Projektarbeit des Studienganges "Ingenieurinformatik" im Rahmen des Softwarepraktikums

Beschleunigung von AES-Verschlüsselungen unter Zuhilfenahme einer GPU

Simon Waloschek, Benedikt Krüger, Ibrahim Alptekin, Daniel Nickchen

25. 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 "GTS 8800" 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, 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.

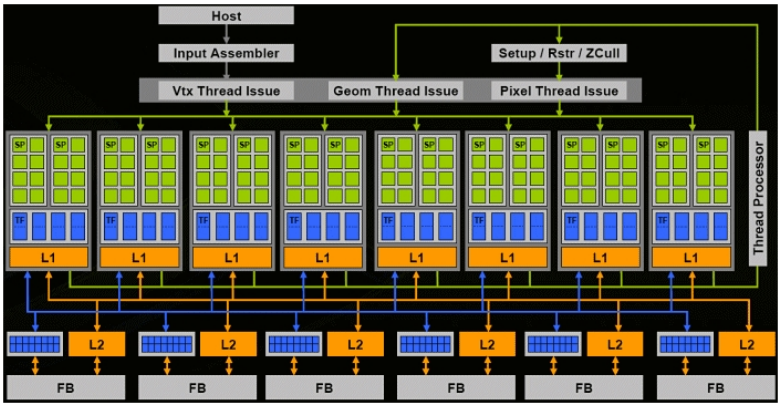

Abbildung 1: 8800GTS-Architektur

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 können GPUs somit Dank der CUDA-API dazu genutzt 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 jedoch verschieden. Im Folgenden wird auf die Architektur der „GeForce 8800 GTS“ von NVIDIA eingegangen, da diese verwendet wurde.

Wie oben dargestellt, besteht die „GeForce 8800 GTS“ aus 8 Multiprozessoren (MP). Jeder der MPs hat einen eigenen Speicher, den sog. „Shared Memory“ (16384 bytes pro MP), sowie eigene Register (8192 pro MP) und 16 Streamingprozessoren (SP). Die SPs führen jeweils genau einen Thread aus. Die Eingabedaten werden aus dem L1-Cache geladen und die Ergebnisse der Berechnungen der SPs werden dort wieder abgelegt.

Während des Programmierens wird entschieden, in wie viele „Blöcke“ das Programm aufgeteilt werden soll - also die Parallelität des Programmes im Detail aussieht. Im optimalen Fall lässt sich ein Vielfaches der Anzahl der MPs - Blöcke bilden, womit eine absolute parallele Ausführbarkeit garantiert wäre.

Ebenfalls beim Programmieren entscheidet der Entwickler, wann welche Daten aus oder in welchen Speicher gelegt werden sollen.

2.1 Speicherhierarchie

Der Speicher, auf den sowohl Grafikkarte als auch CPU Zugriff haben, ist der *RAM* („Global Memory“). Dieser ist für gewöhnlich am größten und 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 Shares 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.

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

2.2 Ausführung

Jeder MP bekommt einen Warp zugewiesen. Bei der verwendeten „GeForce 8800 GTS“ entspricht ein Warp einer Ansammlung von 32 Threads. Die Threads bestehen pro Ausführung in dem Warp immer aus dem gleichen Quellcode. Sie bekommen jedoch intern IDs zugewiesen, über welche sie sich identifizieren - und entsprechend ihren 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 - also im Normalfall 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.

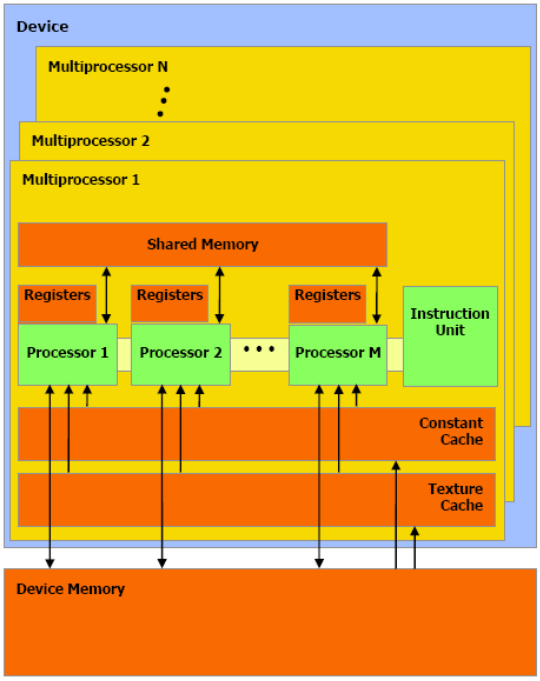

Abbildung 2: Speicherhierarchie Grafikkarte

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 sei, dass bei 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

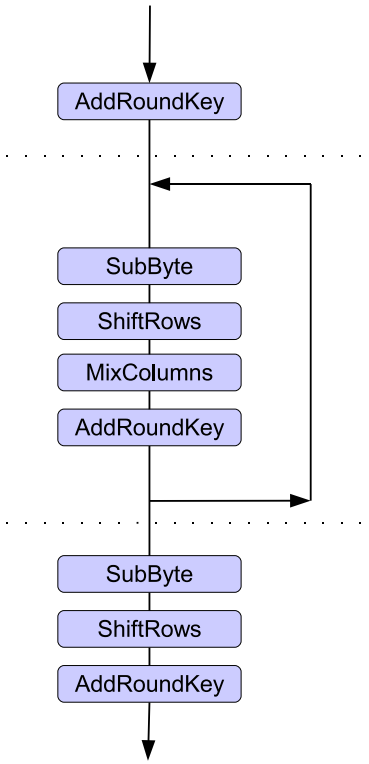

Abbildung 3: Ablaufschema

Der Ablauf jeder einzelnen Block-Verschlüsselung entspricht folgendem Schema:

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 haben, was 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

Abbildung 4: Berechnung der Rundenschlüssel

minimieren sollen. Mit Hilfe der S-Box wird jedes Byte des Blocks durch ein Äquivalent ersetzt und die Daten somit monoalphabetisch verschlüsselt. Diese neuerzeugte 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 4 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. Abbildung 3 veranschaulicht dieses Vorgehen.

3.1.2 Vorrunde

Bei AddRoundKey erfolgt eine XOR-Verknüpfung zwischen dem zu verschlüsselnden Block und dem ersten Rundenschlüssel (siehe Abb.4). Nur an dieser Stelle ist der Algorithmus vom Benutzerschlüssel abhängig.

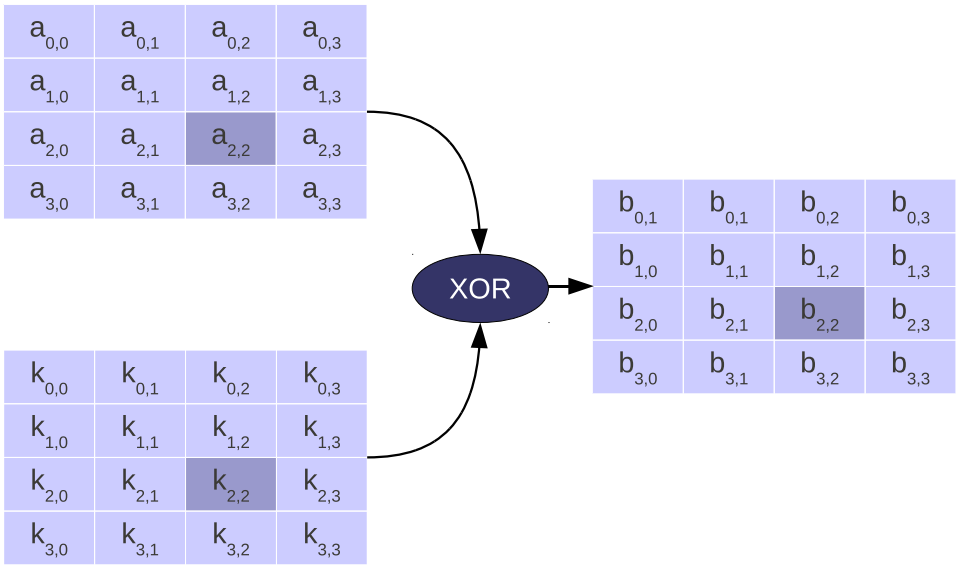

Abbildung 5: AddRoundKey

3.1.3 Verschlüsselungsrunden

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

Im darauffolgenden Schritt, genannt 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 5 veranschaulicht dieses Vorgehen.

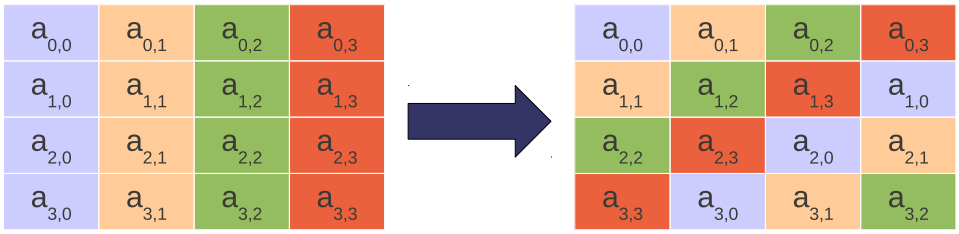

Abbildung 6: ShiftRow

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.

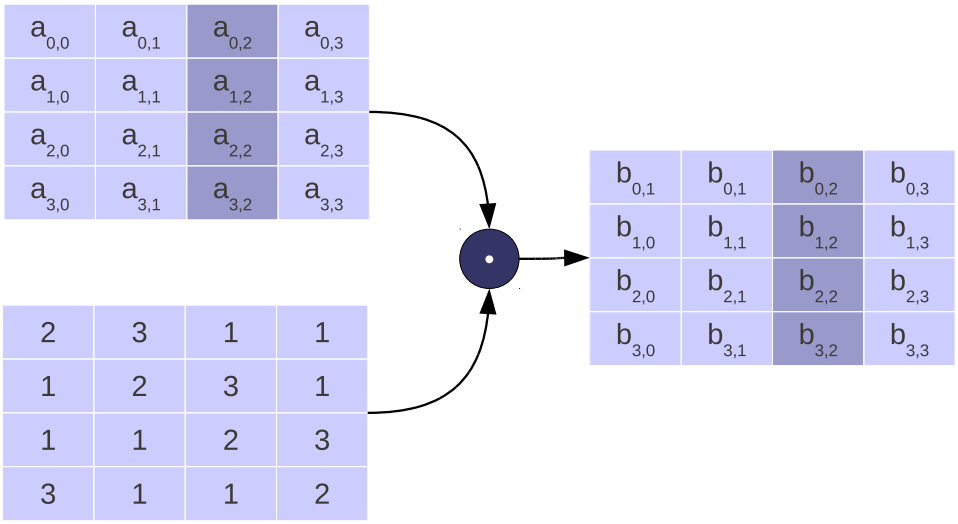

Abbildung 7: MixColumn

Zum 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, 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 können.

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 "öffentliche" Funktionen für die Schlüsselexpansionen sowie für die Ver- bzw. Entschlüsselung.

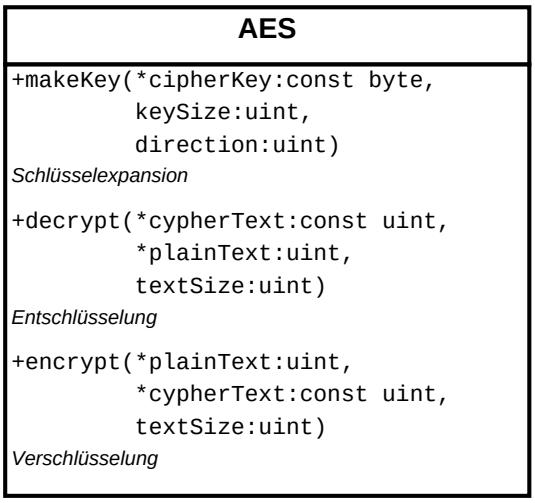

Abbildung 8: Klassendiagramm mit public Funktionen

3.3 CUDA-spezifische Veränderungen

3.3.1 Prozessaufteilung

3.3.2 Speichernutzung

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 derer Geschwindigkeit. Auch können damit eventuelle “Flaschenhälse” 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 Festplattenlesegeschwindigkeit zu minimieren.

4.2 Ergebnisse

5 Ausblick