen, jedoch ist ein beliebiger Wert möglich!

Zum Schluss wird der eigentliche Kernel-Launch ausgeführt. Hierbei ist es möglich, den Threads eine Liste an Parametern zu übergeben:

```
KernelFunktion<<<dimGrid, dimBlock>>>( PARAMETERLISTE);
```

Zu beachten ist, dass alle Threads, welche mit dem Kernel-Launch angestoßen werden, die selben Parameter bekommen.

2.3 Die Architektur der GeForce 8800 GTS

Die Grafikkarte ist über einen Chipsatz mit der CPU verbunden. Hierbei hat der Chipsatz nur Zugriff auf den *Global-Memory* der Grafikkarte: Über diesen wird also mit dem Host kommuniziert.

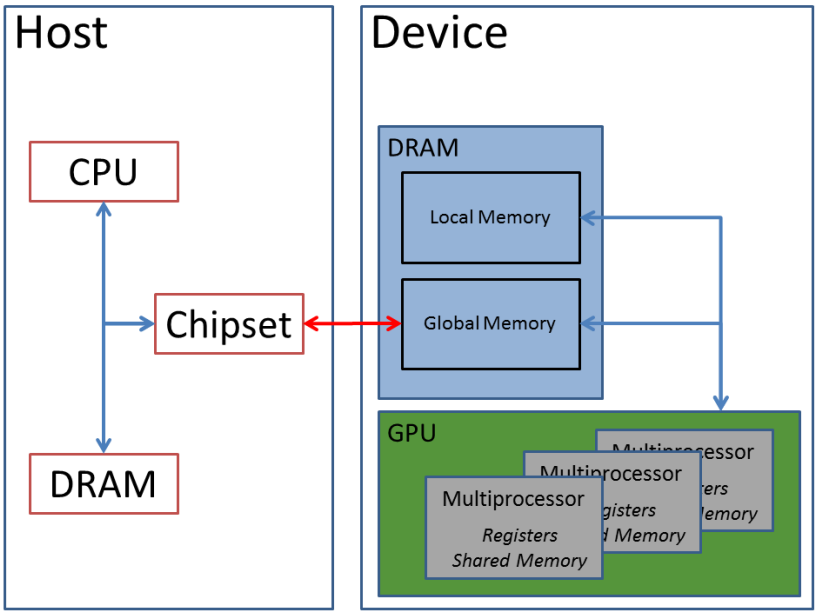

Abbildung 2: Speicherlayout

Die „GeForce 8800 GTS“ besteht aus 16 Multiprozessoren (MP). Jeder der MPs hat einen eigenen Speicher, den sog. „Shared Memory“ (16384 bytes pro MP), sowie eigene Register (8192 pro MP) und 8 Streamingprozessoren (SP). Die SPs führen jeweils genau

einen Thread aus. Ist ein Thread beendet, wird der Nächste gestartet. Diese Aufgabe übernimmt ein Scheduler.

Ist ein Block einem MP zugweisen, werden jeweils 32 Threads aus dem Block zu einem *Warp* zusammengefasst (die Größe der Warps hängt von der Hardware der Grafikkarte ab). Im Warp werden die *threadId*s fortlaufend vergeben (der erste Warp von 0 bis 31, der zweite von 32 bis 63...). Da bei der verwendeten Grafikkarte maximal 768 Threads einem MP zugeordnet werden können, kann jeder MP 768/32=24 Warps beherbergen.

Diese Mehrfachzuordnung wurde eingeführt, um hohe Latenzen bei Zugriffen auf den Speicher besser abfangen zu können. Wenn ein Befehl eines Threads aus einem Warp sehr lange auf das Ergebniss einer Speicher-Operation warten muss, so wird der Warp aus der Ausführung genommen und in eine Liste mit "wartenden Warps" eingegliedert. Dem SP, der den Thread ausführte, wird ein Warp aus der Liste der "startfertigen Warps" zugeordnet und gestartet. Dies führt dazu, dass die SPs nahezu immer ausgelastet sind.

2.4 Speicherhierarchie

Der Speicher, auf den sowohl Grafikkarte als auch CPU Zugriff haben, ist der *(D)RAM* (*Global Memory*). Dieser ist für gewöhnlich am größten - jedoch auch am langsamsten.

Von diesem Speicher werden die Daten im Verlauf des Programmes in den *Shared Memory* geladen. Dieser befindet sich auf der Grafikkarte und ist auch nur von dieser anzusprechen. Jeder MP hat die gleiche Größe des *Shared Memory*: 16 KB. Dieser wird auf die SPs des MPs aufgeteilt.

Zusätzlich gibt es noch einen Speicher für Konstanten des Programmes, den sog. *Constant Memory*. Dieser ist physikalisch auf 64 KB beschränkt und für alle SPs des MPs zugänglich! Für die Berechnungen hat jeder SP darüber hinaus *Register*, welche die schnellsten Zugriffszeiten aufweisen.

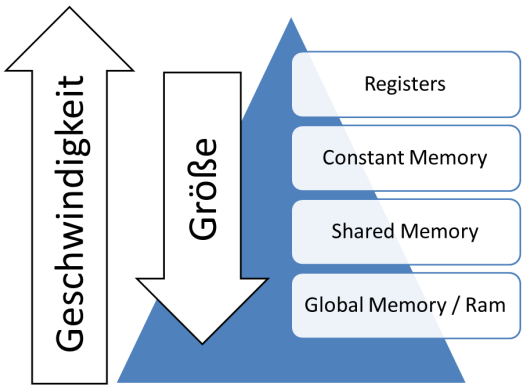

Abbildung 3: Speicherhierarchie Grafikkarte

Bei der Speicherhierarchie ist zu beachten, dass die beiden wichtigsten Ebenen (*DRAM* und *Global Memory*) physikalisch getrennt liegen (Vgl. Abb. 2). Der *RAM* liegt auf dem Mainboard und ist für CPU und Grafikkarte ansprechbar. Der *Global Memory* ist nur von der Grafikkarte ansprechbar. Die Ergebnisse der Berechnungen sind vom *Shared Memory* explizit in den *RAM* zu kopieren!

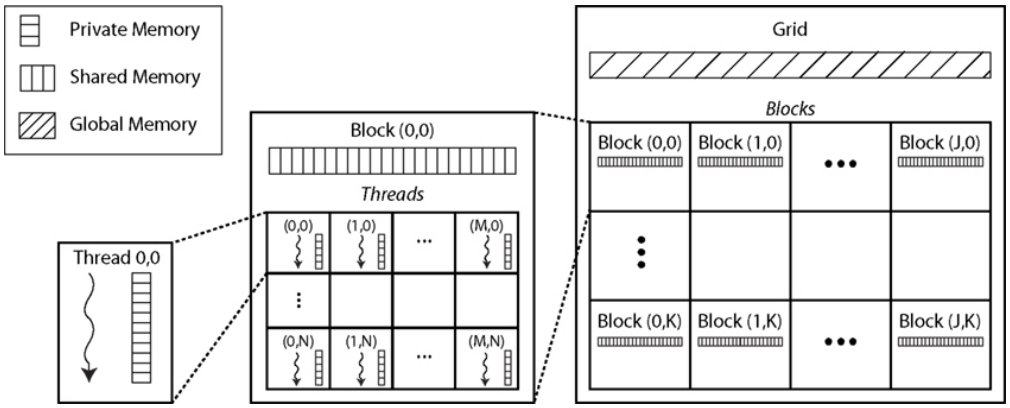

Abbildung 4: 8800GTS-Architektur

Ein Thread wird in seinem Block ausgeführt. Hierbei kann der Thread sowohl auf seine *Register* (vgl. Abb. 4: *Private Memory*), wie auch auf den *Shared Memory* zugreifen. Der *Shared Memory* dient hierbei zur Kommunikation der Threads untereinander; Alle Threads haben Lese- und Schreibzugriff auf ihn. Er ist hierbei nicht gecached, d.h. Veränderungen im *Shared Memory* werden direkt in den Hauptspeicher geschrieben. Der Datenaustausch zwischen CPU und Grafikkarte geschieht über den *Global Memory*. Beim Programmieren wird explizit angegeben, welche Daten an welche Position (unter welchem "Namen") auf die Grafikkarte kopiert werden sollen.

Im Folgenden ein kleines Beispiel zur Veranschaulichung:

```
float *a_h, *a_d, *b_h;
nBytes = sizeof(float);
a_h = (float*) malloc(nBytes);
b_h = (float*) malloc(nBytes);
cudaMalloc ((void**) &a_d, nBytes);
cudaMemcpy(a_d, a_h, nBytes, cudaMemocpyHostToDevice);

[...]

cudaMemcpy(b_h, a_d, nBytes, cudaMemcpyDecviceToHost);
free(a_h);
free(b_h);
cudaFree(a_d);
```

Es werden zunächst 3 Variablen angelegt: a_h, b_h (Host-Daten) sowie a_d (Device-Daten). Hierfür wird der Platz im Speicher -sowohl auf dem Host als auch auf der Grafikkarte- reserviert. Der Befehl *cudaMalloc* legt einen *nBytes* großen Platz im Speicher für die Variable a_d an, in den dann mittels *cudaMemcpy* die Werte von a_h kopiert werden. Nach einigen Berechnungen wird danach der Inhalt der Device-Variable a_d zurück in die Host-Variable b_h kopiert. Hierbei ist zu beachten, dass die Daten wirklich kopiert werden. Es sollten also möglich wenig Daten ausgetauscht werden müssen, bzw. das Kopieren sollte nicht doppelt erfolgen. Zum Schluss werden die Daten aus dem Host- und dem Device-Speicher gelöscht.

2.5 Ausführung

Jeder MP bekommt einen Warp zugewiesen. Bei der verwendeten „GeForce 8800 GTS“ entspricht ein Warp einer Ansammlung von 32 Threads. Die Threads bestehen immer aus dem selben Quellcode. Sie bekommen intern IDs zugewiesen, über welche sie sich identifizieren und - entsprechend ihrem Programmfluss - beeinflussen können. Der Thread wird gemeinsam mit den Eingabedaten auf die Grafikkarte kopiert, wobei die Eingabedaten in den *Shared Memory* abgelegt werden. Anschließend beginnen die SPs mit der Bearbeitung der Daten. Wenn alle Threads berechnet wurden - im Normalfall also jeder SP 2 Threads bearbeitet hat - werden die Ergebnisse aus dem *Shared Memory* zurück in den RAM kopiert und von dort über die CPU weiterverwendet. Im Anschluss ist der MP bereit für den nächsten Warp und das Ganze beginnt wieder von vorne.

3 AES-Implementierung

3.1 Algorithmus im Detail

Im diesem Kapitel wird der Ablauf des AES Algorithmus vorgestellt. Er besteht aus mehreren Phasen, auf die im Folgenden genauer eingegangen wird. Anzumerken ist, dass bei den Beispielen der Einfachheit halber von einer 128 Bit-Verschlüsselung ausgegangen wird.

Jeder Block wird zunächst in eine zweidimensionale Tabelle mit vier Zeilen und vier Spalten geschrieben, deren Zellen ein Byte groß sind. Jeder Block wird nun nacheinander bestimmten Transformationen unterzogen. Anstatt jeden Block einmal mit dem Schlüssel zu verschlüsseln, wendet AES verschiedene Teile des erweiterten Originalschlüssels nacheinander auf den Klartext-Block an. Die Anzahl r dieser Runden variiert und ist von der Schlüssellänge k abhängig:

Schlüssellänge k	Runden r
128	10
192	12
256	14

Tabelle 1: Rundenlängen

Der Ablauf jeder einzelnen Block-Verschlüsselung entspricht folgendem Schema:

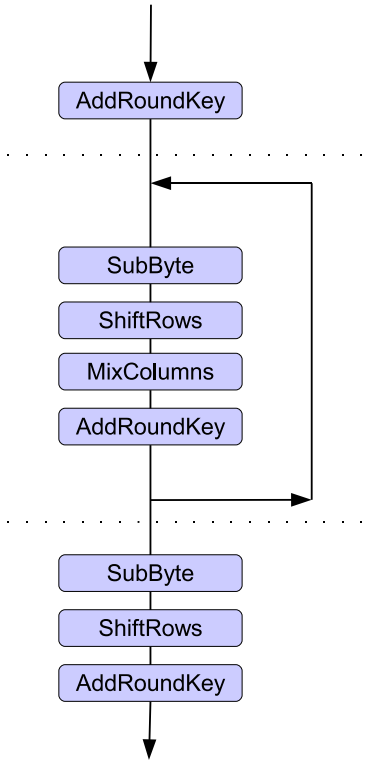

Abbildung 5: Ablaufschema

3.1.1 Schlüsselexpansion

Vor der ersten Chiffrierung muss zunächst einmal der Schlüssel entsprechend aufbereitet werden.

Der Benutzerschlüssel muss in *r+1* Teilschlüssel aufgeteilt werden, die sogenannten Rundenschlüssel. Diese müssen dieselbe Länge wie die Blöcke besitzen. Dies bedeutet, dass der Benutzerschlüssel zunächst auf die Länge *k*(r+1)* expandiert werden muss. Aus Diesem werden die Rundenschlüssel erzeugt und ebenfalls in Tabellen (Arrays) mit vier Spalten und vier Zeilen gespeichert. Für die Erzeugung der ersten Spalte des jeweils nächsten

Rundenschlüssels wird zunächst die letzte Spalte des vorherigen Schlüssels, am Anfang also des Benutzerschlüssels, genommen und um eine Zeile nach oben rotiert. Die oberste Zelle wird unten wieder eingefügt. Nun erfolgt eine Substitution der vier Werte mit Hilfe der sogenannten Substitutionsbox. Sie ist meist als Array aufgebaut und gibt an, wie jedes Byte durch einen anderen Wert zu ersetzen ist.

Die Konstruktion der S-Box unterliegt Designkriterien, die die Anfälligkeit für die Methoden der linearen und der differentiellen Kryptoanalyse sowie für algebraische Attacken minimieren sollen. Mit Hilfe der S-Box wird jedes Byte des Blocks durch ein Äquivalent ersetzt und die Daten somit monoalphabetisch verschlüsselt. Diese neu erzeugte Spalte wird mit der drei Spalten zurückliegenden Spalte und mit der ersten Spalte der sogenannten Rcon-Tabelle XOR-verknüpft.

Die Rcon-Tabelle ist ebenfalls in Form eines Arrays mit vier Zeilen und einer Spalte für jeden Rundenschlüssel aufgebaut. Sie enthält konstante Werte, die auf einem mathematischen System beruhen. Die aus der XOR-Verknüpfung erzeugten Werte ergeben die erste Spalte des nächsten Rundenschlüssels. Die restlichen drei Spalten ergeben sich jeweils aus einer XOR-Verknüpfung der davor liegenden und der zu dieser drei zurückliegenden Spalte. Für alle folgenden Rundenschlüssel läuft die Berechnung analog ab.

3.1.2 Vorrunde

Bei *AddRoundKey* erfolgt eine XOR-Verknüpfung zwischen dem zu verschlüsselnden Block und dem ersten Rundenschlüssel (siehe Abb. 6). Nur an dieser Stelle ist der Algorithmus vom Benutzerschlüssel abhängig.

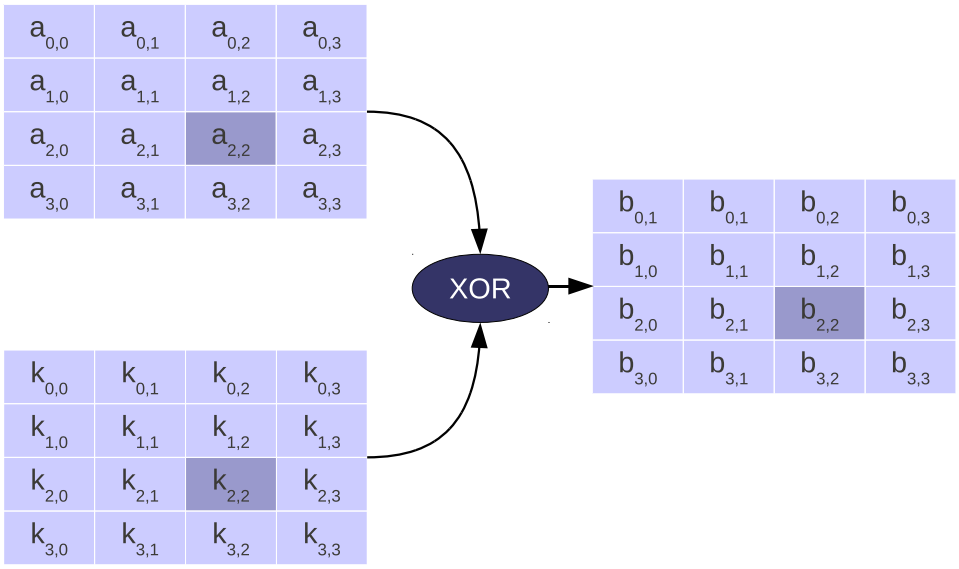

Abbildung 6: AddRoundKey

3.1.3 Verschlüsselungsrunden

In den folgenden Verschlüsselungsrunden wird zunächst eine Substitution mittels der erwähnten S-Box durchgeführt.

Im darauffolgenden Schritt (*ShiftRow*) werden die Zeilen um eine bestimmte Anzahl von Spalten nach links verschoben und links hinausgeschobene Zellen rechts wieder angefügt. Die erste Zeile bleibt konstant, die Zweite wird um eine, die Dritte um zwei und die Vierte um drei Spalten verschoben. Abbildung 7 veranschaulicht dieses Vorgehen.

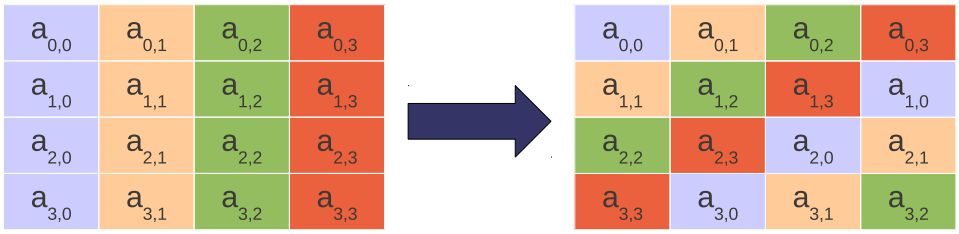

Abbildung 7: ShiftRow

Im *MixColumn*-Schritt wird zunächst jede Zelle mit einer Konstanten multipliziert und dann die Spalten mit den Ergebnissen der Multiplikation XOR verknüpft. Durch eine geschickte Analyse dieser Operation, vereinfacht sich die Rechnung zu einer simplen Matrizenmultiplikation.

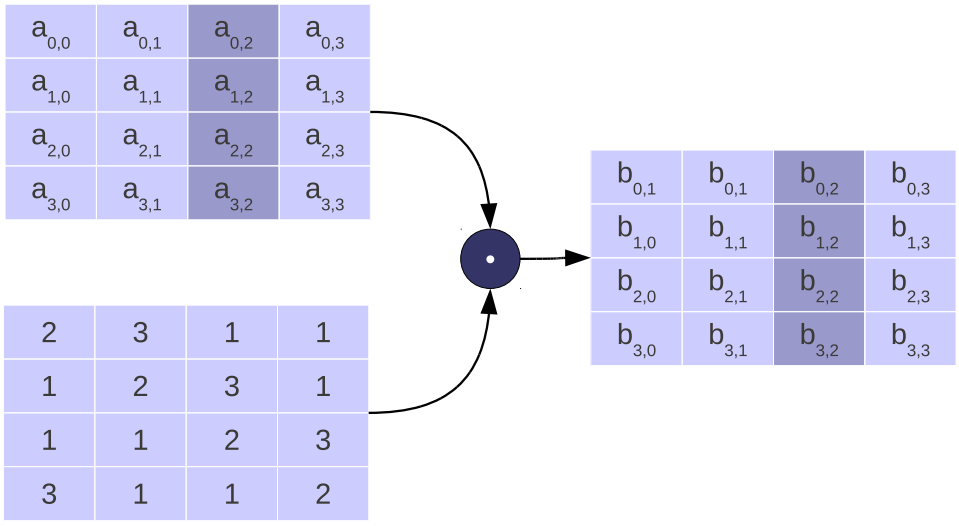

Abbildung 8: MixColumn

Am Ende jeder Verschlüsselungsrunde wird noch einmal *AddRoundKey* ausgeführt. Abgeschlossen wird die Verschlüsselung mit der Schlussrunde. Sie verläuft identisch zu den

übrigen Verschlüsselungsrunden, mit dem Unterschied, dass keine *MixColumn*-Funktion ausgeführt wird.

3.1.4 Entschlüsselung

Bei der Entschlüsselung von Daten wird entsprechend rückwärts vorgegangen. Die Daten werden zunächst wieder in zweidimensionale Tabellen gelesen und die Rundenschlüssel generiert. Allerdings wird nun mit der Schlussrunde angefangen und alle Funktionen in jeder Runde in der umgekehrten Reihenfolge aufgerufen. Durch die vielen (symmetrischen) XOR-Verknüpfungen unterscheiden sich die meisten Funktionen zum Entschlüsseln nicht von denen zum Verschlüsseln. Jedoch muss eine andere S-Box genutzt werden (die sich aus der originalen S-Box berechnen lässt) und die Zeilenverschiebungen erfolgen in die andere Richtung.

3.2 C++ Implementierung

Ausgangspunkt der weiteren Bearbeitung ist eine unfertige C++ Implementierung des Algorithmus von Paulo S. L. M. Barreto. Der Autor hat bereits erste Ansätze zur Datenverschlüsselung auf einer GPU programmiert. Er hat das Projekt jedoch aus unbekannten Gründen eingestellt, sodass noch einige Ergänzungen und Änderungen vorgenommen werden müssen.

Die Implementierung ist Dank der Ausnutzung komplexerer Galois-Feld-Operationen vergleichsweise schnell. Hierbei werden die Berechnungen durch Nutzung vorkalkulierter Tabellen auf Kosten der Speichereffizienz beschleunigt. Der Aufwand der Verschlüsselung verringert sich somit auf eine Reihe (schneller) Binäroperationen, welche sich auf der GPU vorteilhaft parallel ausführen lassen.

Zusätzlich ist anzumerken, dass zugunsten der Parallelität der sogenannte *Electronic Code Book Mode* (ECB) verwendet wird, also jeder Block mit dem gleichen expandierten Schlüssel chiffriert wird.

3.2.1 Schnittstelle zur Außenwelt

Um von außen auf die Verschlüsselungsfunktionen zugreifen zu können, enthält die genutzte AES-Klasse neben dem Konstruktur auch "öffentliche" Funktionen für die Schlüsselexpansionen sowie für die Ver- bzw. Entschlüsselung.

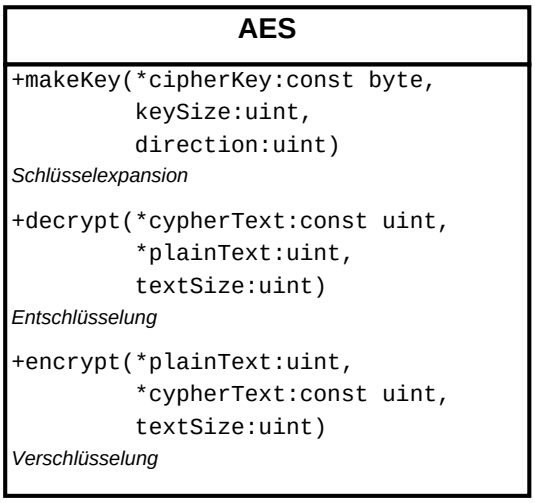

Abbildung 9: Klassendiagramm mit public Funktionen

3.3 CUDA-spezifische Veränderungen

3.3.1 Prozessaufteilung

Zur bestmöglichen Ausnutzung der parallelen Verarbeitungsmöglichkeiten der GPU, sollte die Arbeitslast auf möglichst viele Threads aufgeteilt werden. Zu diesem Zweck werden zur Laufzeit die (hardwarespezifischen) Eigenschaften der GPU ausgewertet und die Programmausführung auf die maximal mögliche Anzahl an Blöcken verteilt, in denen schließlich alle verfügbaren Threads genutzt werden. Im Quelltext wird dies auf folgende Weise gelöst:

```
1  void AES::encrypt(const uint *pt, uint *ct, uint n = 1) {
2      uint *cpt, *cct;
3      uint size = (n << 2)*sizeof(uint);
4
5      // Speicherreservierung auf GPU
6      cudaMalloc((void**)&cpt, size);
7      cudaMalloc((void**)&cct, size);
8
9      // Kopieren des Klartextes in den Global Memory der Grafikkarte
10     cudaMemcpy(cpt, pt, size, cudaMemcpyHostToDevice);
11
12     // Ermitteln der architekturspezifischen Hardware
13     struct cudaDeviceProp prop;
14     cudaGetDeviceProperties(&prop, 0);
15
16     // Aufteilung der Berechnungen auf möglichst viele Threads
17     // Um Kompatibilitätsprobleme zu vermeiden, werden bei fehlender
18     // Parameterübergabe von n sinnvolle Defaultwerte genutzt.
19     uint blocks, threads = 1;
20     if(n != 1) {
```

```
21          threads = (n < prop.maxThreadsPerBlock*2) ? n / 2 : prop.
                maxThreadsPerBlock;
22      }
23      blocks = n / threads;
24      dim3 dimBlock(threads);
25      dim3 dimGrid(blocks);
26
27      // Erstellen des Kernel-Services sowie Start der Verschlüsselung
28      // Das Programm wartet an dieser Stelle auf die Rückgabe der
29      // Berechnungen.
30      AES_encrypt<<<dimGrid, dimBlock, size>>>(cpt, cct, ce_sched, Nr);
31
32      // Kopieren des Chiffrats in den Arbeitsspeicher
33      cudaMemcpy(ct, cct, size, cudaMemcpyDeviceToHost);
34
35      // Speicherfreigabe
36      cudaFree(cpt);
37      cudaFree(cct);
38  }
```

Das Programm ermittelt in den Zeilen 20 bis 25 die benötigten Parameter zur optimalen Aufteilung der Arbeitslast, erstellt Grids und Blöcke in entsprechender Dimensionierung und stößt den Kernel-Service für jeden einzelnen der Threads an.

3.3.2 Speichernutzung

Neben dem obligatorischen Kopiervorgängen vom RAM des Host-PCs in den *Global Memory* der Grafikkarte und zurück, werden die temporären Variablen in den für ihren jeweiligen Zweck laufzeiteffizientesten Speicherarten erstellt und verwendet. Da zur Verschlüsselung - wie bereits beschreiben - eine Vielzahl von vorberechneten Tabellen benutzt werden, werden diese Werte bei Aufruf des Klassenkonstruktors in den schnellen *Constant Memory* kopiert.

Der folgende Ausschnitt demonstriert den Speicherzugriff innerhalb der Kernel-Funktion *AES_ encrypt* sowie (nebensächlich) die Adressierung des zu verschlüsselnden Textes im *Global Memory*.

```
1   __global__ void AES_encrypt(const uint *pt, uint *ct, uint *rek, uint Nr) {
2           // Berechnung des (threadabhängigen) Offsets zur Adressierung des Speichers
3           int x = blockIdx.x * blockDim.x + threadIdx.x;
4           int y = blockIdx.y * blockDim.y + threadIdx.y;
5           int i = x + y * gridDim.x * blockDim.x;
6           int offset = i << 2;
7
8           // Deklarieren der temporären Variablen im Shared Memory
9           __shared__ __device__ uint s0, s1, s2, s3, t0, t1, t2, t3;
10
11          // Beginn der Verschlüsselung...
12          s0 = pt[offset + 0] ^ rek[0];
13          s1 = pt[offset + 1] ^ rek[1];
14          s2 = pt[offset + 2] ^ rek[2];
15          s3 = pt[offset + 3] ^ rek[3];
16
```

```
17          [...]
18
19  }
```

Der threadspezifische Offset zur Speicheradressierung berechnet sich, wie in Kapitel 2.1 beschrieben, aus der *blockId*, der Blockgröße *blockDim* und der *threadId*. Anschließend werden in Zeile 9 die temporären Variablen *s0*, *s1* etc. im *Shared Memory* angelegt. Dies erfordert zwar nach Abschluss der Verschlüsselung eine zusätzliche Kopieroperation zurück in den *Global Memory*, beschleunigt den Algorithmus aber dennoch aufgrund der schnelleren Zugriffsgeschwindigkeit auf die im weiteren Verlauf häufig genutzten Speicherstellen.

4 Tests und Benchmarks

4.1 Testumgebung

Zur Evaluierung der Korrektheit der entwickelten Lösung, werden zunächst vom NIST vorgegebene Klartext-Chiffre-Paare inklusive des passenden Schlüssels verwendet. Diese Paare umfassen verschiedene Verschlüsselungsstärken und Textlängen und eignen sich daher für einen aussagekräftigen Praxistest.

Um die Performance zu beurteilen und relevante Laufzeitanalysen durchzuführen, bietet NVIDIA als Teil des CUDA-SDKs den sogenannten "CUDA Profiler". Dieses grafische Tool erlaubt einen detaillierten Einblick in die Laufzeiten einzelner CUDA-Befehle und eignet sich somit zur Bewertung ihrer Geschwindigkeit. Auch können damit eventuelle Bottlenecks im Code aufgespürt und ggf. korrigiert werden.

Da die folgenden Tests möglichst unabhängig von der restlichen Hardware des Testcomputers sein sollten, wird unter Linux eine RAM-Disk mit verschieden großen (zufälligen) Binärdateien erstellt, um den Einfluss der Festplattenlese- und -schreibgeschwindigkeit zu minimieren. Diese Dateien werden dann unter Einbeziehung des CUDA Profilers vollständig ver- bzw. entschlüsselt.

4.2 Ergebnisse

5 Resumée