**ACADEMIA DE STUDII ECONOMICE  
Facultatea de Cibernetica, Statistica si Informatica Economica  
Cursul de master SECURITATEA INFORMATICA**

**PROIECT DE DIZERTATIE**

*Criptografie folosind procesorul grafic(GPU)*



**Indrumator:  
Prof. Dr. Ing. Valentin Cristea**

**Student:  
Ing. Sandu Gh. Cristian**

**Bucuresti 2009**

Cuprins

[1. Introducere 4](#_Toc223834189)

[1.1. Procesorul grafic ca procesor paralel 4](#_Toc223834190)

[1.2. Modelul de programare CUDA 4](#_Toc223834191)

[2. Obiectivele proiectului 4](#_Toc223834192)

[2.1. Criptare AES pe GPU 5](#_Toc223834193)

[2.2. Perfomanta AES pe GPU 5](#_Toc223834194)

[3. Descrierea implementarii folosind CUDA 5](#_Toc223834195)

[3.1. Tehnologii folosite 5](#_Toc223834196)

[3.1.1. CUDA 5](#_Toc223834197)

[3.1.1.1. Arhitectura GPU-ului expusa de catre CUDA 6](#_Toc223834198)

[3.1.1.2. CUDA – Extensie a limbajului C 8](#_Toc223834199)

[3.1.1.3. CUDA – Blocuri de fire de executie 9](#_Toc223834200)

[3.1.1.4. CUDA – Ierarhia memoriei 11](#_Toc223834201)

[3.1.1.5. CUDA – Arhitectura SIMT 12](#_Toc223834202)

[3.1.1.6. Interoperabilitate CUDA si C++ 13](#_Toc223834203)

[3.1.2. Framework-ul WxWidgets 13](#_Toc223834204)

[3.1.2.1. O scurta istorie a framework-ului WxWidgets 13](#_Toc223834205)

[3.1.2.2. Mai multe despre WxWidgets 15](#_Toc223834206)

[3.1.2.3. Arhitectura WxWidgets 17](#_Toc223834207)

[3.1.3. Algoritmul AES 17](#_Toc223834208)

[3.1.3.1. Specificatia AES 18](#_Toc223834209)

[3.1.3.1.1. Starea, cheia si numarul de runde 18](#_Toc223834210)

[3.1.3.1.2. Rundele 19](#_Toc223834211)

[3.1.3.1.3. Transformarea ByteSUb 19](#_Toc223834212)

[3.1.3.1.4. Transformarea ShiftRow 20](#_Toc223834213)

[3.1.3.1.5. Transformarea MixColumn 21](#_Toc223834214)

[3.1.3.1.6. Transformarea AddRoundKey 21](#_Toc223834215)

[3.1.3.1.7. Derivarea cheilor de runda 22](#_Toc223834216)

[3.1.3.1.7.1. Expansiunea cheilor 22](#_Toc223834217)

[3.1.3.1.7.2. Selectarea cheii de runda 23](#_Toc223834218)

[3.1.3.1.8. Cifrul 23](#_Toc223834219)

[3.1.3.1.9. Implementare pe un sistem pe 32 de biti 24](#_Toc223834220)

[3.1.3.1.10. Paralelism 26](#_Toc223834221)

[3.1.3.2. Implementare eficienta folosind CUDA 26](#_Toc223834222)

[3.2. Descrierea bibliotecii de criptare AES folosind CUDA 27](#_Toc223834223)

[3.2.1. Descrierea bibliotecii de criptare 27](#_Toc223834224)

[3.3. Detalii de implementare a algoritmului AES folosind CUDA 28](#_Toc223834225)

[4. Folosirea bibliotecii de criptare in aplicatii 29](#_Toc223834226)

[4.1. Aplicatie demonstrativa 29](#_Toc223834227)

[5. Masurarea performantelor bilbiotecii AES CUDA 30](#_Toc223834228)

[5.1. Date experimentale 30](#_Toc223834229)

[5.2. Concluzii 31](#_Toc223834230)

[6. Concluzii ale proiectului. Dezvoltari ulterioare 31](#_Toc223834231)

[7. Bibliografie 33](#_Toc223834232)

[8. Anexa 1: Compilarea si rularea aplicatiei 33](#_Toc223834233)

# Introducere

Criptografia este un subiect sensibil in domeniul securitatii. Atat cei care vor sa securizeze o resursa(un document, o retea etc.) cat si cei care vor sa atace securitatea au nevoie de putere de calcul. Algoritmii criptografici trebuie sa fie destul de simpli pentru a rula pe hardware obisnuit si destul de complecsi pentru a asigura securitatea necesara utilizatorului.

In ultimii ani, procesorul grafic inclus in placile video „mainstream” a devenit din ce in ce mai puternic putand fi folosit si pentru calcule ce nu tin de grafica. Odata cu introducerea Framework-urilor de calcul paralel pe GPU, cum este CUDA de la nVidia, s-au deschis foarte multe posibilitati in ce priveste utilizarea GPU-ului in aplicatii ce pot beneficia de paralelism masiv.

Lucrarea isi propune in continuare sa analizeze eficienta utilizarii GPU-ului in criptografie; in particular, implementarea algoritmului Rijndael(AES).

# Procesorul grafic ca procesor paralel

Ideea de GPGPU(General Programming GPU) a devenit relativ de populara in ultimii ani mai ales datorita arhitecturii sale nativ paralele(„manycore”) – aplicatiile grafice presupun multe prelucrari paralele.

In esenta, GPU-ul are un numar de „multiprocesoare” (16, 24, 32... mai multe la arhitecturile mai noi). Fiecare multiprocesor are la randul lui 8 procesoare *scalare* optimizate pentru operatii in virgula mobila. Arhitectura este asemanatoare cu arhitectura SIMD din taxonomia Flynn cu diferenta ca paralelismul este la nivel de „fir de executie”. Un „manunchi” de fire de executie este programat sa fie executat pe fiecare multiprocesor. Firele de executie ce ruleaza pe acelasi multiprocesor trebuie sa respecte anumite restrictii in timp ce fire de executie ce ruleaza pe multiprocesoare diferite sunt independente din punctul acesta de vedere. Arhitectura aceasta este numita SIMT(single instruction, multiple threads).

Aceasta arhitectura paralela se scaleaza bine la probleme de prelucrare paralele, altele decat cele din grafica.

Introducerea Framework-urilor gen nVidia CUDA a deschis si mai mult utilizarea GPU-ului ca GPGPU.

# Modelul de programare CUDA

CUDA pune la dispozitie un model de programare paralel ce permite dezvoltatorilor sa utilizeze puterea procesorului grafic pentru orice algoritm ce suporta paralelizare. Limbajul pentru CUDA este o extensie a limbajului C (este un superset strict al limbajului C) oferind programatorilor o abordare familiara. Alternativa de la AMD se numeste Stream SDK(fostul CTM – close to the metal).

CUDA permite accesul la resursele GPU-ului – memorie si procesoare pentru a fi utilizate pentru programare generala. Programele pe GPU se executa in grupuri de fire de executie pe multiprocesoarele GPU-ului. CUDA ofera destul abstractizare pentru ca dezvoltatorul sa nu se preocupe de felul in care sunt executate aceste fire si destul transparenta pentru a permite optimizari pe baza distributiei firelor de executie pe multiprocesoare.

# Obiectivele proiectului

Proiectul isi propune studieze performantele utilizarii GPU-ului pentru criptare. Modelul de programare paralele va fi cel oferit de nVidia CUDA. Mai concret, isi propune sa studieze o implementare de criptare/decriptare AES pe GPU.

# Criptare AES pe GPU

Algoritmii care beneficiaza cel mai mult de paralelizare sunt cei care executa prelucrari independente pe mai multe elemente de date. In modul standard de functionare (AES ECB), algoritmul de criptare AES este un astfel de algoritm. In modul CBC, in care criptarea unui bloc depinde de criptarea blocului anterior, beneficiul obtinut din paralelizare este minim, in schimb decriptarea poate fi paralelizata eficient.

# Perfomanta AES pe GPU

Conform unei alte lucrari care studiaza criptarea folosind GPU-ului imbunatatirea asteptata este de pana la 5x in functie marimea blocului de criptat si numarul de multiprocesoare ale GPU-ului. Imbunatatirea(*speedup*-ul) este mai semnificativa cu cat blocul de date de criptat este mai mare.

# Descrierea implementarii folosind CUDA

Proiectul va fi implementat sub forma unei biblioteci care combina limbajul C++ cu extensiile CUDA pentru C ce permit executia pe GPU.

# Tehnologii folosite

Pe langa CUDA, proiectul foloseste wxWidgets pentru partea de interfata grafica. Implementarea va fi in limbajul C++ combinat cu parți in C pentru interoperabilitatea cu toolkit-ul CUDA.

# CUDA

Evolutia CPU-urilor „multicore” si „manycore” a adus procesoare paralele pe piata „mainstream”. In plus, paralelismul lor continua sa se scaleze cu legea lui Moore. Provocarea este sa dezvoltam software care se scaleaza transparent la paralelismul procesoarelor si sa beneficieze de numarul din ce in ce mai mare de core-uri, asa cum aplicatiile de grafica 3D isi scaleaza transparent paralelismul lor la GPU-urile „manycore” cu un numar foarte variat de core-uri.

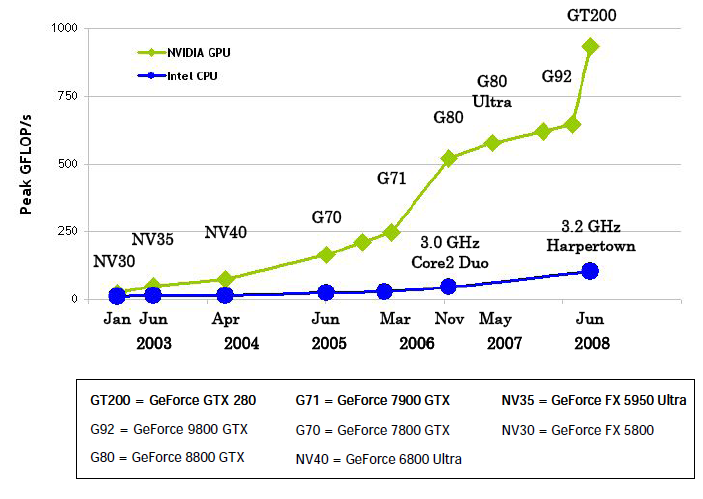
CUDA este un model paralel de programare si un mediu software proiectat sa raspunda la aceasta provocare fiind in acelasi timp acesibil programatorilor familiarizati cu limbaje standard de programare cum este limbajul C.

La baza CUDA stau 3 abstractii cheie – o ierarhie de grupuri de fire de executie, memorie partajata si sincronizare de tip bariera – expuse programatorului ca un set minimal de extensii ale limbajului C.

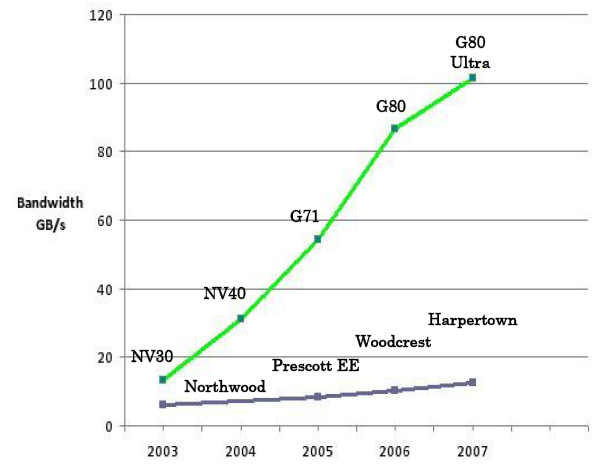
Aceste abstractii ofera paralelism de mare granularitate la nivelul datelor si al firelor de executie, imbinat cu paralelism de granularitate mica la nivelul sarcinilor de executat si datele lor. Acestea ghideaza programatorul sa partitioneze problema in sub-probleme cu granularitate mare care apoi pot fi rezolvate independent in paralel, si apoi in bucati mai mici care pot fi rezolvate cooperativ in paralel. O astfel de descumpunere pastreaza expresivitatea limbajului prin permiterea firelor de executie sa rezolve cooperativ fiecare sub-problema, si in acelasi timp permite scalabilitate transparenta deoarece fiecare sub-problema poate fi programata a fi rezolvata pe orice din core-urile disponibile ale procesorului: un program CUDA compilat poate fi executat pe orice numar de core-uri si numai sistemul de executie(runtime) trebuie sa cunoasca numarul fizic de procesoare.

# Arhitectura GPU-ului expusa de catre CUDA

Impins de nevoia pentru grafica 3D in timp real, de inalta definitie, GPU-ul a evoluat intr-un procesor cu un grad inalt de paralelism, „multithread”, „manycore” cu multa putere de calcul si o latime de banda a memoriei foarte mare, dupa cum se vede in figura urmatoare:

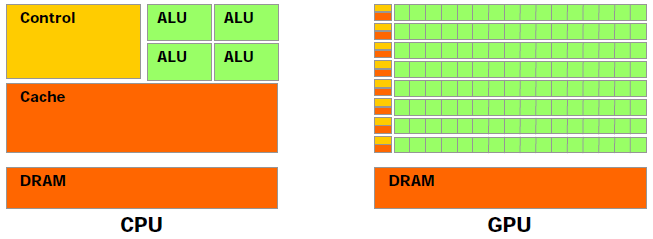


***Puterea de procesare a GPU-urilor comparata cu ce a CPU-urilor***



***Latimea de banda a memoriei de pe GPU comparata cu cea CPU-urilor***

Motivul pentru aceasta discrepanta intre capabilitatile de virgula mobila intre CPU-uri si GPU-uri este faptul ca GPU-ul este specializat pentru calcule intense, cu un grad mare de paralelism – exact pentru acest lucru a fost creata banda de redare grafica – si este proiectat in asa fel incat mai multi tranzistori sunt dedicati procesarii de date decat pentru controlul executiei sau cache, asa cum este reprezentat schematic in urmatoarea figura.



***GPU-ul dedica mai multi tranzistori procesarii de date***

GPU-ul este in mod deosebit potrivit pentru probleme care pot fi exprimate ca prelucrari paralele de date – acelasi program este executat pe mai multe elemente de date in paralel. Deoarece acelasi program este executat pentru fiecare element de date, exista mai putin nevoia pentru control sofisticat al executiei; si pentru ca este executat pe multe elemente de date cu intensitate mare aritmetica, latenta accesului la memorie poate fi ascunsa cu calcule in loc de cache-uri de date mari.

Procesarile paralele pe date mapeaza elementele de date pe fire de executie paralele. Multe aplicatii care proceseaza volume mari de date pot folosi un model de programare paralel pe date pentru a mari viteza de calcul. In redarea 3D, seturi mari de pixeli si vertecsi sunt mapate pe fire de executie paralele. Similar, aplicatii de procesare imagini si filme cum ar fi post-procesarea imaginilor redate, codarea/decodarea video, scalarea imaginilor, stereoscopie, recunoastere de modele(pattern recognition) pot mapa blocuri de imagine si pixeli pe fire de executie paralele. De altfel, multi algoritmi ce nu tin de redarea si procesarea imaginilor pot fi accelerati prin procesare paralela de date, de la procesarea semnalelor sau simulari de fizica la calcule financiare sau biologice.

Modelul de programare CUDA este foarte potrivit pentru a expune capabilitatile de paralelism ale GPU-urilor. Cea mai noua generatie de GPU-uri NVIDIA, bazate pe arhitectura Tesla, suporta modelul de programare CUDA si accelereaza mult aplicatiile CUDA.

# CUDA – Extensie a limbajului C

CUDA extinde limbajul C prin introducerea de functii C numite *kernel*, care apelate, sunt executate de N ori in paralel pe N *fire de executie CUDA*, spre deosebire de functii C obisnuite.

Un kernel este definit folosind specificatorul de declaratie **\_\_global\_\_** si cu numarul de fire de executie CUDA pentru fiecare apel specificat folosind o noua sintaxa cu **<<<...>>>**:

// Definire kernel

\_\_global\_\_ void vecAdd(float\* A, float\* B, float\* C)

{

}

int main()

{

// Apel kernel

vecAdd<<<1, N>>>(A, B, C);

}

Fiecare din firele de executie care ruleaza un kernel primeste un *thread ID* unic care este accesibil in interiorul kernel-ului prin variabila predefinita **threadIdx**. Sa luam ca exemplu codul pentru adunarea a doi vectori A si B de lungime N cu rezultatul in C:

\_\_global\_\_ void vecAdd(float\* A, float\* B, float\* C)

{

int i = threadIdx.x;

C[i] = A[i] + B[i];

}

int main()

{

// Apel kernel

vecAdd<<<1, N>>>(A, B, C);

}

Fiecare fir de executie ce ruleaza **vecAdd()**  executa adunarea unei singure perechi.

Pentru convenienta, **threadIdx** este un vector de 3 componente, astfel incat firele de executie sa poata fi identificate folosind indecsi uni-,bi- sau tridimensionali formand un *bloc de fire de executie* uni-,bi- sau tridimensional. Aceasta abordare ofera o modalitate naturala de a invoca calculul elementelor dintr-un domeniu cum este un vector, o matrice, sau un camp. Ca exemplu avem adunarea a doua matrice A si B de dimensiune NxN cu rezultatul in C:

\_\_global\_\_ void matAdd(float A[N][N], float B[N][N],

float C[N][N])

{

int i = threadIdx.x;

int j = threadIdx.y;

C[i][j] = A[i][j] + B[i][j];

}

int main()

{

// Apel kernel

dim3 dimBlock(N, N);

matAdd<<<1, dimBlock>>>(A, B, C);

}

Indexul unui fir de executie si IDul sau sunt intr-o relatie simpla: pentru un bloc unidimensional, sunt acelasi lucru; pentru un bloc bidimensional de dimensiune *(Dx, Dy),* IDul unui fir de executie de index *(x, y)* este *(x + y Dx)*; pentru un bloc tridimensional de dimensiune *(Dx, Dy, Dz),* IDul unui fir de executie de index *(x, y, z)* este *(x + y Dx + z Dx Dy*).

Firele de executie dintr-un bloc pot coopera intre ele prin partajarea datelor prin *memorie partajata* si prin sincronizarea executiei lor. Mai exact, se pot specifica puncte de sincronizare in kernel prin apelul functiei intrinseci **\_\_syncthreads()** ce actioneaza ca o bariera la care toate firele de executie trebuie sa astepte inainte de a continua.

Pentru o cooperare eficienta, memoria partajata trebuie sa fie de latenta scazuta langa fiecare core, ca un cache L1, **\_\_syncthreads()** trebuie sa fie un apel cu overhead mic si toate firele de executie ale unui bloc trebuie sa fie pe acelasi core de procesor. Numarul de fire de executie per bloc este astfel restrictionat de resursele de memorie ale unui core. Pe arhitectura NVIDIA Tesla, un bloc poate contine pana la 512 fire de executie.

# CUDA – Blocuri de fire de executie

Un kernel poate fi executat de mai multe blocuri de fire de executie de aceasi dimensiune astfel incat numarul total de fire de executie sa fie egal cu numarul de fire de executie per bloc inmultit cu numarul de blocuri. Aceste multiple blocuri sunt organizate intr-o *matrice* uni-dimensionala sau bi-dimensionala ca in figura urmatoare. Dimensiunea matricei este specificata de primul parametru al sintaxei **<<<...>>>.** Fiecare bloc din matrice poate fi identificat printr-un index uni- sau bi-dimensional accesibil prin intermediul variabilei predefinite **blockIdx**. Dimensiunea blocului este accesibila prin variabila predefinita **blockDim**. Exemplul anterior devine:

\_\_global\_\_ void matAdd(float A[N][N], float B[N][N],

float C[N][N])

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

int j = blockIdx.y \* blockDim.y + threadIdx.y;

if (i < N && j < N)

C[i][j] = A[i][j] + B[i][j];

}

int main()

{

// Apel kernel

dim3 dimBlock(16, 16);

dim3 dimGrid((N + dimBlock.x – 1) / dimBlock.x,

(N + dimBlock.y – 1) / dimBlock.y);

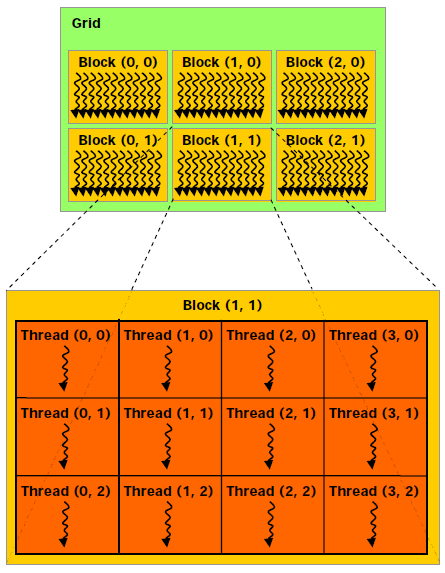
matAdd<<<dimGrid, dimBlock>>>(A, B, C);

}

Dimensiunea blocului de 16x16 = 256 fire de executie a fost aleasa relativ arbitrara, si matricea este creata cu destule blocuri pentru a avea un fir de executie per element de matrice.

Blocurile de fire de executie trebuie sa se execute independent. Este posibil sa fie executate in orice ordine, in paralel sau serial. Aceasta cerinta de independenta permite blocurilor de fire de executie sa fie programate in orice ordine pe orice numar de core-uri, permitand programatorilor sa creeze cod scalabil.

Numarul de blocuri de fire de executie dintr-un grid este in general dictat de dimensiunea datelor de procesat si nu de numarul de procesoare din sistem, pe care il poate depasi cu mult.

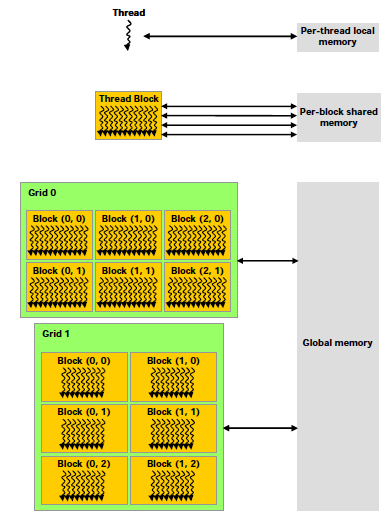


***Fire de executie organizate in blocuri***

# CUDA – Ierarhia memoriei

In ce priveste memoria fiecare fir de executie are acces la o memorie proprie privata, o memorie partajata per bloc si in fine, o memorie globala.

Mai exista doua zone de memorie protejate la scriere accesibile tuturor firelor de executie, memoria constanta si memoria de texturare. Memoria globala, constanta si cea de textura este accesibila si persistenta din toate kernel-urile lansate de aceasi aplicatie.



Memoria gazda (cea a CPU-ului) si memoria dispozitivului(GPU-ul care functioneaza ca un coprocesor) sunt separate, existand insa metode de transfer din una in alta, pentru datele de intrare si cele de iesire de exemplu, asemanator cu apelurile din userspace spre kernel-ul sistemului de operare.

Acest model de programare poate fi vazut ca un model eterogen de programare in care o parte din cod se executa pe gazda(CPU), serial, si alta pe alt dispozitiv(GPU), paralel.

# CUDA – Arhitectura SIMT

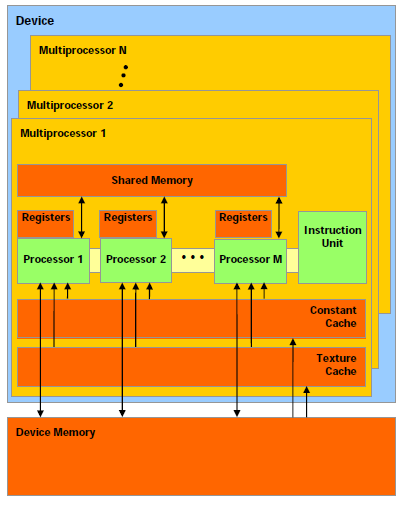
Arhitectura Tesla este construita in jurul unui vector scalabil de SM(Streaming Multiprocessors) multi-fir de executie. Cand un program CUDA de pe gazda invoca o matrice de kernel, blocurile matricei sunt enumerate si distribuite pe multiprocesoarele ce au destule resurse pentru a le executa. Firele de executie ale unui bloc se executa concurent pe un multiprocesor. Pe masura ce se termina blocurile, noi blocuri sunt lansate pe multiprocesoarele libere.

Un multiprocesor consista din 8 core-uri de Procesoare Scalare(SP), o memorie partajata, doua unitati pentru functii specale, o unitate de executie pt mai multe fire de executie. Mltiprocesorul creeaza, gestioneaza si executa concurent fire de executie in hardware cu overhead 0 de programare(scheduling). Implementeaza functia **\_\_syncthreads()** ca bariera de sincronizare cu o singura instructiune. Bariere rapide de sincronizare si creere usoara de fire de executie permit paralelism cu granularitate mare, prin descompunerea problemei si asocierea unui thread pentru fiecare element de date.

Pentru a gestiona sute de fire de executie ce ruleaza diferite programe, mltiprocesorul foloseste o noua arhitectura numita SIMT(single-instruction, multiple-thread). Multiprocesorul SIMT creaza, gestioneaza, programeaza si executa firele de executie in grupuri de 32 de fire paralele numite *„warp”*(termen ce provine din tesut, prima tehnologie paralela). Firele individuale ce compun un warp SIMT pornesc de la aceasi adresa dar pot executa salturi conditionate si sa execute cai diferite.

Cand un multiprocesor are de executat unul sau mai multe blocuri, le imparte in warp-uri care vor fi programate de unitatea SIMT. Blocurile se impart in warps mereu la fel, consecutiv. Warp-urile se executa paralel cel mai bine cand firele din acel warp executa acelasi cod, salturile conditionate afectand performanta in interiorul aceluias warp.

Arhitectura SIMT este asemanatoare cu arhitectura SIMD(Single Instruction, Multiple Data). O diferenta majora este ca arhitecturile SIMD expun dimensiunile SIMD-ului software-ului in timp ce instructiunile SIMT specifica executia si salturile unui singur fir de executie. In contrast cu masinile vectoriale SIMD, SIMT permite programatorilor sa scrie cod paralel la nivel de fir de executie pe fire de executie independente, scalare cat si cod paralel la nivel de date pentru fire de executie coordonate. Codul poate fi optimizat prin asigurarea faptului ca firele de executie dintr-un warp nu iau cai diferite. Software-ul SIMD trebuie sa gestioneze manual divergenta. Arhitectura este ilustrata in figura:



# Interoperabilitate CUDA si C++

Deoarece limbajul pentru CUDA este un superset strict al limbajului C, interoperabilitatea se obtine prin acelasi mecansim ca si cu limbajul C de baza. Cod scris in C++ poate apelat metode scrise in C extins la fel ca si metode simple in C.

Iata un exemplu din SDK-ul CUDA. Intai declaram o interfata in C pentru a fi apelata de codul in C++:

extern "C"

void runTest(const int argc, const char\*\* argv, char\* data, int2\* data\_int2, unsigned int len)

Functia astfel declarata, cu linkare stil C, poate apela kernelul de pe GPU:

\_\_global\_\_ void kernel( int\* g\_data );

Mai multe detalii in capitolul despre implementare.

# Framework-ul WxWidgets

Framework-ul WxWidgets(fostul WxWindows) a fost creat pentru a permite dezvoltarea de aplicatii cu interfara grafica independente de sistemul de operare si care ruleaza nativ pe toate platformele suportate, cu look & feel nativ.

# O scurta istorie a framework-ului WxWidgets

In 1992, la Institutul de Inteligenta Artificiala de la Universitatea din Edinburgh, proiectul wxWidgets este initiat de Julian Smart. Julian dezvolta un utilitar de tip CASE numit Hardy ce trebuia sa ruleze atat pe Windows cat si pe Unix-uri cu interfata grafica bazata pe X-server. Software-ul comercial existent pentru o aplicatie cross-platform disponibil in acel moment era considerat ca fiind prea scump pentru a fi folosit la dezvoltarea unui proiect intern. Astfel, singura alternativa era sa se dezvolte unul propriu. wxWidgets(w de la Windows, x de la X) a pornit cu suport pentru XView si MFC 1.0 si a fost lansat pe Internet. Destul de curand a devenit evident ca XView creat de Sun Microsystems era sortit pieirii asa ca a fost creata o versiune pentru Motif. Utilizatorii de Borland C++ au inceput sa ceara o versiune care sa nu fie dependenta de MFC, asa ca versiunea de Windows a fost rescrisa pentru a folosi API-ul nativ de Windows.

De-a lungul timpului o mica dar entuziasta comunitate de utilizatori wxWidgets a fost stabilita si a fost creata o lista de discutii. Contributiile si solutiile la probleme erau trimise aici, cel mai multe contributii fiind facute la versiunile de Mac si Xt. Pe parcurs, wxWidgets a adunat din ce in ce mai multi utilizatori: comerciali, din cadrul guvernului si mai ales companii care au descoperit ca era un produs mai bun si oferea suport mai bun decat variantele comerciale.

In 1995, Markus Holzem a lansat versiunea sa de wxWidgets pentru Xt(X toolkit). Asta insemna ca wxWidgets putea fi folosit pe sisteme cu X server fara a instala Motif care era un produs comerical.

In 1996, wxWidgets a stagnat un pic. Julian a parasit AIAI si se lansau putine versiuni noi. Avansul aparent de neoprit al Java incepuse sa creeze o aura de indoiala in jurul bibliotecilor C++. Dupa multe contributii, codul wxWidgets arata „obosit” si avea nevoie de o reformulare. La inceputul anului 1997 era momentul hotararii de a reface wxWidgets sau de a il abandona; dupa o intalnire intre Julian Smart si Mrkus Holzem(creatorul versiunii pentru Xt) au hotarat ca merita continuat. Astfel ideile si codul pentru wxWidgets 2 au inceput sa ia forma.

In 1997 intr-un efort de a produce un mediu desktop standard pentru Linux se lucra la GNOME. GNOME se baza pe GTK+(construit peste X11) si se parea ca GTK+ avea sa devina standardul in universul Linux. Exista o problema: GTK+ era scris in C si nu exista decat un wrapper subtire si neportabil de C++ pentru el. In august 1997, Wolfram Gloger a sugerat ca wxWidgets sa fie portat pentru GTK+. Robert Roebling a devenit dezvoltatorul principal al wxGTK si la inceputul anului 1998 au aparut primele versiuni alfa. In acest timp Vladim Zeitlin devenise unul dintre cei mai activ dezvoltatori si inca mai este.

In 1998, Julian Smart a inceput o noua versiune de wxMotif; Xt a fost abandonat intrucat existau alte alternative gratuite: wxGTK si wxMotif cu clona gratuita, Lesstif.

In septembrie 1998, Stefan Csomor a inceput o noua versiune a wxMac 2 bazata partial pe codul scris intitial de Greg Whitehead.

Versiunea 2.1.1 a wxWidgets a adus wxHTML, clasele pentru afisare cross-platform de documente HTML ale lui Vaclav Slavik.

La mijlocul anului 2000 Robert Roebling a lansat wxDesigner un editor si utilitar de dezvoltare rapida comercial pentru wxWidgets. Tot in 2000 a inceput wxUniversal, setul de componente wxWidgets pentru platformele care nu au un set propriu, creat in principal de Vadim Zeitlin si sprijinit financiar de SciTech Software, Inc.

In 2001 si 2002, Julian Smart si Robert Roebling au lucrat la versiunea wxX11 care foloseste doar wxUniversal si necesita doar Linux si X11, fiind potrivita pentru sisteme cu resurse limitate. In octombrie 2002 au reusit sa compileze wxX11 pe un Linux pe iPAQ. De asemenea wxGTK a fost adaptat la GTK+ 2 cu suport Unicode, deschizand calea catre alte piete.

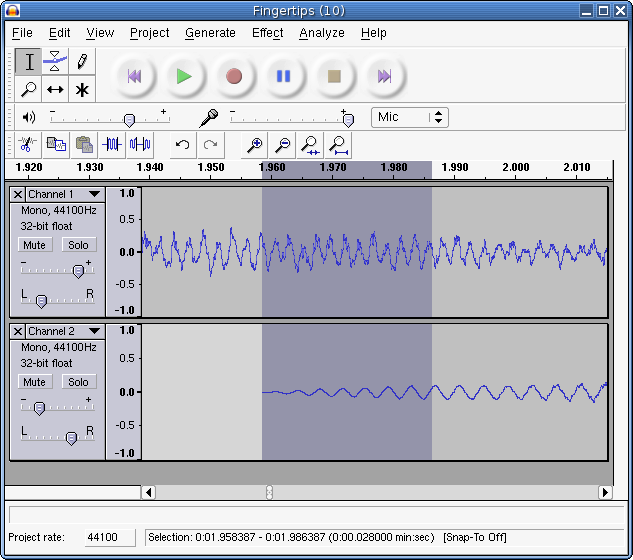
In 2003 s-a inceput lucrul la versiunea pentru WindowsCE. Versiunea pentru OS/2 s-a imbunatatit incet dar sigur de-a lungul ultimilor ani.

# Mai multe despre WxWidgets

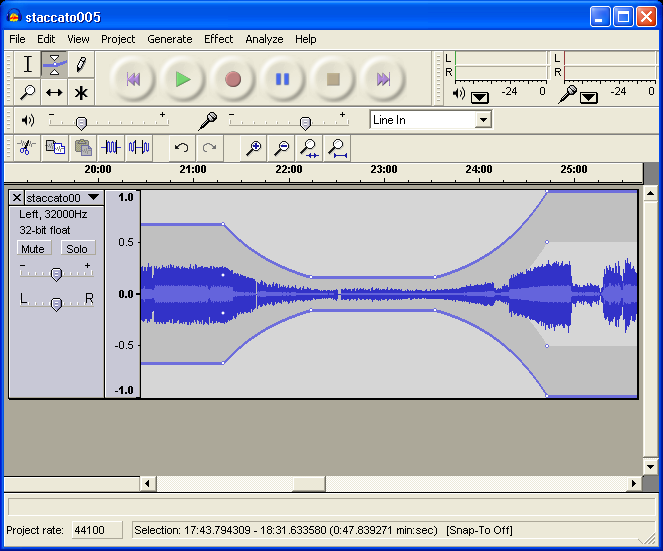
Desi wxWidgets este etichetat ca o biblioteca de dezvoltare aplicatii cu interfata grafica(GUI), este in realitate mai mult de atat oferind functionalitati care sunt utile pentru multe aspecte ale dezvoltariide aplicatii. wxWidgets trebuie sa fie astfel construit pentru ca intreaga aplicatie trebuie sa fie portabila pe diferite platforme, nu numai partea de interfata grafica. wxWidgets ofera clase pentru manipulare fisiere, fluxuri de date, fire de executie, comunicatie interprocese, acces la baza de date si multe altele.

De ce am ales wxWidgets? Spre deosebire de alte framework-uri, ca MFC de exemplu, wxWidgets este prin excelenta independent de platforma(sau cel putin cross-platform). wxWidgets ofera un API care este acelasi, sau aproape acelasi, pe toate platformele. Asta inseamna ca putem scrie o aplicatie pe Windows, de exemplu, si cu cateva schimbari minore(daca e nevoie) sa o recompilam pe Linux sau Mac OS X. Acesta este un avantaj major fata de alte API-uri care sunt specifice unei platforme si ar necesita rescrierea partii de GUI pentru fiecare platforma pe care ar rula produsul. In plus, wxWidgets permite compatibilitatea cu viitoare versiuni ale sistemelor de operare pe cae ruleaza, fara modificari din partea programatorului.

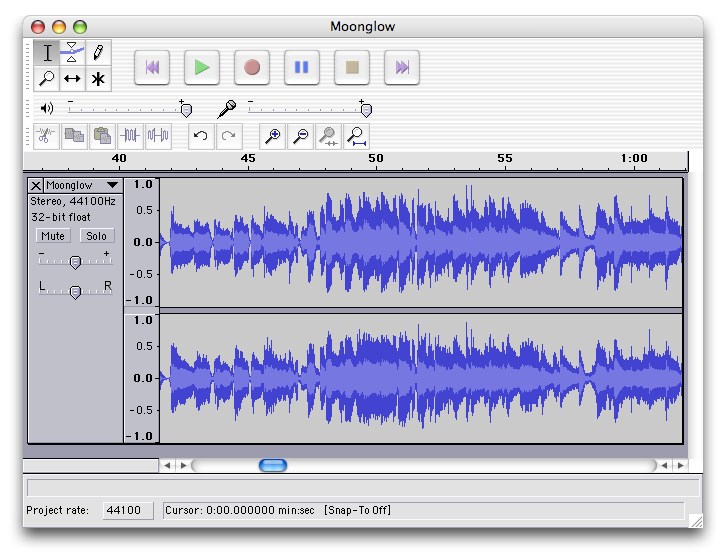
Un alt aspect important care distinge wxWidgets de alte framework-uri este faptul ca ofera pe fiecare platforma pe care ruleaza aspectul nativ. Alte framework-uri folosesc aceleasi elemente de interfata pe toate platformele, poate cu o tema aplicata pentru a simula aspectul nativ. Prin contrast, wxWidgets foloseste elemente de interfata native de cate ori este posibil astfel incat nu numai ca aplicatia arata nativ pe toate platformele dar si este. Aceasta este foarte important pentru utilizatorul pentru care si cele mai mici diferente intre felul in care se comporta o aplicatie si felul in care se asteapta utilizatorul poate induce o experienta neplacuta acestuia. Pentru a exemplifica, iata o aplicatie scrisa cu wxWidgets numita „Audacity”, un editor audio, ce ruleaza atat pe Windows cat si pe Linux si Mac OS X.



*Audacity pe Linux*

**

*Audacity pe Windows*

**

*Audacity pe MacOS X*

Dupa cum se observa „Audacity” apare ca o aplicatie nativa pe toate cele trei platforme.

De ce nu am folosi Java(pentru interfata grafica cel putin)? Desi Java este foarte bun pentru aplicatii web, nu este intotdeauna cea mai buna alegere pentru aplicatii desktop. In general, aplicatiile scrise in C++ ce folosesc wxWidgets sunt mai rapide, au un aspect nativ si sunt mai usor de instalat pentru ca nu depind de prezenta masinii virtuale Java. C++ ofera in plus acces la functionalitatea de nivel scazut si este mai usor de integrat cu cod deja existent in C si C++. In plus, a scrie un engine 3D in Java l-ar priva de posibilitatea da a atinge performanta maxima posibila. O alternativa, mai dificila ce-i drept, ar fi scrierea interfetei in Java si interfatarea cu restul codului prin JNI. Din aceste motive si multe altele wxWidgets este superior unui framework Java de redare a interfetei grafice.

De asemenea de remarcat la wxWidgets este faptul ca este un proiect open-source. In mod natural, asta inseamna ca utilizarea sa este gratuita dar are si alte implicatii. Software-ul open-source are obiceiul de a depasi in longevitate echivalentul lor comercial. Ca dezvoltator ce foloseste wxWidgets ai siguranta ca nu va dispare niciodata codul pe care te bazezi. Poti rezolva orice problema singur prin modificarea codului sursa. De asemenea comunitatea din spatele wxWidgets e mult mai activa decat in cazul echipei de suport ce sta la dispozitia utilizatorului unui produs comercial. Utilizatorii wxWidgets includ AOL, AMD, CalTech, NASA, Xerox si multi altii.

In continuare vom arunca o privire asupra arhitecturii generale a wxWidgets.

# Arhitectura WxWidgets

Pe scurt arhitectura wxWidgets arata astfel:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| API wxWidgets | | | | | | | | |
| Portare wxWidgets | | | | | | | | |
| wxMSW | wxGTK | wxX11 | wxMotif | wxMac | wxCocoa | wxOS2 | wxPalmOS | wxMGL |
| API specific platformei | | | | | | | | |
| Win32 | GTK+ | Xlib | Motif/  Lesstif | Carbon | Cocoa | PM | Palm OS API-urile Protein | MGL |
| Sistem de operare | | | | | | | | |
| Windows/  Windows CE | Unix/Linux | | | Mac OS 9/  Mac OS X | Mac OS X | OS/2 | Palm OS | Unix |

Utilizatorul foloseste direct API-ul wxWidgets de nivel cel mai inalt, fara a se preocupa de ce portare in sine sta la baza, decat daca exista probleme de trecere de la una la alta mai ales daca unul din scopurile sale este de a construi o aplicatie cross-platform. Se observa multitutdinea de platforme pe care ruleaza wxWidgets.

Mai putem adauga si ca wxWidgets nu este disponibil numai din C++, ci exista versiuni si pentru Python, Perl si C#. In plus, wxWidgets a fost portat si pe platforme mobile.

# Algoritmul AES

Algoritmul *Rijndael* a fost ales in 2001 ca standard de criptare devenind AES(Advanced Encryption Standard). Algoritmul a fost propus de doi matematicienii belgieni, Joan Daemen si Vincent Rijmen. Numele algoritmului este o combinatie numelor lor. In continuare vom prezenta pasii acestui algoritm si cum se poate paraleliza.

# Specificatia AES

Algoritmul Rijndael(AES) este un algoritm cifru bloc iterativ cu lungimea blocului si a cheii variabila. Lungimea blocului si a cheii pot fi specificate independent ca 128, 192 sau 256 de biti.

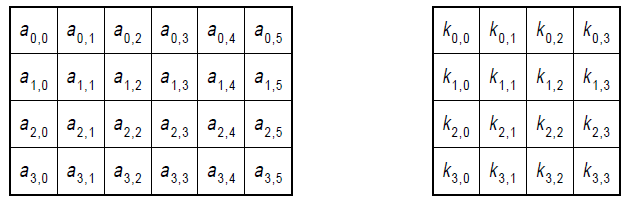
# Starea, cheia si numarul de runde

Diferitele transformari opereaza pe un rezultat intermediar, numit *Stare*

Starea poate fi vizualizata ca o matrice de octeti. Matricea are patru randuri, numarul de coloane notat **Nb** si este egal cu lungimea blocului impartita la 32.

Cheia cifrului poate fi simimar vizualizata ca o matrice cu patru randuri. Numarul de coloane ale cheii de cifru este notat **Nk** si este egal cu lungimea cheii impartita la 32.

Aceste reprezentari sunt prezentate in figura urmatoare.



***Exemplu de Stare si Cheie pentru Nb = 6 si Nk = 4***

In unele instante, aceste blocuri pot fi considerate ca un vector uni-dimensional de vectori de 4 octeti, unde fiecare vector este coloana corespunzatoare din reprezentarea sub forma de matrice. Acesti vectori au deci lungimi de 4,6 sau 8 si indicii in gamele 0..3, 0..5 sau 0..7. Vectorii de 4 octeti pot fi numiti si cuvinte(32 de biti).

Unde este necesar sa se specifice cei patru octeti individuali din vectorul de 4 octeti sau cuvant notatia (a, b, c, d) va fi folosita unde a, b,c si d sunt octetii de la pozitiile 0, 1, 2 si 3 din coloana, vectorul sau cuvantul luat in considerare.

Intrarea si iesirea folosite de Rijndael la interfata sa externa sunt considerate a fi vectori de octeti numerotati de la 0 la 4\***Nb**-1. Aceste blocuri au lungimi de 16, 24 sau 32 de octeti iar indicii sunt in gama 0..15, 0..23 sau 0..31. Cheia cifrului este considerata a fi un vector de octeti numerotati de la la 4\***Nk**-1. Aceste blovuri au de asemenea lungimi de 16, 24 sa 32 de octeti.

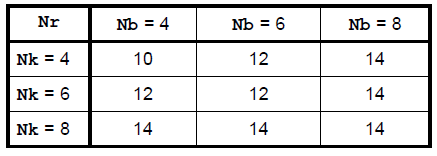
Octetii de intrare („plaintext”-ul daca se foloseste modul ECB) sunt mapati in octetii de stare in ordinea a0,0, a1,0, a2,0, a3,0, a0,1, a1,1, a2,1, a3,1, a4,1 ... iar octetii cheii sunt mapati in matrice in ordinea k0,0, k1,0, k2,0, k3,0, k0,1, k1,1, k2,1, k3,1, k4,1 … La sfarsitul operatiei de cifru, iesirea operatiei este extrasa din stare luand octetii starii in aceasi ordine.

Astfel, daca indexul unidimensional al unui octet in interiorul unui bloc este *n* si indexul bidimensional e *(i,j)*, avem:

*i* *n* mod 4 ; *j* *n* / 4; *n* *i* 4 \* *j*

Mai mult, indexul *i* este de asemenea si numarul octetului intr-o secventa de 4 octeti sau cuvant si *j* este indexul pentru acel vector sau cuvant in blocul din care face parte.

Numarul de runde este notat **Nr** si depinde de valorile **Nb** si **Nk**, conform tabelului:



# Rundele

O runda este compusa din patru transformari diferite, in pseudo cod ar arata astfel:

Round(State,RoundKey)

{

ByteSub(State);

ShiftRow(State);

MixColumn(State);

AddRoundKey(State,RoundKey);

}

Ultima runda este un pic diferita:

FinalRound(State,RoundKey)

{

ByteSub(State) ;

ShiftRow(State) ;

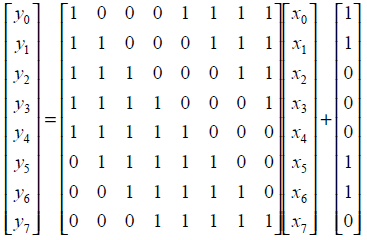
AddRoundKey(State,RoundKey);

}

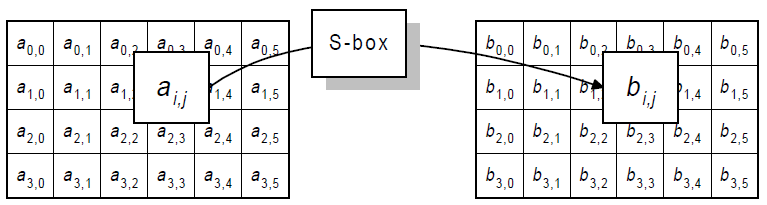
In aceasta notatie, „functiile” (Round, BytesSub, ShiftRow, ...) opereaza pe matrice (State, RoundKey) catre care primesc pointeri. Se poate vedea ca in runda finala este eliminat pasul MixColumn. Transformarile componente sunt descrise in urmatoarele sectiuni.

# Transformarea ByteSUb

Transformarea ByteSub este substitutie non-liniara de octeti, aplicata pe fiecare din octetii starii independent. Tabela de susbtitutie (S-box) este inversabila si este compunerea a doua transformari(in figura urmatoare xi si yi sunt biti):



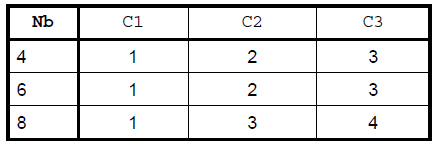
Transformarea este aplicata asupra fiecarui octet al starii, independent:



Inversul lui ByteSub este o substitutie de octeti in care se aplica tabela inversa.

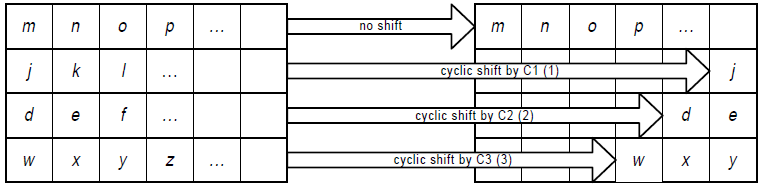
# Transformarea ShiftRow

In ShiftRow, randurile starii sunt deplasate ciclic cu diferite offesturi. Randul 0 nu este deplasat, randul 1 este deplsat C1 octeti, randul 2 este deplasat C2 octeti iar randul 3, C3 octeti. Offeset-urile de deplasare C1, C2 si C3 depind de lungimea blocului, **Nb**. Diferitele valori sunt specificate in tabel:



Operatia de deplasare a randurilor starii este notata *ShiftRow(State) .*

Figura urmatoare ilustreaza operatia ShiftRow asupra starii.

****

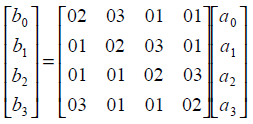
Inversa operatiei ShiftRow este o deplasare ciclica a randurilor de jos cu Nb-C1, Nb-C2 si Nb-C3 octeti astfel incat octetul de la pozitia *j* in randul *i* se muta la pozitia *(j + Nb – Ci) mod Nb*.

# Transformarea MixColumn

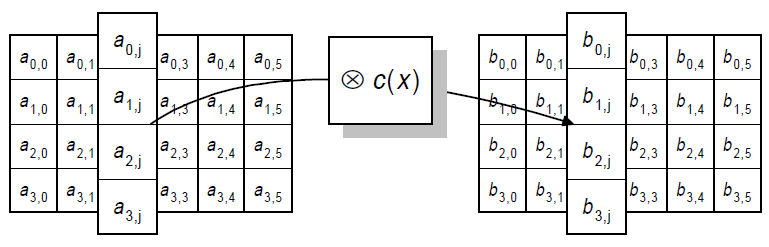
In MixColumn, coloanele starii sunt considerate polinoame ce sunt inmultitie modula x4 + 1 cu un polinom fix c(x) dat de:

c(x ) = ‘03’ x3 + ‘01’ x2 + ‘01’ x + ‘02’

Inmultirea polinoamelor poate fi scrisa ca o inmultire de matrice, fie b(x ) = c(x ) a(x ),



Aplicarea *MixColumn(State)* pe coloanele starii intermediare poate fi ilustrata astfel:



Inversul lui MixColumn este similar, fiecare coloana este transformata prin inmultirea cu un polinom *d(x)*, definit ca ( ‘03’ x3 + ‘01’ x2 + ‘01’ x + ‘02’ ) d(x ) = ‘01’ unde d(x ) = ‘0B’ x3 + ‘0D’ x2 + ‘09’ x + ‘0E’.

# Transformarea AddRoundKey

In acest pas, o cheie de runda este aplicata starii printr-un simplu sau-exclusiv(XOR) pe biti. Cheia de runda este derivata din cheia cifrului printr-un program de chei(key schedule). Lungimea cheii de runda este egala cu lungimea blocului **Nb**.

Transformarea ce consta din aplicarea unui XOR intre stare si cheia de runda este notata *AddRoundKey(State, RoundKey)* si poate fi ilustrata astfel:



Inversa lui AddRoundKey este ea insasi.

# Derivarea cheilor de runda

Cheile de runda sunt derivate din cheia de cifru printr-un program de chei (key schedule), acesta are doua componente: expandarea cheii si selectarea cheii de runda. Principiul de baza este urmatorul:

* Numarul total de biti ai cheii de runda este egal cu lungimea blocului inmultita cu numarul de runde plus 1. (e.g. pentru un bloc de 128 de biti si 10 runde, va fi nevoie de 1408 biti pentru cheia de runda).
* *Cheia de cifru* este expandata in *cheia extinsa*
* Cheile de runda sunt luate din aceasta cheie extinsa in felul urmator: prima cheie de runda consta in prinele **Nb** cuvinte, a doua urmatoarele **Nb** cuvinte si asa mai departe.

# Expansiunea cheilor

*Cheia extinsa* este un vector de cuvinte de 4 octeti si este notata *W[Nb\*(Nr+1)]*. Primele **Nk** cuvinte contin cheia de cifru; toate celelalte cuvinte sunt definite recursiv. Functia de expansiune a cheii depinde de valoarea lui **Nk**: exista o versiune pentru **Nk** mai mic sau egal cu 6 si una pentru **Nk** mai mare decat 6.

Pentru **Nk**  6, avem urmatorul algoritm:

KeyExpansion(byte Key[4\*Nk] word W[Nb\*(Nr+1)])

{

for(i = 0; i < Nk; i++)

W[i] = (Key[4\*i],Key[4\*i+1],Key[4\*i+2],Key[4\*i+3]);

for(i = Nk; i < Nb \* (Nr + 1); i++)

{

temp = W[i - 1];

if (i % Nk == 0)

temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];

W[i] = W[i - Nk] ^ temp;

}

}

In aceasta reprezentare, *SubByte(W)* este o functie care returneaza un cuvant de 4 octeti in care fiecare octet este rezultatul aplicatii S-box-ului Rijndael pe octetul de la pozitia corespunzatoare din cuvantul de intrare. Functia *RotByte(w)* aplica o permutare ciclica pe cuvantul de intrare astfel incat daca intrarea e (a,b,c,d) iesirea va fi (b,c,d,a).

Se observa ca primele **Nk** cuvinte sunt completate cu cheia de cifru. Fiecare cuvant *W[i]* este egal cu XOR intre cuvantul anterior *W[i-1]* si cuvantul cu **Nk** poztii mai inainte *W[i-Nk]*. Pentru cuvintele in pozitii multiplii de **Nk**, o transformare este aplicata lui *W[i-1]* inainte de XOR si o constanta de runda este de asemenea combinata cu XOR. Aceasta transformare consta intr-o deplasare ciclica a octetilor dintr-un cuvant (RotByte), urmata de aplicarea unui tabele de cautare pe toti cei patru octeti ai cuvantului (SubByte).

Pentru **Nk** > 6, algoritmul este:

KeyExpansion(byte Key[4\*Nk] word W[Nb\*(Nr+1)])

{

for(i = 0; i < Nk; i++)

W[i] = (key[4\*i],key[4\*i+1],key[4\*i+2],key[4\*i+3]);

for(i = Nk; i < Nb \* (Nr + 1); i++)

{

temp = W[i - 1];

if (i % Nk == 0)

temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];

else if (i % Nk == 4)

temp = SubByte(temp);

W[i] = W[i - Nk] ^ temp;

}

}

Diferenta fata de algoritmul anterior este ca pentru i – 4 multiplu de **Nk**, se aplica SubByte pe *W[i-1]* inainte de XOR.

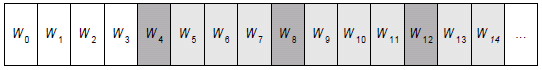
Constantele de runde sunt independente de **Nk** si sunt definite ca:

Rcon[i] =(RC[i], ‘00’, ‘00’, ‘00’) unde RC[i] este un element cu valoarea x(i – 1) astfel incat:

RC[1] = 1 (i.e. ‘01’), RC[i] = x (i.e. ‘02’) \* (RC[i-1]) = x(i-1)

# Selectarea cheii de runda

Cheia de runda *i* este data de cuvintele W[Nb\*i] pana la W[Nb\*(i+1)] din bufferul de chei de runda. Pentru **Nb** = 6 si **Nk** = 4 avem:



Programul de generare a cheilor poate fi implicit fara utilizarea explicita a vectorului *W[Nb\*(Nr+1)]*. Pentru implementari unde RAM-ul este in cantitati mici, cheile de runda pot fi calculate pe loc folosind un buffer de **Nk** cuvinte cu aproape niciun overhead computational.

# Cifrul

Cifrul Rijndael consta in:

* O adunare initiala a cheii de runda
* **Nr** - 1 runde
* O runda finala

In pseudocod C, algoritmul arata astfel:

Rijndael(State,CipherKey)

{

KeyExpansion(CipherKey,ExpandedKey) ;

AddRoundKey(State,ExpandedKey);

For( i=1 ; i<Nr ; i++ ) Round(State,ExpandedKey + Nb\*i) ;

FinalRound(State,ExpandedKey + Nb\*Nr);

}

Expansiunea cheilor poate fi facuta inainte si algoritmul Rijndael se poate raporta direct la cheia extinsa:

Rijndael(State,ExpandedKey)

{

AddRoundKey(State,ExpandedKey);

For( i=1 ; i<Nr ; i++ ) Round(State,ExpandedKey + Nb\*i) ;

FinalRound(State,ExpandedKey + Nb\*Nr);

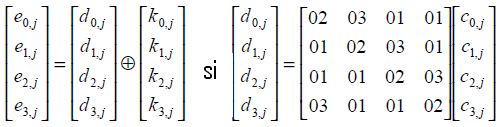
}

Cheia extinsa va fi mereu derivata din cheia de cifru si nu va fi niciodata specificata direct. Pe de alta parte, nu exista nicio restrictie in ce priveste cheia de cifru.

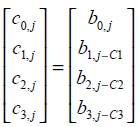
# Implementare pe un sistem pe 32 de biti

Diferitii pasi ai transformarilor de runda pot fi combinati intr-un singur set de cautari intr-un tabel, permitand implementari foarte rapide pe procesoare cu lungimea cuvantului de 32 de biti sau mai mult.

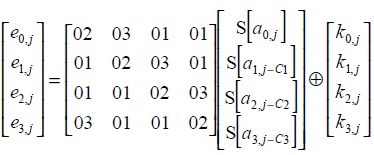
Notam o coloana a rezultatului de runda cu *e*. In aceasta notatie, *ai,j* denota un octet al unui rand al intrarii *a* unde *i* este randul si *j* coloana, *aj* denota coloana *j* a unei stari *a*. Pentru adunarea cheilor si transformarea MixColumn, avem:



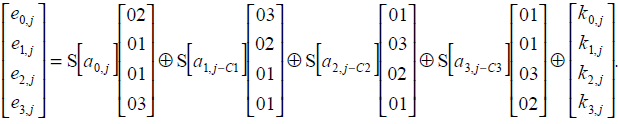
Pentru transformarile ShiftRow si ByteSub avem:

 unde *bi,j = S[ai,j]*

In aceasta expresie indicii de coloana trebuie considerati ca modulo **Nb**. Prin substitutie, expresiile de mai sus pot fi combinate in:

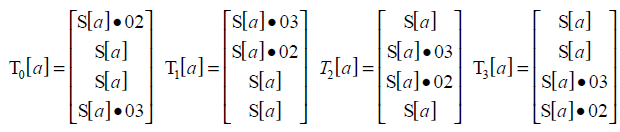


Inmultirea matricelor poate fi exprimata ca o combinaie liniara de vectori:



Factorii de multiplicare *S[ai,j]* ai celor patru vectori sunt obtinuti prin efectuarea unei cautari pe baza octetilor de la intrare *ai,j* in tabelul S-box S[256].

Definim tabelele T0 pana la T3 :



Acestea sunt 4 tabele cu 256 de cuvinte pe 32 de biti ce reprezinta pana la 4KB din spatiul total. Folosind aceste tabele transformarea unei runde se poate scrie:



Astfel, o implementare cu tabele de cautare cu 4 KB de tabele foloseste doar 4 cautari in atebl si 4 XOR-uri per runda.

Se poate vedea ca Ti[a] = RotByte(Ti-1[a]). Cu costul a 3 rotatii aditionale per runda per coloana, implementarea cu tabele de cautare poate fi realizata cu un singur tabel, i.e., cu o dimensiune a tabelei de doar 1KB. Avem:



Dimensiunea codului (relevanta pe dispozitive cu putina memorie – smartcarduri de exemplu) poate fi micsorata prin includerea de cod care sa genereze tabelele in loc de a stoca tabelele propriu-zise.

In runda finala, nu exista operatie de MixColumn. Asta duce la faptul ca tabelul S trebuie folosit in locul tabelelor T. Nevoia de tabele aditionale poate fi suprimata prin extragerea tabelei S din tabelele T prin mascare in timpul executiei ultimei runde.

Majoritatea operatiilor din expansiunea cheilor poate fi implementate ca XOR-uri pe cuvinte de 32 de biti. Transformarile aditionale sunt aplicari ale S-box-ului si o deplasare ciclica pe 8 biti. Aceste operatii pot fi implementate foarte eficient.

# Paralelism

Se vede ca exista un grad semnificativ de paralelism in transformarea de runda. Toate transformarile componente ale rundei actioneaza paralel pe octetii, coloanele si randurile starii.

In implementarea cu tabele de cautare, in principiu, toate cautarile pot fi efectuate in paralel. XOR-urile pot fi efectuate de asemenea in paralel in marea lor parte.

Expandarea cheilor este clar de o natura mai secventiala: valoarea lui *W[i-1]* trebuie cunoscuta pentru calculul lui *W[i]*. Totusi, in majoritatea aplicatiilor unde viteza este critica, expansiunea cheilor trebuie facuta numai o data pentru un numar mare de executii ale cifrului. In aplicatii unde cheia de cifru se schimba des (in extrema, o data per aplicare a cifrului), expansiunea cheii si rundele pot fi efectuate in paralel.

# Implementare eficienta folosind CUDA

Pana de curand singurele metode de a folosi arhitectura paralela a GPU-ului pentru programare generala erau prin API-urile destinate graficii 3D, OpenGL si Direct3D. Marile dezavantaje al utilizarii acestora pentru programare generala sunt legate de faptul ca acestea sunt orientate pe grafica. Maparea unei probleme de interes general pe domeniul graficii nu este mereu o sarcina simpla si performanta finala a solutiei este dramatic dependenta de maparea aleasa.

CUDA(si solutia de la AMD – Stream SDK) permite pentru prima data folosirea cat mai eficienta a GPU-uluu pentru a rezolva probleme de uz general.

Vom folosi in implementare toate optimizarile descrise in capitolele anterioare destinate procesoarelor pe 32 de biti. Mai concret, daca *a* este intrarea rundei iesirea ei va fi:

*ej* = *T0[ao,j]**⊕ T1[a1,j+1]**⊕ T2[a2,j+2]**⊕ T3[a3,j+3]**⊕ kj*

unde *T[]* este un tabel de cautare, ⊕ inseamna XOR si *kj* este o coloana a cheii de runda. Aceasta solutie necesita numai 4 cautari in tabela si 4 XORuri per coloana per runda.

Conform modelului de programare CUDA, dezvoltatorul are destula libertate in ce priveste tipul de memorii folosite, memoria partajata intre firele de executie, dimensiunea blocului de fire de executie si nu numai. Aceasta abordare conduce la o complexitate considerabila in ce priveste proiectarea unei aplicatii eficiente in comparatie cu modelul obisnuit de programare pe CPU.

Folosind optimizarile pentru procesoare pe 32 de biti mentionate in sectiunile anterioare se poate implementa cu patru tabele de cautare cu 256 de valori pe 32 de biti. Deoarece GPU-ul contine procesoare scalare nu este nevoie sa fie combinate operatiile in operatii pe vectori. In plus, disponibilitatea nativa a XOR-ului pe 32 de biti sporeste mult performanta posibila pe GPU pentru criptografie, XOR-ul fiind o operatie fundamentala. Astfel o singura runda AES poate fi executata pe o stare cu 16 cautari si 16 XOR-uri pe 32 de biti (4 iteratii ale ecuatiei de mai sus).

In primul rand, datele de intrare si cheia expandata sunt stocate in spatiul de memorie global al GPU-ului. Copierea datelor in si din memoria GPU-ului este cea mai scumpa operatie datorita latimii de banda a magistralei PCIExpress care este doar 3.2GB/s in comparatie cu 50GB/s pentru un G80(GeForce 8800). T-boxurile (tabelele de cautare) precalculate sunt incarcate in memoria constanta a GPU-ului. Memoria constanta are un mecanism de cache si tabelul de cautare se potriveste ca marime cu marimea cache-ului. Datele de intrare sunt impartite in bucati de 1024 de octeti care sunt criptati si decriptati complet in paralel. Un bloc de fire de executie CUDA este responsabil pentru procesarea unei bucati. Un bloc este alcatuit din 256 fire de executie pe GPU. Aceasta valoare a fost determinata empiric ca fiind cea care duce la implementarea cea mai rapida. Cand se ruleaza codul kernelului pe GPU performanta creste odata ce numarul de blocuri creste. Asadar blocuri de intrare mai mari duc la performanta mai mare.

Fiecare fir de executie al blocului de pe GPU calculeaza patru octeti al iesirii rundei de AES, deci patru fire de executie cripteaza/decripteaza intreaga stare. Fire de executie din acelasi bloc au nevoie sa partajeze informatii si sa acceseze frecvent informatia cheii expandate. Din acest motiv cheia este incarcata in memoria partajata impreuna cu portiunea datelor de intrare procesate de acel bloc. Mai precis doi vectori de 1KB de memorie partajata sunt folosite pentru intrare, citind date din primul si salvand rezultatul fiecarei runde in al doilea. Apoi vectorii sunt inversati pentru runda urmatoare. Aceasta strategie permite criptarea si decriptarea completa a bucatii de intrare fara a iesi din kernel. Se realizeaza operatia complet fara ca CPU-ul sa controlez secvential rundele AES. La sfarsitul calcului resultatul este scris inapoi in memoria globala si returnat CPU-ului.

Deoarece folosirea memoriei pardajata poate duce la conflicte de scriere/citire ce conduc la executii serializate, trebuie redusa folosirea acesteia, mai ales deoarece cheia extinsa trebuie accesata des. Pentru a beneficia de memoria partajata cheia trebuie incarcata la inceputul executiei fiecarui bloc de fire de executie. O alternativa este de a stoca cheia extinsa in memoria de texturare care are de asemenea cache. Un acces aleator la memoria de texturare este disponibil fiecaruia din firele de executie de pe GPU si CPU-ului pentru a stoca cheia. Dimensiunea limitata a cheii extinsa garanteaza faptul ca va fi stocata complet in cache-ul memoriei de texturare. In plus, trebuie observat ca aceasta aborare permite texturilor unor aplicatii third party sa fie criptate direct pe GPU.

# Descrierea bibliotecii de criptare AES folosind CUDA

Vom descrie in continuare arhitectura aplicatiei, folosind diagrame UML.

Aplicatia are doua parti mari: biblioteca de criptare/decriptare AES si o aplicatie cu interfata grafica pentru a oferi utilizatorului o metoda de a cripta/decripta fisiere.

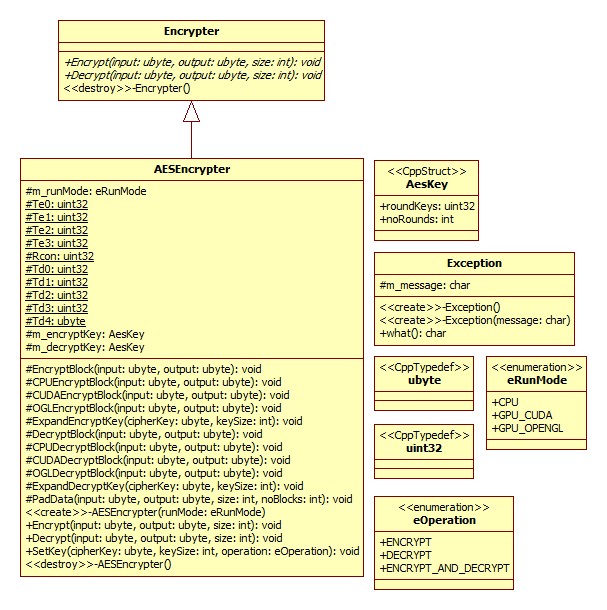
# Descrierea bibliotecii de criptare

Arhietctura aplicatiei de criptare/decriptare este reprezentata in schema urmatoare. Clasa abstracta Encrypter expune interfata ce trebuie implementata de orice clasa care efectueaza operatii de criptare/decriptare.

AESEncrypter implementeaza interfata Encrypter folosind criptare/decriptare AES ECB. Exista doua cai de rulare, pe CPU si pe GPU. Implementarea pe CPU este bazata pe biblioteca OpenSSL. Datele de intrare sunt impartite in blocuri de 128 de biti care sunt criptate cu cheie pe 128/192 sau 256 de biti.

Pentru varianta CUDA se apeleaza o functie C care la randul ei apeleaza kernel-ul CUDA.

Functiile care executa criptarea sunt CPUEncryptBlock() si CUDAEncryptBlock(). Decriptarea e facuta de CPUDecryptBlock() si CUDADecryptBlock().



Datele de intrare care nu sunt multiplu de 128 de biti(16 octeti) trebuie extinse cu 0-uri pana la un multiplu de 128 de biti. Pentru a putea decripta cu acelasi rezultat, trebuie pastrat „padding”-ul. Din aceste motive Fisierul de iesire va fi uneori mai mare decat cel de intrare. Functia care face „padding” la bufferul de intrare si cel de iesire este PadData().

Pentru paralelizare nu ne intereseaza decat modul ECB al cifrului AES. Modul CBC nu prezinta interes pentru ca nu ar fi posibila paralelizarea.

# Detalii de implementare a algoritmului AES folosind CUDA

Functiile C++ care folosesc GPU-ul sunt CUDAEncrypt()/CUDADecrypt(). Acestea apeleaza o functie externa, cu linkeditare C care la randul ei apeleaza kernel-ul CUDA.

Kernelul CUDA este definit intr-un fisier .cu ce va fi compilat compilatorul nVidia *nvcc*. Compilatorul *nvcc* compileaza partile de pe GPU cu un compilator intern iar cele de CPU cu un compilator specific fiecarei platforme.

Algoritmul este acelasi ca si pe CPU numai ca, acum se lucreaza cu memoria pe GPU. Cheia este expandata in continuare pe CPU, deoarece expandarea ei pe GPU nu ar aduce beneficii prea mari. Se copiaza apoi in memoria globala cheile de runda. Fiecare bloc CUDA primeste in memoria partajata blocul de date pe care trebuie sa il cripteze.

Implementarea folosind CUDA a fost detaliata in capitolele anterioare, cand s-au dat si specificatiile algoritmului AES si s-a stabilit cum ar fi paralelizat folosind CUDA.

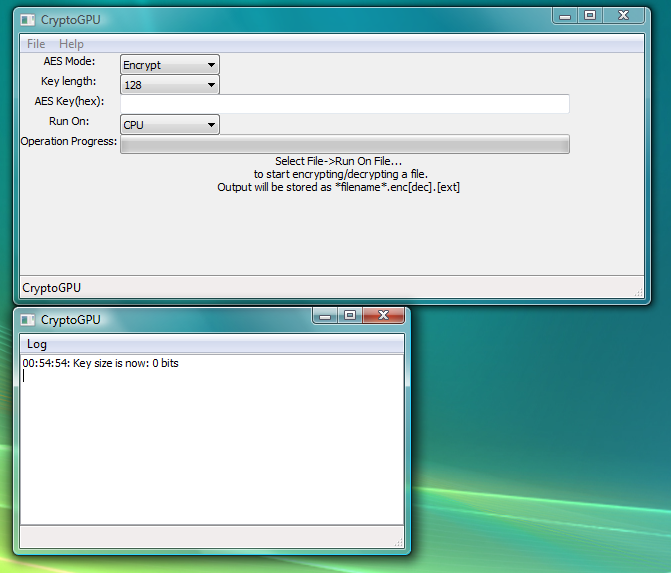
# Folosirea bibliotecii de criptare in aplicatii

Ca orice aplicatie scrise in C++ ea poate fi refolosita, fie avand acces doar la codul obiect sub forma de biblioteca partajata sau statica sau avand acces direct la codul ei, in functie de nevoile utilizatorului. Daca are nevoie sa modifice felul in care functioneaza o anumita parte a bibliotecii atunci va avea nevoie de acces la cod, daca nu este destul sa extinda clasa AESEncrypter.

Intern biblioteca nu foloseste wxWidgets insa aplicatia cu interfata grafica pe care am construit-o pentru a testa biblioteca de criptare foloseste acest toolkit pentru a crea un GUI relativ simplu de folosit pentru criptat si decriptat fisiere.

# Aplicatie demonstrativa

Aplicatia demonstrativa construita pentru a testa algoritmul arata astfel(pe Windows Vista – wxWidgets foloseste pe toate platformele toolkit-ul nativ):



Aplicatia ofera urmatoarele setari:

* Mod: criptare/decriptare
* Lungime cheie: 128/192/256
* Cheia cifrului in hexa

Pentru a cripta/decripta se seteaza cheia si parametrii si apoi din meniul „File”. Rezultatul va fi scris in acelasi director ca si fisierul de intrare si i se va adauga .enc sau .dec inainte de extensie in functie de operatia care s-a efectuat pe fisier.

Timpii de executie sunt mici chiar si pentru fisiere mari pe procesorul pe care s-a rulat (Core 2 Duo E8400) si a fost nevoie de o precizie mare in calcularea timpilor de executie.

# Masurarea performantelor bilbiotecii AES CUDA

Vom evalua in continuare performanta rularii pe GPU fata de CPU.

# Date experimentale

In urma rularii pe GPU, si anume pe un GeForce 9800 GTX(G92) s-au obtinut urmatoarele rezultate(AES-128).

G92 are 192 de core-uri(24 de multiprocesoare a cate 8 core-uri scalare).

|  |  |  |  |
| --- | --- | --- | --- |
| Dimensiune intrare | Timp pe CPU(ms) | Timp pe GPU(ms) | Speedup |
| 2KB | <1 | 0.26 | - |
| 512KB | 9 | 2.10 | 4.28 |
| 1MB | 19 | 3.74 | 5.08 |
| 4MB | 74 | 13.90 | 5.69 |
| 8MB | 148 | 27.34 | 5.41 |

Dupa cum se observa Speedup-ul este mai semnificativ pentru volume mai mari de date unde este mai bine folosita natura paralela a GPU-ului. Pentru volume mai mari de date.

# Concluzii

Paralelizarea algoritmului AES are fara indoiala beneficii; asa cum a fost mentionat si in capitolul de specificatii, AES ECB se poate paraleliza usor, blocurile si octetii putand fi criptati/decriptati in paralel odata ce a fost expandata cheia de cifru.

Sporul de viteza este nesemnificativ sau foarte mic in cazul volumelor mici de date pentru ca in aceste situatii nu se folosesc decat cateva din procesoarele disponibile pe GPU, pierzandu-se timp si cu transferul datelor de intrare in memoria GPU-ului si cu cele de iesire din memoria GPU-ului in RAM-ul sistemului.

Optimizarile posibile sunt legate de folosirea cat mai putin a memoriei partajate a GPU-ului si de reducerea transferurilor dintre memoria CPU si cea GPU.

Dupa cum se vede din rezultatele experimentale sporul de viteza este semnificativ pentru un volum mai mare date(8MB), acesta fiind apoi limitat de numarul de procesoare disponibil pe GPU si ne mai putand depasii acea valoare. Mai multe core-uri inseamna evident posibilitatea unui speedup mai mare.

# Concluzii ale proiectului. Dezvoltari ulterioare

In urma cu mai bine de 10 ani, o companie numita 3dfx lansa primul „accelerator grafic”. Acest chip era menit a rula aplicatii grafice 3D si 2D insa nu putea mentine sistemul de afisare pe cont propriu, fiind necesara existenta si unui chip normal de redare. Fara indoiala 3dfx au fost pionieri in GPU-uri la acea vreme, avand spre final, la sfarsitul anilor 1990, si primele GPU-uri multiprocesor(in sensul de mai multe GPU-uri pe aceasi placa video). Totusi programabilitatea in acel moment era limitata, mai ales ca 3dfx insista ca dezvoltatorii sa foloseasca API-ul propietar Glide pentru a obtine cele mai bune rezultate.

Din diferite motive, printre care lipsa de viziune si investitii exagerate in cercetare & dezvoltare, 3dfx a fost nevoita sa se inchida in 1999. In urma ei, au prosperat doua companii: nVidia si ATI. Nu este niciun secret ca nVidia a cumparat majoritatea propietatilor intelectuale ale defunctei 3dfx preluand si o buna parte din angajatii ei.

Primul GPU cu adevarat programabil a fost nVidia GeForce3 introdus in 2001. Acesta permitea modificarea pipeline-ului standard de redare grafica prin inroducerea de programe scrise intr-un limbaj specific GPU-ului numite „shaders”(Era vorba de HLSL pentru DirectX 8 la acea vreme). Deruland inainte, in 2006 aparea primul GPU cu adevarat programabil dintr-un punct de vedere mai general, GeForce8(G80). Din acest punct putem spune ca a inceput sa fie cu adevarat exploatata puterea GPU-ului si pentru altceva decat grafica.

Printre cele mai cunoscute aplicatii ce folosesc GPU-ul pentru accelerare este proiectul MIT, Folding@home, un proiect din domeniul biologiei si medicinei destinat studierii proteinelor si posibilelor combinatii ale acestora in speranta intelegerii naturii unor boli genetice si nu numai. De asemenea pagina dedicata CUDA a companiei nVidia ofera o multitudine de aplicatii si lucrari pe tema accelerarii folosind GPU-ul a aplicatiilor de uz general.

In ce priveste criptografia au existat doar doua directii notabile pentru a folosi GPU-ul. Prima, pornita in 2006 de D. L. Cook si A. D. Keroymytis, cu lucrarea“Cryptographics: Exploiting Graphics Cards for Security”, propune folosirea API-urilor 3D pentru programarea GPU-ului cu „shadere”. Aceasta solutie, discutata si intr-un articol al cartii GPU Gems 3, editata de nVidia, nu aduce imbunatatiri prea mari ca viteza de rulara datorita multor impedimente legate de folosirea resurselor GPU-ului insa are avantajul de a fi integrabila cu orice aplicatie 3rd party ce isi poate salva datele ca textura si poate rula pe orice placa video standard. Pe de alta parte solutiile care folosesc API-urile propietare de acces la resursele reale ale GPU-ului pentru programare de uz general, criptografie in acest caz, ofera fara indoiala imbunatatiri mai mari dar au restrictii din punct de vedere al hardware-ului ce poate fi folosit. Pentru CUDA este nevoie de un chip G80(GeForce8) sau mai nou. Numarul de multiprocesoare este teoretic transparent dezvoltatorului insa acesta influenteaza performantele, in cazul AES insemnand cate blocuri pot fi criptate in paralel.

Studiul realizat de aceasta lucrare foloseste a doua varianta, cea a utilizarii de API-uri propietare pentru acces la GPU, in particular nVidia CUDA. Asta inseamna ca nu poate rula decat pe hardware nVidia, in particular G80, G92 sau mai nou.

Dupa cum am discutat si in capitolele anterioare se poate observa o imbunatatire semnificativa daca se foloseste CUDA cu optimizarile discutate, atunci cand volumul de date e destul de mare pentru a folosi cat mai multe din core-urile GPU-ului.

Deoarece imbunatatirile sunt semnificative numai la un volum mai mare date nu este practica folosirea GPU-ului pentru criptarea de pachete mici cum ar fi trafic de retea in timp real, mai ales ca transferul de date intre GPU si CPU e limitat de bus-ul PCI Express. Aplicatiile mai evidente sunt in domeniul criptarii de fisiere sensibile, unde GPU-ul poate accelera semnificativ operatiile. Un scenariu comun ar fi o aplicatie de criptat/decriptat fisiere cu AES pe GPU.

Puterea de calcul a GPU-ului a fost aplicata in ultima perioada si pentru spargerea diferitelor criptari. De exemplu, GPU-ul a fost folosit cu succes in accelerarea spargerii protectiei WEP cu cheie partajata. Disponibilitatea mare a GPU-urilor pune oarecum sub intrebare siguranta diferitelor mecanisme de securitate existente si sugereaza ca aceasta ar trebui sporita. Din pacate, pentru a sporii siguranta, trebuie sa folosim chei mai lungi si deci putere de calcul mai mare la criptare/decriptare, fapt ce ar avea un impact negativ asupra utilizatorului de rand.

Desi nu putem prezice cu certitudine in acest moment care va fi amploarea fenomenului de utilizare a GPU-ului in aplicatii de uz general, el pare a fi o resursa cu un potential mare de a rula si a accelera aplicatii care suporta un grad mare de paralelism.

O dezvoltare naturala a proiectului ar fi implementarea si a altor algoritmi din criptografie, cum este MD5, algoritmi din sfera certificatelor digitale, SHA de exemplu, si nu numai. Se poate consrui in acest fel o biblioteca destul de generala de criptare/decriptare pe GPU.

O alta directie de cercetare ar fi rularea pe alte arhitecturi si optimizarea in functie de numarul de core-uri disponibile aplicatiei, insotita de colectarea a cat mai multor informatii legate de performantele algoritmului pe arhitectura nVidia CUDA.

# Bibliografie

1. Documentatia oficiala NVIDIA CUDA <http://www.nvidia.com/object/cuda_develop.html>
2. „CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography” <http://www.manavski.com/downloads/PID505889.pdf>
3. Specificatii AES <http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf>
4. GPU Gems 3 <http://developer.nvidia.com/object/gpu-gems-3.html>
5. „General-Purpose Computation Using Graphics Hardware” <http://www.gpgpu.org>
6. AMD <http://ati.amd.com/companyinfo/researcher/documents.html>

# Anexa 1: Compilarea si rularea aplicatiei

Pentru a compila aplicatia demonstrativa inclusa, CryptoGPU avem nevoie de urmatoarele instrumente:

1. Driver nVidia cu suport CUDA(si hardware-ul adecvat, bineinteles)
2. Toolkit-ul nVidia pentru compilarea partii pt GPU
3. SDK-ul nVidia CUDA pentru unele biblioteci ajutatoare CUDA
4. O versiune de wxWidgets (preferabil 2.8.x)
5. Visual Studio 2005

Dupa instalarea elementelor pentru CUDA, trebuie definite doua variabile de mediu: CUDA\_SDK\_INC\_PATH si CUDA\_SDK\_LIB\_PATH care sa dea calea catre locul unde sunt bibliotecile SDK-ului nVidia CUDA. Din pacate acestea nu sunt definite automat asa cum sunt CUDA\_INC\_PATH/CUDA\_LIB\_PATH pentru Toolkit-ul CUDA.

In ce priveste wxWidgets, pe Windows se foloseste portarea wxMSW care poate fi compilata atat pentru arhitecturi x86 cat si x64. Odata compilat wxWidgets, se defineste variabila de mediu WXWIN care dea calea catre directorul radacina in care a fost instalat wxWidgets(e.g. c:\wxWidgets-2.8.x).

Toae sursele proiectului sunt disponibile pe svn la svn://chriswinxp.no-ip org sau in browser(svnweb) <http://chriswinxp.no-ip.org/websvn/>. Momentan, din motive de securitate porturile de acces la SVN si HTTP sunt blocate. Discul atasat acestei lucrari contine un snapshot-ul al SVN-ului la momentul scrierii acestui document.