Универзитет у Београду

Математички факултет

Милош Ђурић

Реализација алгоритма AES на платформи nVidia CUDA

мастер рад

Београд 2016.

Универзитет у Београду

Математички факултет

Аутор:	Милош Ђурић
Наслов:	Реализација алгоритма AES на платформи nVidia CUDA
Ментор:	др Миодраг Живковић Универзитет у Београду, Математички факултет
Чланови комисије:	др Предраг Јаничић Универзитет у Београду, Математички факултет
	др Саша Малков Универзитет у Београду, Математички факултет
Датум:	

Садржај:

1. Увод

Заштита података и комуникације је један од најбитнијих сегмената рачунарства. Честа је потреба у којој две стране размењују поруке. *Отвореним текстом* се назива порука коју једна страна треба да пошаље другој. *Шифрат* је верзија отвореног текста која је трансформисана на одређени начин. Процес стварања шифрата се назива *шифровање*. Трансформација шифрата у оригинални текст се назива *дешифровање. Криптологија* је наука која се бави изучавањем шифровања и дешифровања, *криптографија* је област која се бави њеним применама, а к*риптоанализа* је област која се бави истраживањем како да се шифрат пребаци у отворени текст без претходног знања о томе на који начин је отворени текст шифрован [1]. Како рачунарство напредује тако напредује и потреба за ефикаснијим средствима заштите. Као последица развоја могућности програмирања на графичким картицама појавила се могућност бржег извршавања једноставнијих израчунавања, која представљају основу познатих криптографских проблема. Један од основних разлога зашто је изабрана ова тема је да се покаже да ли је и колико ефикасније решавати одређене типове криптографских проблема на графичким процесорским јединицама.

У овом раду детаљно се описују поступци шифровања и дешифровања уз помоћ једног од најпознатијих криптографских алгоритама *AES* (енг. *Advanced Encryption Standard*). Приказује се неколико његових паралелних имплементација на графичкој картици уз помоћ платформе *CUDA* (енг. *Compute Unified Device Architecture*). Зарад приказивања ефикасности наведене имплементације приказане су и одговарајуће секвенцијалне имплементације алгоритма, са упоредним резултатима ових имплементација.

1.1. Кратка историја криптографије

Шифровање у циљу заштите информација примењивало се још у доба пре нове ере. Глинене плоче из Месопотамије су пример једне од првих практичних употреба криптографије. Једна таква плоча, настала око 1500. године пре нове ере, садржи шифрован рецепт како се формира глазура за прављење гранчарије [2]. Римски Конзул Јулије Цезар је штитио своје поруке замењивањем сваког слова текста одговарајућим словом азбуке које се налази три места улево од оригиналног. Овај једноставни метод шифровања је познат као Цезарова шифра и један је од најпознатијих техника шифровања.

Арапи су први почели систематски да документују методе криптоанализе и ту се налазе зачеци модерне криптологије. Око 800 године нове ере појавила се прва техника за анализу понављања са сврхом разбијања текстова шифрованих техником замене коришћењем једне азбуке. Творац ове технике арапски математичар Ал-Кинди је написао прву књигу на тему криптографије, *Risalah fi Istikhraj al-Mu'amma* (прев. Рукопис о дешифровању криптографских порука) [3]. У књизи су описане разне технике криптоанализе, класификација шифара, први описи анализе понављања, као и неке шифре замене засноване на вишеструким азбукама

(фамилија машина Енигма које су коришћене у Другом светском рату су засноване на шифрама овог типа).

Временом се појавила потреба за заштитом података на глобалном нивоу, као и за високим нивоима заштите података разних државних и приватних агенција. На основу новонасталих потреба за шифровањем почетком седамдесетих година двадесетог века тадашњи Амерички биро за стандарде, данашњи *NIST* (енг. *National Institute for Standards and Technology*) је издао захтев за стандард са ригорозним критеријумима заштите [4]. Тако је 1976. године одобрен први стандард за шифровање – *DES* (енг. *Data Encryption Standard*). Годину дана касније *DES* је јавно прихваћен и објављен. Алгоритам *DES* је био подржан од стране Америчке агенције за националну безбедност *NSA* (енг. *National Security Agency*). То је допринело да *DES* почне да се користи свуда по свету. Америчка влада је захтевала да се све банкарске трансакције у Америци обрађују алгоритмом *DES*. Основни алгоритам *DES* шифрује текст величине 64 бита са кључем од 56 бита. *DES* је базиран на Фајстејловој структури [5]. Фајстелова структура је симетрична структура која се користи за формирање шифрата. Код Фајстелове структуре операције шифровања и дешифровања су јако сличне и самим тим имплементација алгоритма је много лакша. Услед симетричног својства често није потребно имплементирати шифровање у оба смера, већ је довољно имплементирати у једном смеру и само мењати кључ у зависности од смера шифровања [6].

Иако је било неких сумњи *DES* се показао као стандард који задовољава прописане захтеве за заштиту података у тренутку у ком је објављен. Због мале дужине кључа порастом рачунарске снаге која је уследила по Муровом закону, деведесетих година 20. века *DES* је постао рањив на нападе грубом силом (енг. *brute force attacks*). Као нова мера заштите појавио се троструки *DES* (енг. *tripple DES*). Име је добио по методу шифровања, троструки *DES* трипут шифрује текст величине 64 бита. Кључ за троструки *DES* се састоји од три кључа дужине 56 бита (K_1, K_2, K_3). Процес шифровања се очито састоји од три корака, прво се текст дужине 64 бита шифрује кључем K_1 па се дешифрује кључем K_2 па се шифрује кључем K_3. Дешифровање се обавља као инверзна радња. Постоји неколико препорука за зависност кључева. Препоручена опција је да су сва три кључа различита ($K_1 \neq K_2 \neq K_3 \neq K_1$). Изричито је забрањено да су сва три кључа идентична. Троструки *DES* сме да се користи у сврхе шифровања података државних институција Америке до краја 2030. године [7].

DES је настао од Фајстеловог алгоритма *Lucifer* [8] који је за разлику од *DES* -а користио пун 64-битни кључ. Иако *DES* заправо има 64-битни кључ, последњих осам цифара се користе за контролу парности и оне не утичу на безбедност алгоритма. Ово је изазвало одјек у академском свету, пре свега са питањем да ли је 56-битни кључ довољно дугачак да адекватно заштити податке. Већ споменута Америчка агенција за националну безбедност је оригинално одбила да направи стандард, али је имала велики утицај на развој самог стандарда. Утицали су на смањење кључа на величину од 56 бита као и на дизајн такозване „кутије замене“, односно табеле супституције (енг. *S-box*). Цео процес формирања стандарда је био скривен од јавности.

Услед свих ових чињеница појавило се друго питање, да ли је *NSA* потенцијално оставила такозвана „задња врата“ за разбијање, односно неки начин брзе криптоанализе у стандарду *DES*? И академски и приватни сектор су изразили своје сумње, али, не постоји доказ да таква „врата“ постоје. Наредних 20 година је било доста неслагања између академског и приватног сектора са једне стране и државних агенција са друге, због цензуре и ограничавања слободе говора. Када су професори са Масачусетског Института за Техонологију одлучили да објаве своје иновативно истраживање повезано са криптографијом јавних кључева, сусрели су се са налозима о тајности и претњама хапшењем у случају објављивања рада. И поред тога они су успели да објаве свој рад уз помоћ адвокатског тима, у коме је описан данас најпознатији алгоритам са јавним кључем, *RSA* (енг. *Rivest-Shamir-Adleman*), назван по творцима Ривесту, Шамиру и Аделману.

1.2. Историјат алгоритма *AES*

Средином деведесетих година 20. века било је очигледно да је *DES* застарео стандард и да ће троструки *DES* застарети у периоду од 30 година. Јануара 1997. *NIST* је упутио позив ка академском и приватном сектору за формирање новог стандарда за криптовање. Ово је била велика промена у односну на *DES* и цензуру јер је цео процес био транспарентан. Од старта је *NIST* одржала реч прихвативши све критике везане за захтеве алгоритма, критеријуме процене, итд. Након осам месеци расписан је конкурс за алгоритме. Формиран је документ који је садржио минималне захтеве алгоритма и критеријуме процене. Захтеви које је алгоритам морао да задовољава су следећи:

- да буде јавно дефинисан,
- симетричан,
- да шифрује блокове,
- да буде прилагодљив различитим величинама кључева (кључеви величине 128, 192 и 256 бита),
- да може да се извршава и у хардверу и у софтверу,
- и да буде бесплатан.

Критеријуми на основу којих су оцењивани алгоритми су били следећи:

- безбедност,
- флексибилност,
- ефикасност,
- искоришћеност меморије,
- могућност лаке имплементације у хардверу и софтверу,
- једноставност,
- тип лиценцирања,
- и отпорност на све познате криптоаналитичке нападе, чак и на оне који још увек не могу да се користе у пракси.

Нови стандард је добио име *Напредни енкрипциони стандард - AES*.

Другог октобра 2000. *NIST* објављује да је Рајндол (енг. *Rijndael*) изабран за алгоритам за нови стандард *AES*. *NIST* је одлучио да не именује другопласирани алгоритам као алгоритам подршке, јер би аутоматски оба алгоритма за шифровање морала бити имплементирана (*AES* као основни и другопласирани као подршка), а то би повећало цену производње и смањило интероперабилност.

NIST је објавио да је Рајндол најбоља опција од свих кандидата. Као главни разлози споменути су ефикасност у хардверским и софтверским имплементацијама, као и на разним платформама, мала искоришћеност меморије која чини алгоритам идеалан за имплементацију на уређајима ограничених ресурса (као што су паметне картице, енг. *Smart cards*), логика је таква да се лако брани од познатих напада.

Алгоритам *AES* је званично објављен 26. новембра 2001. године. Успех и начин избора алгоритма *AES* је наишао на велико одобрење од свих заједница. Дејвид Аусмит, главни безбедносни архитекта корпорације *Intel,* оценио је одабирни процес као „модел индустријске, академске и државне сарадње“ [8].

2. Алгоритам *AES*

У тачки 2.1. дат је груби опис алгоритма *AES.*

У тачки 2.2. описана је табела супституције *S (S-box)*, која се користи у неколико операција алгоритма *AES*.

У тачки 2.3. описана је операција формирања кључева који се користе у трансформацији *DodavanjeKључаRunde.*

У тачки 2.4. односно тачки 2.5. детаљно се описују акције шифровања и дешифровања алгоритмом *AES*. Разлика између ове две акције је у трансформацијама, које иако су јако сличне се разликују у детаљима, као и у редоследу којим се примењују трансформације.

У тачки 2.6. се налази текст о квалитету алгоритма *AES.*

2.1. Груби опис алгоритма *AES*

Већина блоковских алгоритама за шифровање као основу за дефинисање рунди трансформације користи Фајстелову структуру, споменуту у претходној тачки. Алгоритам *AES* не користи Фајстелову структуру. Алгоритам *AES* је симетрични блоковски метод за шифровање. Стандард *AES* прописује да се искључиво користи блок величине 128 бита и кључ који може бити величине 128, 192 или 256 бита [9]. У зависности од величине кључа име стандарда се мења у *AES*-128, *AES*-192 и *AES*-256. Број рунди шифровања блока зависи од дужине кључа. Ако се користи кључ дужине 128 бита број рунди шифровања је 10, док за кључеве дужине 192 и 256 бита, број рунди је 12, односно 14. У употреби је најчешће кључ дужине 128. У наставку се подразумева да се користи кључ дужине 128 бита.

Алгоритам Рајндол дизајниран је са следећим карактеристикама:

- отпорност на све познате нападе,
- брзина и компактност кода на великом броју различитих платформи,
- једноставан дизајн [10].

Остатак поглавља 1 и поглавље 2 су већим делом засновани на званичној документацији о стандарду *AES*, издате од стране *NIST* [11].

Основна јединица над којим се извршавају акције алгоритма *AES* је бајт, секвенца од 8 бита која се третира као јединствени ентитет. Сви бајтови из алгоритма *AES* су представљени као конкатенација индивидуалних битова у витичастим заградама у следећем редоследу $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. Бајтови се интерпретирају као елементи коначног поља следеће полиномијалне репрезентације:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 = \sum_{i=0}^{7} b_i x^i.$$

На пример, {10001011} означава елемент коначног поља: $x^7 + x^3 + x + 1$.

Користи се и хексадекадна нотација у којој бајт {10001011} има вредност {8b}.

Коначно поље GF(2^8) састоји се од свих могућих остатака бинарних полинома при дељењу са дефинишућим полиномом m(x):

$$m(x) = x^8 + x^4 + x^3 + x + 1;$$

односно {01} {1b} у хексадекадном запису.

Дефинишући полином m(x) је несводљив. Полином је несводљив ако је једино дељив самим собом и јединицом.

Алгоритам *AES* се интерно извршава над дводимензионалним низом који се зове **матрица стања**. Матрица стања се састоји од четири реда бајтова где сваки садржи *n* бајтова, где је *n* једнако дужини блока подељеној са 32. У случају алгоритма *AES* матрица стања је величине 4x4 и свако поље садржи тачно један бајт. Сваки индивидуални бајт ***m*** има 2 индекса *r* и *k* ($0 \leq r, k < 4$) који означавају његову позицију у матрици стања. Као што је већ споменуто, величина блока који се обрађује, шифрованог блока и матрице стања је 128 бита.

При шифровању и дешифровању алгоритам *AES* користи функцију рунде која је сачињена од четири различите битске трансформације:

1. ***Замена бајтова*** (енг. ***SubBytes***) - замена бајтова одговарајућим вредностима из табеле супституције
2. ***Померање редова*** (енг. ***ShiftRows***) - померање чланова редова удесно са различитим кораком за различите редове
3. ***Измена колона*** (енг. ***MixColumns***) – измена вредности колона матрице стања
4. ***Додавање кључа рунде*** (енг. ***AddRoundKey***) **–** додавање по модулу два, бит по бит (применом ексклузивне дисјункције, енг. *XOR*) одговарајућег кључа те рунде матрици стања

На почетку се реч која се шифрује, величине 128 бита, трансформише у низ бајтова ul_o, ul_1, ul_2...ul_{15} и тај низ се уписује у матрицу стања (слика 1). Након тога се извршавају акције шифровања или дешифровања над матрицом стања и резултат се уписује у излазни низ битова iz_0, iz_1, iz_2...iz_{15}.

Слика 1: Проток података

Улазни низ *ul* се копира у матрицу стања на следећи начин:

$$m_{r,k} = ul[r + 4k], \text{ где је } 0 \leq r, k < 4.$$

На сличан начин се резултат из матрице стања копира у излазни низ *iz*:

$$iz[r + 4k] = m_{r,k}, \text{ где је } 0 \leq r, k < 4.$$

Четири бајта из сваке колоне матрице стања формирају 32-битну **реч**, где број реда *r* означава индекс бајта унутар речи. Самим тим матрица стања може се посматрати као низ четири речи. На примеру слике 1, речи би изгледале:

$$rec_0 = m_{0,0}\, m_{1,0}\, m_{2,0}\, m_{3,0}$$

$$rec_1 = m_{0,1}\, m_{1,1}\, m_{2,1}\, m_{3,1}$$

$$rec_2 = m_{0,2}\, m_{1,2}\, m_{2,2}\, m_{3,2}$$

$$rec_3 = m_{0,3}\, m_{1,3}\, m_{2,3}\, m_{3,3}$$

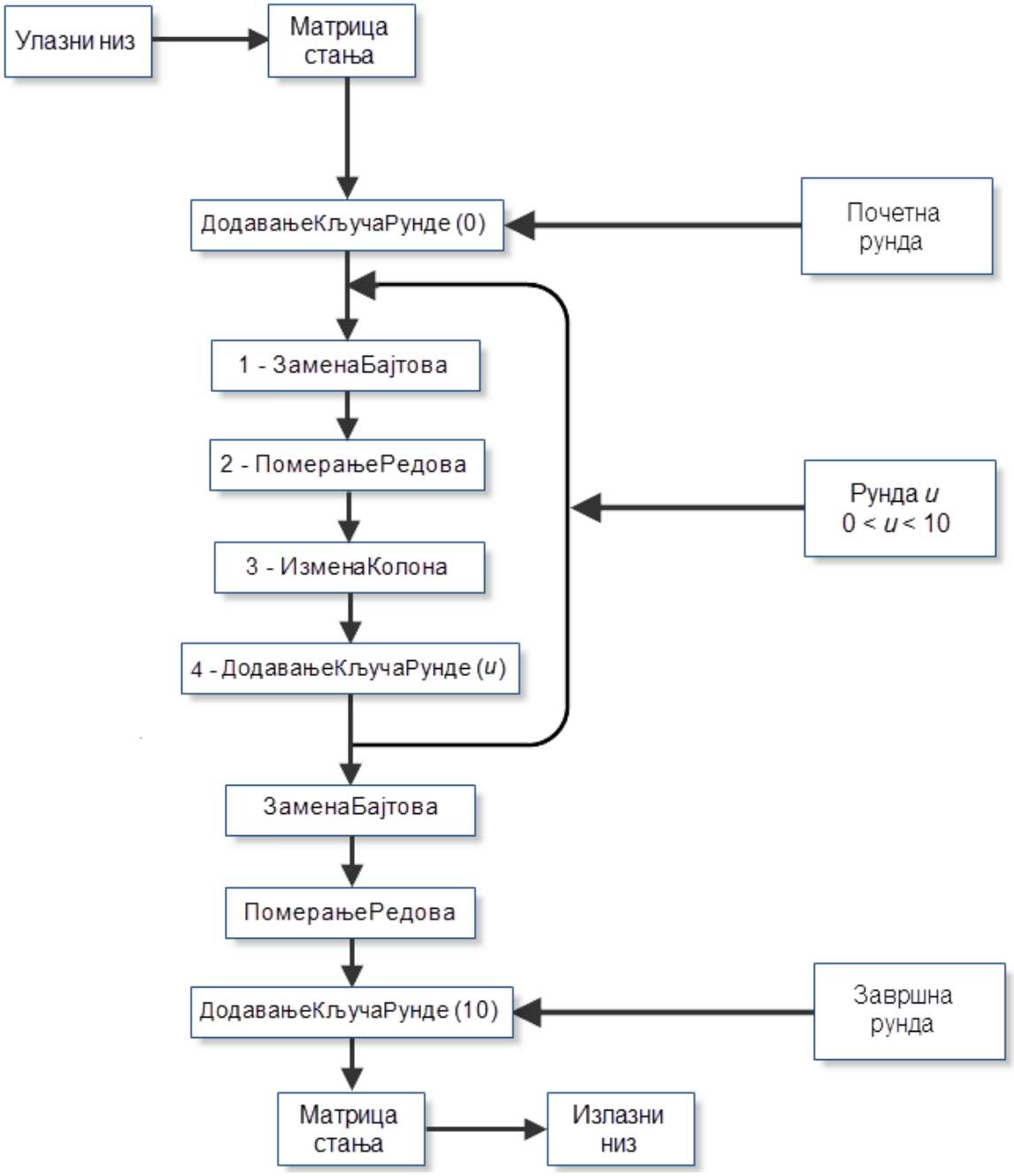
Након формирања матрице стања следећи корак је иницијална рунда. Иницијална рунда се састоји од битске трансформације *DodavanjeKључаRunde,* која се извршава над матрицом стања уз оригинални кључ. Након тога матрица стања улази у прву рунду функције рунде. Иако алгоритам има 10 рунди, последња рунда се разликује од претходних 9 јер не садржи битску трансформацију *IzmenaKolona*. По речима твораца алгоритма ове измене су направљене да би шифровање и дешифровање били сличније у структури и да не утичу на сигурност алгоритма. Тачка 1.3. се бави квалитетом алгоритма *AES.*

Алгоритам *AES* може се приказати следећим псеудокодом, односно дијаграмом на слици 2.

Псеудокод 1: Алгоритам AES

```
Ulaz: ul (ulazni niz), kljuc (niz reči ključeva)
Izlaz: iz (izlazni niz)
AES(byte ul[4, 4]), byte iz[4, 4], word kljuc[44])
begin
        byte matrica_stanja[4 ,4]
        matrica_stanja = ul
        DodavanjeKljucaRunde(matrica_stanja, kljuc[0, 3])
        for runda = 1 step 1 to 9
                ZamenaBajtova(matrica_stanja)
                PomeranjeRedova(matrica_stanja)
                IzmenaKolona(matrica_stanja)
                DodavanjeKljucaRunde(matrica_stanja, kljuc[runda*4, (runda+1)*4-1])
        end for
        ZamenaBajtova(matrica_stanja)
        PomeranjeRedova(matrica_stanja)
        DodavanjeKljucaRunde(matrica_stanja, kljuc[40, 43])
        iz = matrica_stanja
end
```

Због прегледности, поред псеудокода дата је визуелна репрезентација дијаграмом на слици 2.



Слика 2. Дијаграм алгоритма AES

Сви бајтови у алгоритму *AES* се интерпретирају као елементи коначног поља GF(2^8) са раније уведеном нотацијом. Елементи коначних поља се множе и сабирају на другачији начин од бројева. У наредном сегменту уводе се основни математички појмови неопходни за дубље разумевање функционисања алгоритма *AES*.

Сабирање елемената поља је сабирање одговарајућих полинома. Са коефицијентима се рачуна по модулу два, односно примењује се операција ексклузивне дисјункције. Ознака $\oplus$ се користи за ексклузивну дисјункцију (и сабирање у коначном пољу). Самим тим и одузимање је еквивалентно сабирању. Други начин на који може да се посматра сабирање у коначном пољу је сабирање одговарајућих битева по модулу два у бајту. За два бајта $\{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\}$ и $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$, збир је једнак $\{c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0\}$, где је свако c_i једнако $a_i \oplus b_i$.

У репрезентацији полиномима множење у коначном пољу GF(2^8) одговара акцији множењем полинома по модулу несводљивог полинома *m(x)(*)*, где је * број који је добила једнакост која дефинише *m(x)*. Ознака • се користи за множење полинома у коначном пољу GF(2^8).

Свођење по модулу *m(x)* обезбеђује да резултат множења буде бинарни полином, степена мањег од осам, тако да се може представити као један бајт. Овако дефинисано множење је асоцијативно, неутрал за множење је {01} хексадекадно. За сваки ненула бинарни полином *b(x)* степена мањег од осам, постоји инверз за множење $b^{-1}(x)$ и одређује се уз помоћ проширеног Еуклидовог алгоритма који се користи да се израчунају полиноми *a(x)* и *c(x)* такви да важи:

$$b(x)a(x) + m(x)c(x) = 1.$$

Следи $a(x) \bullet b(x) = 1$, из чега следи:

$$b^{-1}(x) = a(x).$$

Још важи да за сваки *a(x)*, *b(x)* и *c(x)* из коначног поља важи следећа једнакост:

$$a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x).$$

Самим тим, скуп од 256 могућих вредности бајта са уведеним операцијама сабирања и множења јесте коначно поље GF(2^8).

Специјално, множење *m* полинома

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

полиномом *x* добија се

$$b_7 x^8 + b_6 x^7 + b_5 x^6 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x.$$

Резултат $x \bullet b(x)$ се добија када се резултат сведе по модулу $m(x)$. Ако је $b_7 = 0$, резултат је већ у свом сведеном облику. Ако је $b_7 = 1$ редукција се постиже одузимањем (односно применом ексклузивне дисјункције) полинома $m(x)$. Следи да множење са x може да се имплементира на битском нивоу као леви померај и одговарајућа ексклузивна дисјункција са {1b} хексадекдано.

Множење вишим степенима од x може се имплементирати вишеструком применом ове описане акције.

У оквиру алгоритма *AES* користе се и полиноми степена највише три по y, којима су коефицијенти елементи поља:

$$a(y) = a_3 y^3 + a_2 y^2 + a_1 y + a_0$$

који се могу представити колоном $[a_0 , a_1 , a_2 , a_3]^T$. Ови полиноми су елементи прстена остатка при дељењу са $y^4 + 1$ полинома са коефицијентима из коначног поља.

Зато што полином $y^4 + 1$ није несводљив над пољем GF(2^8), множење са фиксираним полиномом од четири терма није обавезно инверзно. Али, у алгоритму *AES* користи се полином:

$$a(y) = \{03\}y^3 + \{01\}y^2 + \{01\}y + \{02\}$$

који има инверз у прстену:

$$a^{-1}(y) = \{0b\}y^3 + \{0d\}y^2 + \{09\}y + \{0e\}.$$

Због илустрације операција сабирања и множења нека је

$$b(y) = b_3 y^3 + b_2 y^2 + b_1 y + b_0$$

дефиниција другог полинома са четири терма. Сабирање се извршава сабирањем коефицијената коначног поља уз одговарајуће степене y. Ова операција сабирања одговара операцији ексклузивне дисјункције за све бите над одговарајућим бајтовима сваке речи.

Сабирање претходна два полинома би изгледало овако:

$$a(y) + b(y) = (a_3 \oplus b_3) y^3 + (a_2 \oplus b_2) y^2 + (a_1 \oplus b_1) y + (a_0 \oplus b_0).$$

Производ $d(y) = a(y) \bullet b(y)$ једнак је остатку при дељењу производа $a(y)b(y)$ са полиномом y^4+1.

Другим речима коефицијенти полинома *d(y)*, дефенисаног као

$$d(y) = d_3 y^3 + d_2 y^2 + d_1 y + d_0$$

су једнаки модуларном производу *a(y)* и *b(y)* који се означава са *a(y)*⊗*b(y)*:

$$d_0 = (a_0 \bullet b_0) \oplus (a_3 \bullet b_1) \oplus (a_2 \bullet b_2) \oplus (a_1 \bullet b_3)$$

$$d_1 = (a_1 \bullet b_0) \oplus (a_0 \bullet b_1) \oplus (a_3 \bullet b_2) \oplus (a_2 \bullet b_3)$$

$$d_2 = (a_2 \bullet b_0) \oplus (a_1 \bullet b_1) \oplus (a_0 \bullet b_2) \oplus (a_3 \bullet b_3)$$

$$d_3 = (a_3 \bullet b_0) \oplus (a_2 \bullet b_1) \oplus (a_1 \bullet b_2) \oplus (a_0 \bullet b_3)$$

Када је *a(y)* фиксирани полином, операција дефинисана у претходној једнакости може се представити матричним производом као:

$$\begin{vmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{vmatrix} = \begin{vmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{vmatrix} \begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{vmatrix}$$

2.2. Табела супституције *S (S-box)*

Табела супституције се користи у неколико операција алгоритма *AES*. У табели 1 приказана је целокупна табела супституције *S (S-box)* у хексадекадној форми.

Табела 1: Табела супституције S (S-box)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	1	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	4	C7	23	C3	18	96	5	9A	7	12	80	E2	EB	27	B2	75
4	9	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	0	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	2	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	6	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	8
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	3	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

2.3. Операција *ProширивањеКључа*

Алгоритам *AES* формира распоред кључева (проширени кључ) проширивањем почетног кључа *pk.* У оквиру проширивања кључа генерише се $n(nr + 1)$ речи, док број рунди *nr* варира. У наставку се подразумева да је кључ дужине 128 бита и број рунди $nr = 10$, тако да проширени кључ садржи једанаест кључева величине четири речи, односно 44 речи.

Кључ почетне рунде је једнак почетном кључу *pk* и он је први кључ у проширеном кључу. Остали кључеви се израчунавају применом одређених трансформација на речи првог кључа. Те трансформације су *ZameniRec*, *RotirajRec* и ексклузивна дисјункција (*XOR*) са низом константних речи *Rkon*.

Трансформација *ZameniRec* сваки бајт у речи замењује одговарајућом вредношћу из табеле супституције из тачке 2.2. Нова вредност се одређује тако што се тражи пресек индекса са леве стране табеле који одговара првом индексу бајта, са индексом десне стране табеле који одговара другој вредности бајта.

Трансформација *RotirajRec* узима реч $[a_0, a_1, a_2, a_3]$ и циклично помера сваки бајт за једну позицију улево и враћа реч $[a_1, a_2, a_3, a_0]$. *Rkon* је низ који садржи константне речи. *Rkon* за индекс *j* враћа реч $[x^{j-1}, \{0,0\}, \{0,0\}, \{0,0\}]$, где су x^{j-1} степени од полинома *x* (*j* се означава са {02}) у пољу $GF(2^8)$. Битно је напоменути да за низ *Rkon* индекс *j* не почиње од нуле већ од броја 1.

Прве четири речи проширеног кључа, који је представљен низом rec[i], $0 \leq i \leq 44$, су једнаке почетном кључу *pk*. Речи, rec[i], за које је индекс *i* већи од броја три, и није дељив са бројем четири једнаке су ексклузивној дисјункцији између речи на позицији [i-1] и речи на позицији [i-4].

Речи, rec[i], за које је индекс *i* већи од броја три, и дељив је бројем четири добијају се тако што се на rec[i-1] прво примени трансформација састављена од трансформације *RotirajRec* и трансформације *ZameniRec*. Када се ова трансформација обави на добијену реч примени се ексклузивна дисјункција са одговарајућим чланом низа *Rkon*. Индекс члана низа *Rkon* са којим се извршава ексклузивна дисјункција једнак је тренутној рунди формирања кључа. На крају се на трансформисану реч још изврши ексклузивна дисјункција са речју на позицији [i-4].

Псеудокод 2: Проширивање кључа

```
ProsirivanjeKljuca(byte kljuc[16], word rec[44])
begin
        word tmp
        i = 0
        while (i < 4)
                rec[i] = word(kljuc[4*i], kljuc[4*i + 1], kljuc[4*i + 2], kljuc[4*i + 3])
                i = i + 1
        end while
        i  = 4
        while (i < 44)
                tmp = rec[i-1]
                if(i mod 4 = 0)
                        tmp = ZameniRec(RotirajRec(tmp)) xor Rkon[i/4]
                end if
                rec[i] = rec[i – 4] xor tmp
                i = i + 1
        end while
end
```

2.4. Шифровање алгоритмом *AES*

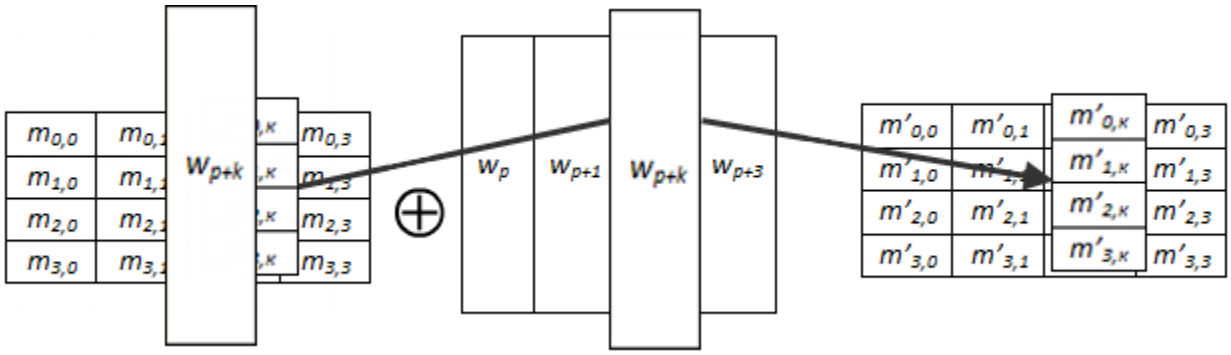
Шифровање алгоритмом *AES* састоји се од четири битске трансформације које се извршавају одређеним распоредом и наведене су у тачки 2.1. Скуп тих трансформација назива се функција рунде. Функција рунде се извршава више пута у зависности од величине кључа.

2.4.1. Трансформација *ДодавањеКључаРунде*

Трансформација *ДодавањеКључаРунде* се састоји од примене ексклузивне дисјункције на једну колону матрице стања са одговарајућим кључем из проширеног кључа. Сваки кључ рунде се састоји од четири речи из проширеног кључа таквих да важи:

$$[m'_{0,k}, m'_{1,k}, m'_{2,k}, m'_{3,k}] = [m_{0,k}, m_{1,k}, m_{2,k}, m_{3,k}] \oplus [w_{runda*4+k}],\ 0 \le k < 4$$

где $[w_i]$ представља проширени кључ i, *runda* је вредност између 0 и 10 ($0 \le runda \le 10$). Трансформација *ДодавањеКључаРунде* се први пут одвија у почетној рунди када је *runda* = 0,. На слици 3 се налази илустрација ове трансформације, где је померај $p = runda * 4$.



Слика 3: Акција додавања кључа

2.4.2. Трансформација *ZamenaBajtova*

Трансформација *ZamenaBajtova* је нелинеарна замена бајтова која независно утиче на сваки бајт матрице стања користећи табелу супституције (*S-box*) приказану у тачки 2.2.

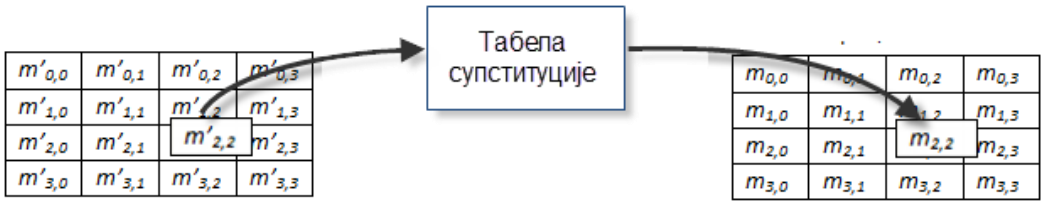
Трансформација одређена табелом супституције је композиција две трансформације:

1. Трансформација која ненула елементе пресликава у њихов инверз, а {00} у {00}
2. Примена следеће афине трансформације над GF(2^8) дефинисане изразом:

$$b_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

где је $0 \le i < 8$, b_i је i-ти бит бајта, c_i је i-ти бит бајта c који има вредност {63} или {01100011}.

На слици 4 илустровано је како табела супституције делује на матрицу стања. Индекси са леве стране табеле одговарају првом индексу бајта матрице стања, док индекси са десне (изнад) одговарају другом индексу бајта матрице стања. На пример, ако би се мењао бајт $m_{1,3}$ = {3a}, вредност замене би била једнака вредности у табели супституције из тачке 2.2. која се налази на пресеку реда са индексом '3' и колоне са индексом 'a'. Вредност $m'_{1,3}$ би била једнака {80}.



Слика 4: Акција замене бајтова

2.4.3. Трансформација *PomeranjeRedova*

Овом трансформацијом последња 3 реда матрице стања се циклично померају за одређени број бајтова. Број за колико места ће се бајтови преместити улево је једнак индексу реда (*r*).

Први ред, *r = 0*, остаје исти, док се бајтови у другом реду, чији је индекс *r = 1,* циклички померају за једно место улево, у трећем за два и у четвртом за три. На слици 5 је приказ матрице стања пре и после трансформације *PomeranjeRedova*.

Матрица стања *m*

$m_{0,0}$	$m_{0,1}$	$m_{0,2}$	$m_{0,3}$
$m_{1,0}$	$m_{1,1}$	$m_{1,2}$	$m_{1,3}$
$m_{2,0}$	$m_{2,1}$	$m_{2,2}$	$m_{2,3}$
$m_{3,0}$	$m_{3,1}$	$m_{3,2}$	$m_{3,3}$

Матрица стања *m'* након извршене трансформације

$m_{0,0}$	$m_{0,1}$	$m_{0,2}$	$m_{0,3}$
$m_{1,1}$	$m_{1,2}$	$m_{1,3}$	$m_{1,0}$
$m_{2,2}$	$m_{2,3}$	$m_{2,0}$	$m_{2,1}$
$m_{3,3}$	$m_{3,0}$	$m_{3,1}$	$m_{3,2}$

Слика 5:. Акција померања редова

2.4.4. Трансформација *IzmenaKolona*

Трансформација *IzmenaKolona* изводи се над матрицом стања колону по колону, посматрајући сваку колону као полином од четири терма, као што је описано у тачки 2.1. Колоне се посматрају као полиноми над GF(2^8) и множе по модулу $x^4 + 1$ са фиксираним полиномом:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}.$$

Као што је описано у тачки 2.1. ово израчунавање може се представити као множење матрица.

Нека је $m'(x) = a(x) \otimes m(x)$:

$$\begin{bmatrix} m'0,k \\ m'1,k \\ m'2,k \\ m'3,k \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} m0,k \\ m1,k \\ m2,k \\ m3,k \end{bmatrix}$$, где је $0 \le k < 4$.

Као резултат овог множења, четири бајта у добијеним колонама су одређени изразом:

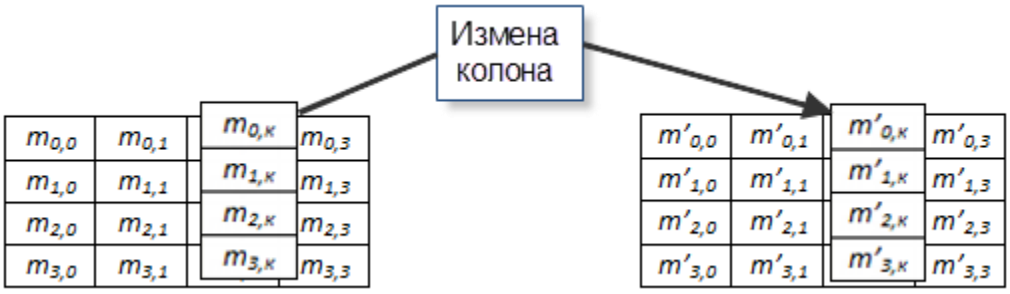
$$m'_{0,k} = (\{02\} \bullet m_{0,k}) \oplus (\{03\} \bullet m_{1,k}) \oplus m_{2,k} \oplus m_{3,k}$$

$$m'_{1,k} = m_{0,k} \oplus (\{02\} \bullet m_{1,k}) \oplus (\{03\} \bullet m_{2,k}) \oplus m_{3,k}$$

$$m'_{2,k} = m_{0,k} \oplus m_{1,k} \oplus (\{02\} \bullet m_{2,k}) \oplus (\{03\} \bullet m_{3,k})$$

$$m'_{3,k} = (\{03\} \bullet m_{0,k}) \oplus m_{1,k} \oplus m_{2,k} \oplus (\{02\} \bullet m_{3,k})$$

Слика 6 илуструје трансформацију *IzmenaKolona*.



Слика 6: Акција измене колона

2.5. Дешифровање (инверзни *AES*)

За имплементацију инверзног алгоритма *AES* довољно је инвертовати оригинални алгоритам *AES*. За дешифровање се користе следеће трансформације: *InverzZamenaBajtova*, *InverzPomeranjeRedova*, *InverzIzmenaKolona* и *DodavanjeKljučaRunde*. Код дешифровања уз помоћ инверзног алгоритма *AES* скоро све трансформације су инверзне трансформацијама алгоритма *AES*, осим трансформације *DodavanjeKljučaRunde* која је инверзна самој себи, па се користи у оригиналном облику.

Дешифровање алгоритма *AES* може се приказати следећим псеудокодом.

Псеудокод 3: Дешифровање AES

```
Ulaz: ul (ulazni niz), kljuc (niz reči ključeva)
Izlaz: il (izlazni niz)
AESDesifrovanje(byte ul[4, 4]), byte iz[4, 4], word kljuc[44])
begin
        byte matrica_stanja [4,4]
        matrica_stanja = ul
        DodavanjeKljucaRunde(matrica_stanja, kljuc[40, 43])
        for runda = 9 step 1 downto 1
                InverzZamenaBajtova(matrica_stanja)
                InverzPomeranjeRedova(matrica_stanja)
                DodavanjeKljucaRunde(matrica_stanja, kljuc[runda*4, (runda + 1)*4] - 1)
                InverzIzmenaKolona(matrica_stanja)
        end for
        InverzZamenaBajtova(matrica_stanja)
        InverzPomeranjeRedova(matrica_stanja)
        DodavanjeKljucaRunde(matrica_stanja, kljuc[0, 3])
        iz = matrica_stanja
end
```

2.5.1. Трансформација *InverzZamenaBajtova*

InverzZamenaBajtova је трансформација инверзна трансформацији *ZamenaBajtova*. У њој се користи инверзна табела супституције која се примењује на исти начин као и код оригиналне трансформације. Добија се уз помоћ инверзне трансформације афиној трансформацији из 2.4.2. и затим узимањем инверза за множење у GF(2^8). Табела 2 се користи за дешифровање и инверзна је табели из тачке 2.2.

Табела 2: Табела супституције за дешифровање

	0	1	2	3	4	5	6	7	8	9	a	b	c	D	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

2.5.2. Трансформација *InverzPomeranjeRedova*

Ова трансформација је директан инверз трансформације *PomeranjeRedova*. Бајтови последња три реда матрице се циклично померају за одређени број. Први ред, *r = 0* остаје непромењен а бајтови у последња три реда се циклички померају удесно за број места једнак индексу реда. На слици 7 је приказана илустрација ове трансформације.

Матрица стања *m*

$m_{0,0}$	$m_{0,1}$	$m_{0,2}$	$m_{0,3}$
$m_{1,0}$	$m_{1,1}$	$m_{1,2}$	$m_{1,3}$
$m_{2,0}$	$m_{2,1}$	$m_{2,2}$	$m_{2,3}$
$m_{3,0}$	$m_{3,1}$	$m_{3,2}$	$m_{3,3}$

Матрица стања *m'*
након извршене трансформације

$m_{0,0}$	$m_{0,1}$	$m_{0,2}$	$m_{0,3}$
$m_{1,3}$	$m_{1,0}$	$m_{1,1}$	$m_{1,2}$
$m_{2,2}$	$m_{2,3}$	$m_{2,0}$	$m_{2,1}$
$m_{3,1}$	$m_{3,2}$	$m_{3,3}$	$m_{3,0}$

Слика 7: Акција померања редова код дешифровања

2.5.3. Трансформација *InverzIzmenaKolona*

Слично двема претходним трансформацијама *InverzIzmenaKolona* је инверз трансформације *IzmenaKolona*. Трансформација *InverzIzmenaKolona* изводи се над матрицом стања колону по колону, посматрајућу сваку колону као полином од четири терма, као што је описано у тачки 2.1.

Колоне се посматрају као полиноми над GF(2^8) и множе по модулу $x^4 + 1$ са фиксираним полиномом:

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}.$$

Као што је описано у тачки 2.1. ово израчунавање може се представити као множење матрица. Нека је $m'(x) = a(x) \otimes m(x)$:

$$\begin{bmatrix} m'0,k \\ m'1,k \\ m'2,k \\ m'3,k \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} m0,k \\ m1,k \\ m2,k \\ m3,k \end{bmatrix}$$, где је $0 \leq k < 4$.

Као резултат овог множења, четири бајта у добијеним колонама су одређени изразом:

$$m'_{0,k} = (\{0e\} \bullet m_{0,k}) \oplus (\{0b\} \bullet m_{1,k}) \oplus (\{0d\} \bullet m_{2,k}) \oplus (\{09\} \bullet m_{3,k})$$

$$m'_{1,k} = (\{09\} \bullet m_{0,k}) \oplus (\{0e\} \bullet m_{1,k}) \oplus (\{0b\} \bullet m_{2,k}) \oplus (\{0d\} \bullet m_{3,k})$$

$$m'_{2,k} = (\{0d\} \bullet m_{0,k}) \oplus (\{09\} \bullet m_{1,k}) \oplus (\{0e\} \bullet m_{2,k}) \oplus (\{0b\} \bullet m_{3,k})$$

$$m'_{3,k} = (\{0b\} \bullet m_{0,k}) \oplus (\{0d\} \bullet m_{1,k}) \oplus (\{09\} \bullet m_{2,k}) \oplus (\{0e\} \bullet m_{3,k})$$

2.6. Квалитет алгоритма *AES*

За квалитет шифровања постоје два основна критеријума, сигурност и ефикасност. Осим ова два критеријума алгоритам *AES* поседује и једноставност и понављање.

Сигурност се мери отпорношћу шифровања на нападе. Показало се да је алгоритам *AES* отпоран на све познате криптоаналитичке нападе. Алгоритам *AES* као блоковска шифра мора да обезбеди дифузију и нелинеарност. Дифузија представља ширење утицаја бита у току шифровања. Потпуна дифузија значи да сваки бит стања зависи од свих битова претходног стања. Две узастопне рунде алгоритма *AES* обезбеђују потпуну дифузију. Нелинеарност алгоритма *AES* потиче од табеле супституције и проширивања кључа. Нелинеарност у проширивању кључа обезбеђује да познавање дела кључа не омогућује једноставно одређивање осталих бита кључа.

Једноставност алгоритма *AES* је одређена тиме што су кораци алгоритма једноставни за имплементацију. Она повећава кредибилност тако што једноставни кораци сугеришу да се може лако разбити, па то покушава да уради више људи. Уколико се такви покушаји заврше неуспехом, шифра улива веће поверење.

Понављање представља вишеструку употребу истих функција у оквиру алгоритма. Поред предности које доноси може учинити шифру осетљивом на нападе. Конструкција алгоритма *AES* обезбеђује да понављање не изазива смањење сигурности. На то највише утиче коришћење константе *Rkon,* која се користи код формирања распореда кључа, односно код проширивања кључа.

Ефикасност представља однос брзине шифровања/дешифровања и мере у којој алгоритам користи ресурсе. Предност алгоритма *AES* је што може да се извршава на рачунарима и уређајима различите величине и снаге, од стоних рачунара до малих уређаја. Понављање омогућује да се паралелизацијом убрза шифровање/дешифровање. Поклапање редоследа корака за шифровање и дешифровање омогућује да се исти чип користи и за шифровање и за дешифровање. Једноставност обезбеђује да имплементације алгоритма *AES* буду једноставне и поздане [12].

Један од најбоље познатих напада на алгоритам *AES* представља би-клик напад. За разбијање кључа уз помоћ овог напада је потребно $2^{126,1}$ корака уколико се користи AES-128, $2^{189,7}$ уколико се користи *AES-192* и $2^{254,4}$ уколико се користи *AES-256* [13].

3. *ECB* и *CBC* режими рада

Постоји више режима рада блоковских метода за шифровање. Приликом имплементације коришћени су режими рада:

- *ECB* (енг. *Electronic Codebook*) и
- *CBC* (енг. *Cipher Block Chainig*) [14].

3.1. *ECB* режим рада

Нека су C_j, j-ти блок шифрата, K тајни кључ, P_j, j-ти блок отвореног текста, $ŠIF_K(X)$ блоковска функција шифровања са кључем K примењена на блок података X, $ŠIF^{-1}_K(X)$ блоковска функција инверзна функцији $ŠIF_K$ са кључем K примењена на блок података X и n је број блокова у отвореном тексту. Шифровање у режиму *ECB* се дефинише као:

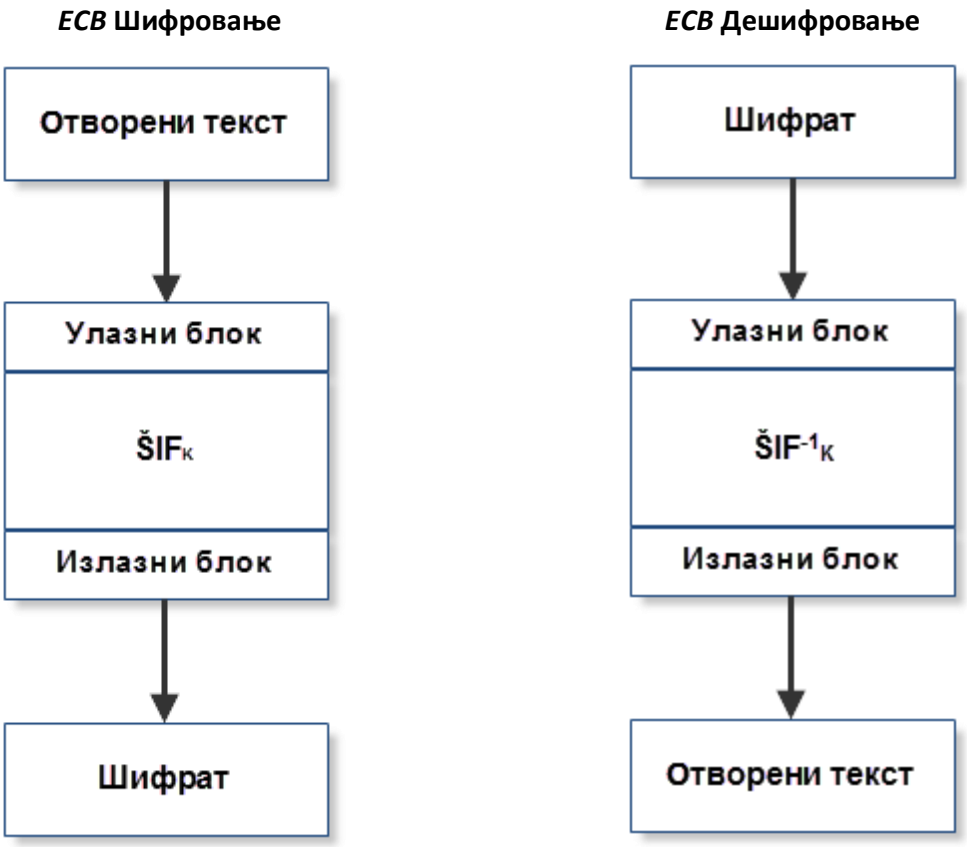
$$C_j = ŠIF_K(P_j)\ ,\ 1 \le j \le n,$$ а дешифровање као:

$$P_j = ŠIF^{-1}_K(C_j)\ ,\ 1 \le j \le n.$$

У режиму *ECB*, функција шифровања се примењује директно на сваки блок отвореног текста, независно у односу на остале. Резултирајући низ блокова формира шифрат.

ECB је најједноставнији и стандардни режим рада, али његов главни недостатак је да ако се примењује исти кључ на два иста отворена текста, добијени шифрати ће бити исти.

На слици 8 је приказано шифровање, односно дешифровање у режиму *ECB.*



Слика 8: Шифровање и дешифровање у режиму рада ECB

3.2. *CBC* режим рада

Режим рада *CBC* поред отвореног текста као улазни параметар користи и додатни блок података, (случајни, не-тајни) иницијализациони вектор *IV*.

Нека су C_j, *j*-ти блок шифрата, *K* тајни кључ, P_j, *j*-ти блок отвореног текста, $ŠIF_K(X)$ блоковска функција шифровања са кључем *K* примењена на блок података *X*, $ŠIF^{-1}_K(X)$ блоковска функција инверзна функцији $ŠIF_K$ са кључем *K* примењена на блок података *X, IV* иницијализациони вектор и *n* је број блокова у отвореном тексту.

Шифровање у режиму *CBC* се дефинише као:

$$C_1 = ŠIF_K(P_1 \oplus IV);$$

$C_j = ŠIF_K(P_1 \oplus C_{j-1})$, $2 \le j \le n$, а дешифровање као:

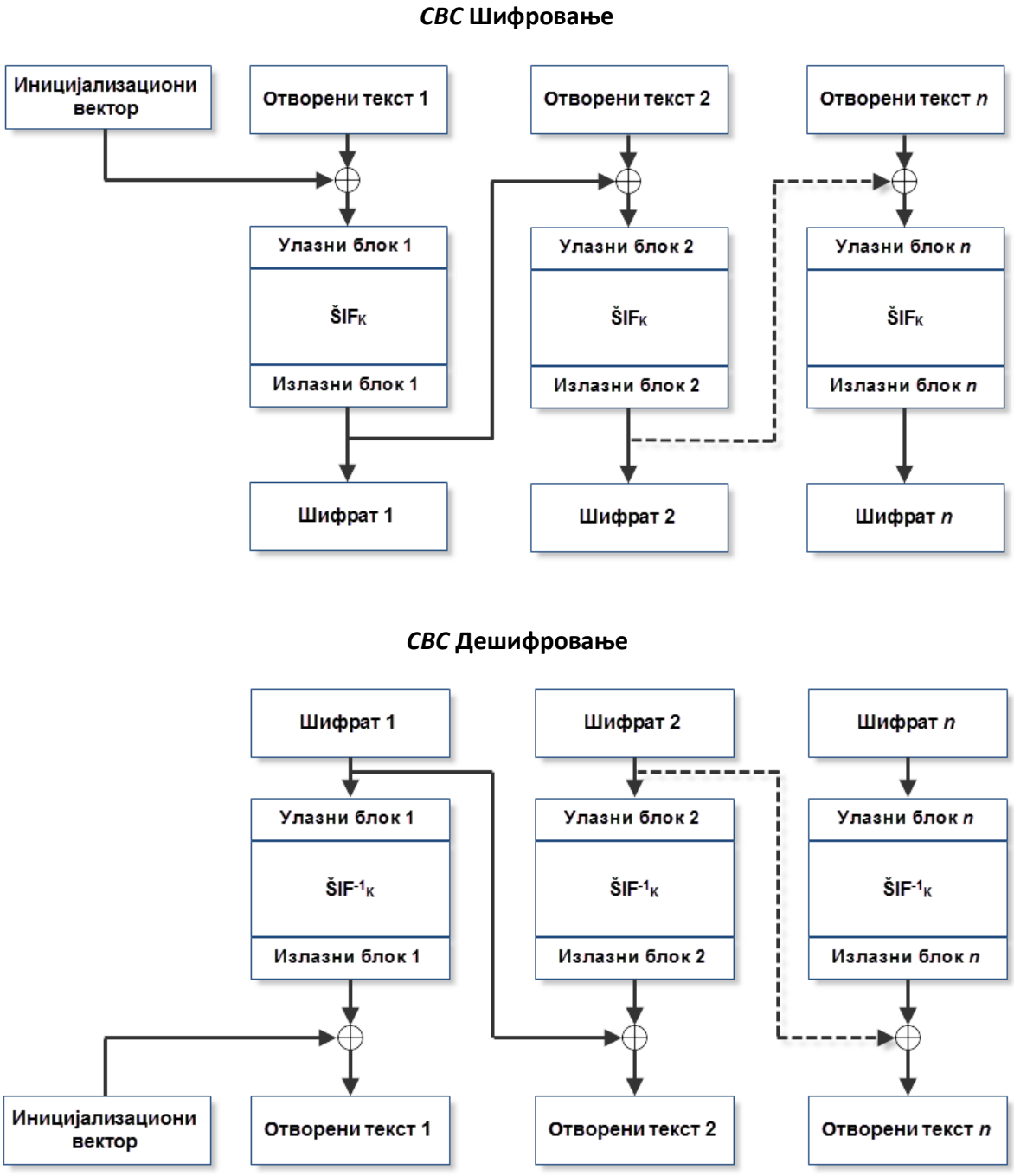
$$P_1 = ŠIF^{-1}{}_K(C_1) \oplus IV;$$

$$P_j = ŠIF^{-1}{}_K(C_j) \oplus C_{j-1},\ 2 \le j \le n.$$

У режиму *CBC* први блок шифрата се добије тако што се функција шифровања примени на блок текста добијен ексклузивном дисјункцијом првог блока отвореног текста и *IV*. Остали блокови шифрата се добијају применом функције шифровања на блок текста добијен применом ексклузивне дисјункције на одговарајући блок отвореног текста и претходни блок шифрата. Пошто сви блокови шифрата после првог зависе од претходног блока шифрата, паралелна имплементација није могућа у режиму *CBC.*

Дешифровање у режиму *CBC* се одвија тако што се инверзна функција шифровања примењује директно на блок шифрата, а након тога се извршава ексклузивна дисјункција са одговорајућим блоком. Први блок отвореног текста се формира тако што се примени инверзна функција шифровања на први блок шифрата и онда се изврши ексклузивна дисјункција са *IV.* Остали блокови отвореног текста се добијају применом инверзне функције шифровања на одговарајући блок шифрат и ексклузивном дисјункцијом са претходним шифратом. У случају дешифровања паралелизација је могућа, јер за добијање сваког наредног блока отвореног текста потребно је знати или *IV* или претходни блок шифрата, што је заправо низ улазних података овог режима. Још једна занимљивост овог режима је да у случају кад је *IV* неисправан, може да се дешифрује цео отворени текст, осим првог блока.

На слици 9 је приказано шифровање, односно дешифровање у режиму рада *CBC.*



Слика 9: Шифровање и дешифровање у режиму рада CBC

4. Познате имплементације алгоритма *AES* и примене

Алгоритам *AES* је један од најраспрострањенијих алгоритама шифровања. Постоје више познатих имплементације алгоритма *AES*. Као што је већ наведено алгоритам *AES* је слободан за употребу.

Познате библиотеке које садрже *AES* (по програмским језицима који су уписану у заградама):

- **CyaSSL** (C) – је имплементација отвореног кода *SSL* протокола за програмирање на *embedded* уређајима [15].
- **GnuTLS** (C) – је библиотека отвореног софтвера за безбедну комуникацију која имплементира *SSL, TLS* и *DTLS* [16].
- **Solaris Cryptographic Framework** (C/ASM) – је криптографска платформа која је саставни део оперативног система *Solaris* и садржи вишеструке имплементације алгоритма *AES* [17]
- ***Botan*** (C++) – је криптографска библиотека лиценцирана под *BSD* (енг. *Berkeley Software Distribution*) лиценцом и садржи имплементације многих криптографских алгоритма, међу којима и имплементацију алгоритма *AES* [18]
- **Java Cryptography Extension** (Java) – је званична стандардна екстензија за платформу Java [19]

Познате примене алгоритма *AES*:

- *AES* се користи код многих алата за архивирање и компресију – 7z, RAR, WinZIP, и других.
- Оперативни систем *Windows* за шифровање фајл система као примарни алгоритам користи *AES* од верзије *Windows XP SP1* (од верзије *Windows 7* користи комбиновани алгоритам, *AES* + *SHA* + *ECC*) [20]
- Стандард IEEE 802.11i (амандман стандарда IEEE 802.11) прописује употребу *AES*-128 у *CCMP* (енг. *Counter Mode CBC-MAC Protocol*) режиму као безбедносни механизам за бежичне мреже [21]
- IPsec (енг. *Internet Protocol Security*) користи *AES* за заштиту тајности [22]
- Апликација отвореног кода *Pidgin* која служи за директну размену порука између корисника, нуди могућност штићења порука алгоритмом *AES* кроз свој *OTR* (енг. *Off-the Record*) додатак [23].

Постоји више познатих имплементација алгоритма *AES* за платформу *CUDA.* Неке од занимљивих су имплементације из чланка „Имплементације и Анализе шифровања *AES* на *GPU*“ (енг. *Implementation and Analysis of AES Encryption on GPU* [24]) са конференције HPCC-ICESS 2012. Аутори овог чланка су такође имплементирали алгоритам *AES* за платформу *CUDA* у режимима рада *ECB* и *CBC,* међутим код њих главни акценат је на оптимизацији коришћења меморије графичке картице и на детекцији уских грла при раду имплементација.

За разлику од тога овај рад примарно се бави детаљима имплементација, различитим начинима паралелизације алгоритма *AES,* као и кључним разликама између режима рада *ECB* и *CBC*, односно секвенцијалних и паралелних верзија. Приказана је детаљнија анализа резултата између секвенцијалних и паралелних имплементација, али такође и између разних паралелних имплементација.

5. Платформа *CUDA*

CUDA је платформа за паралелно израчунавање и програмерски модел који омогућава вишеструко повећање перформанси израчунавања ефикасним искоришћењем снаге графичке процесорске јединице – *GPU* (енг. *Graphical Processing Unit*) [25]. У последњих 10 година микропроцесори су се развијали на два начина. Централне процесорске јединице *CPU* (енг. *CPU – Central Processing Unit*) су се развијале као вишејезгарне (енг. *multicore*) процесорске јединице, док графичке процесорске јединице су се развијале као многојезгарне (енг. *manycore*) процесорске јединице. Платформа *CUDA* је развијена од стране *nVidia* корпорације, која је позната по производњи графичких картица. Ово поглавље је великим делом засновано на званичном документу о раду са платформом *CUDA*, издатог од стране корпорације *nVidia* [26].

5.1. Основно о платформи *CUDA*

Платформа *CUDA* нуди велики скуп могућности за програмере и истраживаче у оквиру *CUDA* алата (енг. *CUDA Toolkit*) и *CUDA C/C++* програмских језика. Платформа *CUDA* користи принцип познат као *GPGPU* (енг. *General purpose computing on graphics processing units*), који означава коришћење *GPU* (која у основи служи за израчунавања везана за компјутерску графику) за израчунавања која су уобичајена за централну процесорску јединицу [27]. Скуп операција *GPU* је примитивнији од скупа операција централне процерсорске јединице *CPU*, зато што је основна сврха *GPU* да извршава израчунавања релевантна за компјутерску графику.

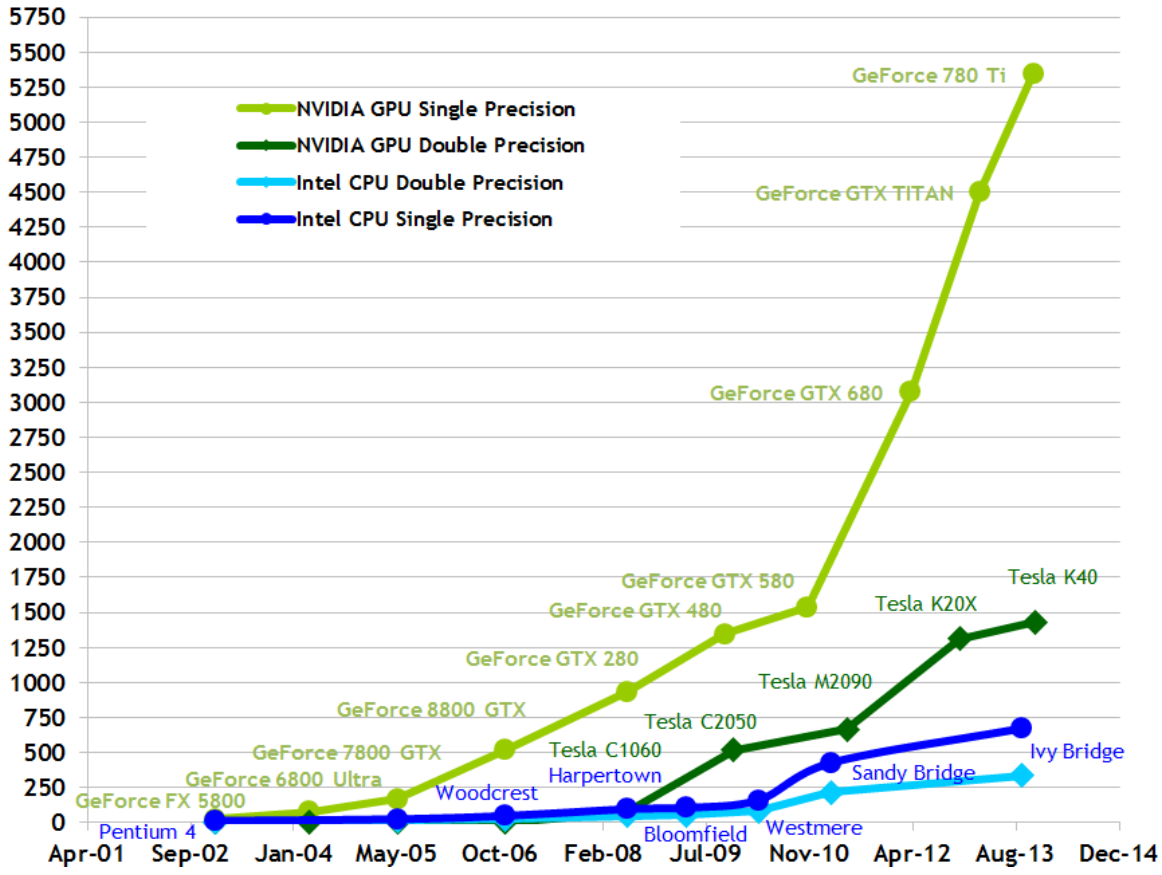
Платформа *CUDA* не омогућава потпуни паралелизам, већ користи програмерску парадигму проточног процесирања (енг. *Stream processing*). Проточно процесирање се заснива на *SIMD* принципу (енг. *Single instruction multiple data*), који омогућава искоришћење више процесорских јединица да паралелно израчунавају више инстанци истих функција над великим скупом података. Скуп података се посматра као ток над чијим елементима се паралелно извршава скуп операција (најчешће се извршава само једна операција над елементима тока). Из тог разлога коришћење *GPU* највише одговара проблемима који имају потребу да извршавају исти скуп инструкција над великом количином међусобно независних података.

Поред наведених програмерских алата постоје још многе могућности за рад са платформом *CUDA*. На пример, постоје многи пакети подршке програмским језицијама за

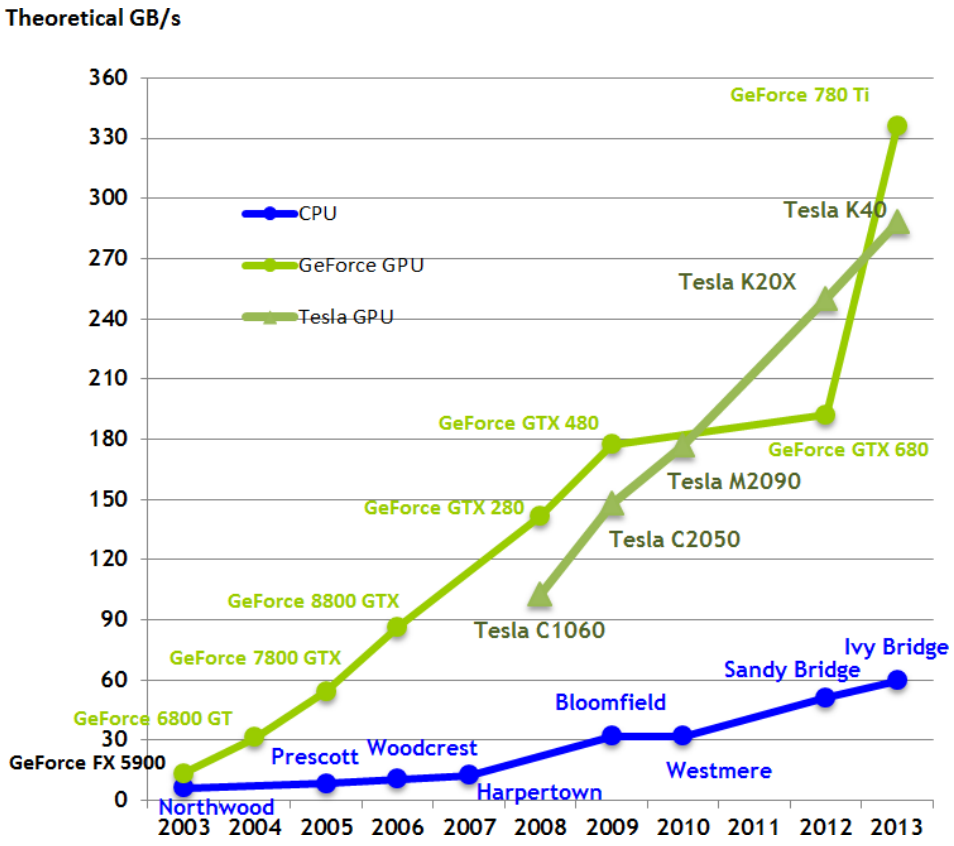
развој на платформи *CUDA* који нису развијани од стране корпорације *nVidia.* Неки од њих подржавају друге познате језике као што су: *JAVA, Python, C#, Ruby, MatLab* и други.

Већина процесорских чипова у практичној употреби су паралелни системи, пошто у последњих десет година микропроцесори су се развијали као вишејезгарне или многојезгарне процесорске јединице. Вишејезгарни приступ за циљ има да омогући извршавање већег броја секвенцијалних програма, док многојезгарни приступ унапређује ефикасност паралелног извршавања. Са константним унапређењем вишејезгарних *CPU* и многојезгарних *GPU* појавио се изазов да се апликативни софтвер развија тако да пресликава свој паралелизам на све већи број језгара. Суштина платформе *CUDA* је управо у томе да се превазиђе овај изазов, коришћењем основних програмских језика, као што је *C* програмски језик, без увођења нових, комплексних технологија. Платформа *CUDA* обезбеђује својим корисницима приступ виртуелном скупу инструкција, као и специфичном меморијском моделу који су основа рада са *CUDA GPU*.

На слици 10 и 11 су приказане разлике између водећих *GPU* и *CPU* по брзини извршавања као и протоку меморије.

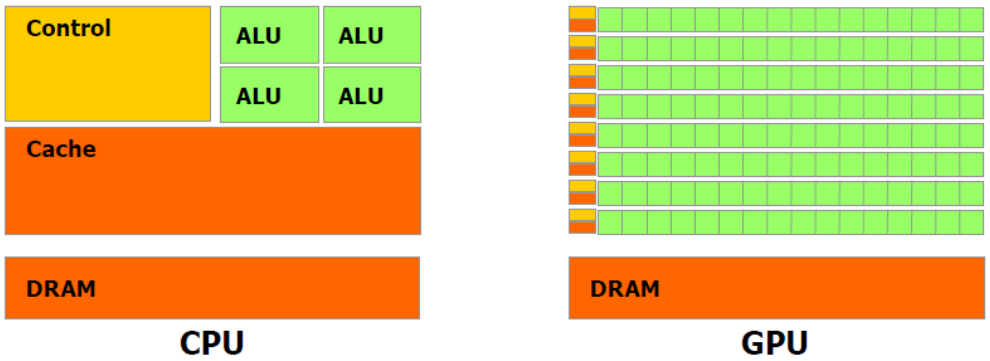


Слика 10: Теоријска разлика у брзини између GPU и CPU



Слика 11: Теоријска разлика у протоку меморије између GPU и CPU

GPU су испред *CPU* по брзини извршавања због своје основне сврхе, паралелног извршавања операција захтевних за израчунавање. Из тог разлога *GPU* су одличне за употребу код проблема који су оптерећени са много већим број извршавања аритметичких операција у односу на операције са меморијом. Нема потребе за великим кешом и великим нивоом контроле тока, пошто се исти програм извршава над великим скупом различитих података. Ова разлика је илустрована на слици 12.

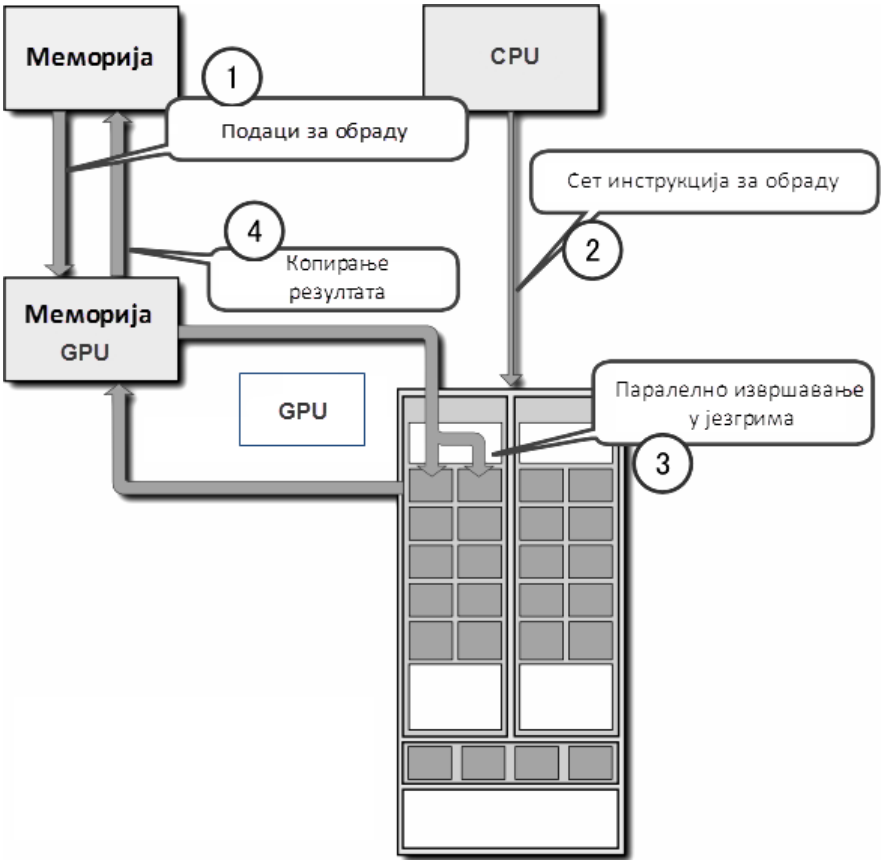


Слика 12: Разлика између CPU и GPU

Захваљујући ефикасном убрзању израчунавања уз помоћ графичке процесорске јединице *GPU,* платформа *CUDA* се користи у многим областима рачунарства, и шире. Неке од познатијих области где се *CUDA* користи су: биоинформатика, графичка обрада, истраживање података, безбедност, финансије, нумеричка анализа, временска истраживања, и друге.

5.2. Архитектура платформе *CUDA*

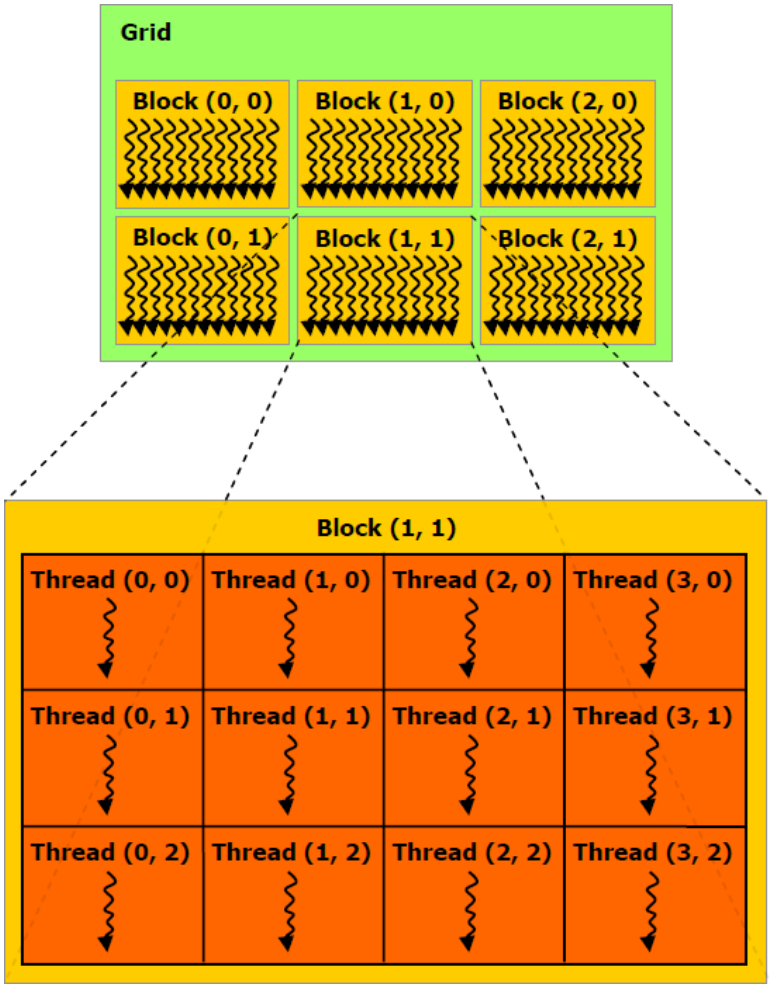
У тачки 5.1. наведено је да *GPU* користи проточно процесирање као основни модел рада. Скуп операција који се примењује на ток података састоји се од такозваних језгарних функција (енг. *kernel functions*). Програмски пакет *CUDA C* обезбеђује додатак програмском језику *C* тако да програмери могу да дефинишу језгарне функције у њему. У имплементацији алгоритма *AES* коришћен је овај пакет. За разлику од секвенцијалног извршавања у програмском језику *C* где се функција након позива изврши само једном, језгарна функција се након позива изврши *N* пута, и то паралелно на *N* различитих *CUDA* нити. Нити на *GPU* се умногоме разликују од нити на *CPU*, понајвише у великој брзини креирања нити и много једноставнијој контроли над нитима. На слици 13 је приказано како се одвија проток података и операција при програмирању на *GPU*.



Слика 13: Проток података и операција на GPU

Подаци се учитавају из спољне меморије у меморију *GPU*, инструкције се учитавају преко *CPU*. Након тога се инструкције паралелно извршавају над подацима у независним нитима, и на крају се резултат враћа у спољну меморију.

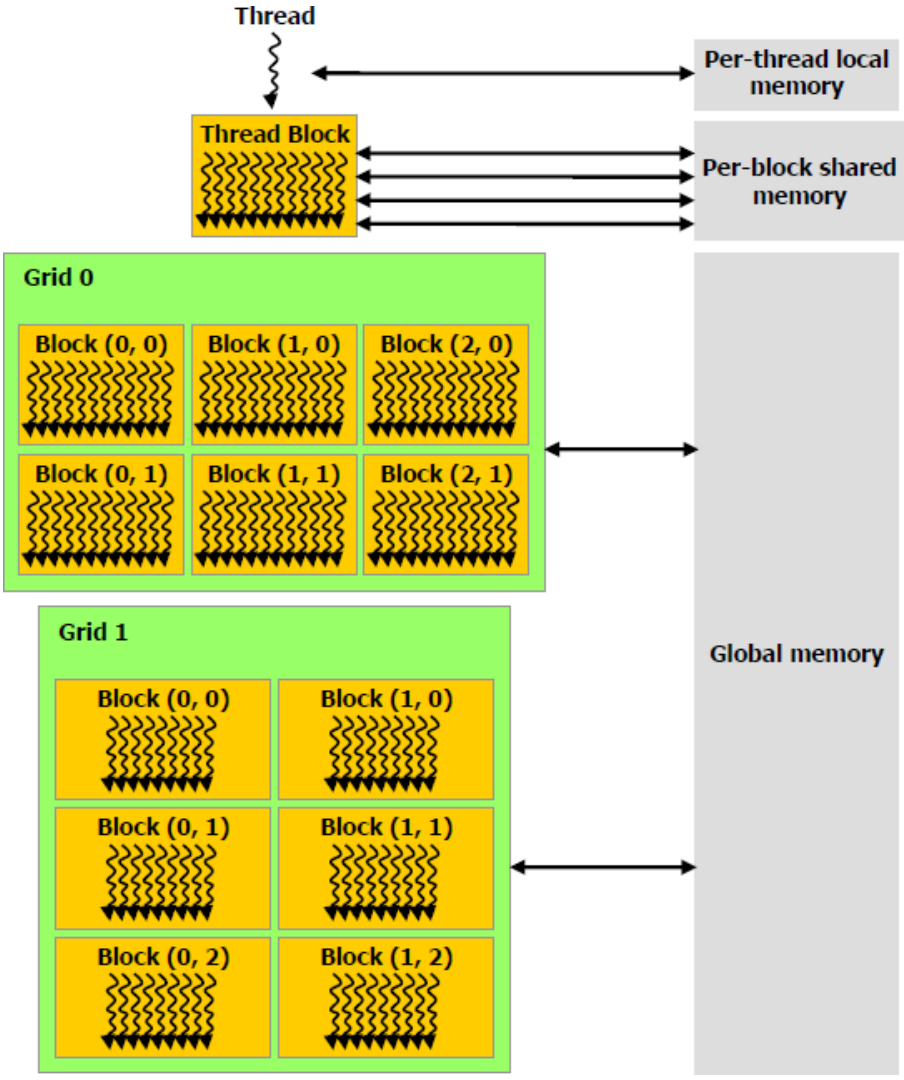
Нити се групишу у блокове, а блокови се даље групишу у решетке. Постоји ограничење колико један блок може садржати нити. На најновијим *GPU* тај број је 1024, као и на графичкој картици која је коришћена за имплементацију и тестирање алгоритма *AES*. Свака од *CUDA* нити која извршава једну језгарну функцију има свој идентификатор у оквиру блока и свакој од њих се може приступити кроз уграђени ***threadIdx*** идентификатор који одређује сваку нит. Идентификатори су заправо тродимензиони вектори и могу бити организовани тако да имају једну, две или три димензије. Блокови нити су распоређени у решетке које могу имати једну, две или три димензије заједно, што је илустровано на слици 14. Број блокова зависи од величине скупа података који се обрађује или броја процесора.



Слика 14: Решетке, блокови и нити

CUDA нити у раду комуницирају са неколико типова меморије, што је илустровано на слици 15. Свака нит има приступ својој локалној меморији, сваки блок нити има приступ заједничкој меморији дељивој на нивоу блока, и све нити свих блокова имају приступ глобалној меморији.

Нити које припадају истом блоку комуницирају ефикасније путем (заједничке) дељене меморије. За разлику од дељене меморије, постоји и глобална меморија *GPU* која је доступна свим нитима на *GPU*. На уштрб доступности, приступ овој врсти меморија је и до два реда величине спорији од приступа дељеној меморији на нивоу блока. Нити које се налазе у различитим блоковима не могу се синхронизовати и могу комуницирати једино преко глобалне меморије што много успорава рад апликација.



Слика 15: Типови меморије GPU и комуникација са меморијом

6. Имплементације алгоритма *AES*

У овом поглављу су описане секвенцијалне и паралелне имплементације алгоритма *AES* са њиховим режимима рада и упоредном анализом резултата. Приказано је пет различитих имплементација алгоритма *AES*, две секвенцијалне и три паралелне:

- Секвенцијална имплементација у режиму рада *ECB;*
- Секвенцијална имплементација у режиму рада *CBC;*
- Паралелна имплементација у режиму рада *ECB* (где се сваки корак алгоритма *AES* извршава над блоком текста);
- Паралелна имплементација појединачних корака у режиму рада *ECB* (где је сваки корак алгоритма *AES* имплементиран да се извршава над најмањим независним скупом података тог корака);
- Паралелна имплементација појединачних корака дешифровања у режиму рада *CBC.*

6.1. Основни детаљи имплементација

За секвенцијалну имплементацију коришћен је програмски језик *C,* док је за паралелну имплементацију коришћен програмски језик *CUDA C* и библиотека за развој *CUDA toolkit 4.0.17*. Имплементације су развијене у програмском окружењу *Microsoft Visual C++ 2010*.

Свака секвенцијална имплементација се састоји од:

- помоћних *C* функција (функција множења у пољу GF(2^8) – *GMul,* функција проширивање кључа која одговара операцији описаној у тачки 2.2., итд.),
- глобалних променљивих (низ *Rkon*, табела супституције из тачке 2.2. представљена као низ, итд.),
- *C* функција које одговарају трансформацијама дефинисаним у тачки 2.4. за шифровање, односно у тачки 2.5. за дешифровање и
- главне функције.

Свака паралелна имплементација се састоји од:

- помоћних *C* функција (функција проширивање кључа која одговара операцији описаној у тачки 2.2.),
- помоћних језгарних функција (функција множења у пољу GF(2^8) – *GMul,* описана у тачки 6.3., функција шифровања, функција дешифровања, итд*),*
- глобалних променљивих,
- језгарних функција које одговарају трансформацијама дефинисаним у тачки 2.4. за шифровање, односно у тачки 2.5. за дешифровање и
- главне функције.

Кључна разлика између имплементација у режиму рада *ECB* у односу на имплементације у режиму рада *CBC* (код паралелних имплементација у режиму рада *CBC* могуће је само дешифровање) је у додатној променљивој коју представља иницијализациони вектор *IV* и функцији која обавља ексклузивну дисјункцију на начин описан у тачки 3.2. Додатак А1 садржи програмски код те функције за паралелну имплементацију појединачних корака шифровања.

За сваку од трансформација дефинисаних у тачки 2.4. за шифровање, односно у тачки 2.5. за дешифровање, имплементирана је одговарајућа језгарна функција у паралелним имплементацијама. Додатак А2 садржи програмски код језгарне функције паралелне имплементације појединачних корака, која одговара трансформацији из тачке 2.4.1. *DodavanjeKljučaRunde*, која се идентично примењује и за шифровање и за дешифровање, као и програмски код како се та функција позива.

У секвенцијалним имплементацијама, у оба режимра рада, свака функција обрађује блок по блок дужине 16 бајта док се не обради цео улазни низ.

Приказане су три различите паралелне имплементације. Једна од кључних разлика између имплементација је начин на који паралелизују алгоритам *AES*. Имплементиране језгарне функције трансформација у паралелној имплементацији алгоритма *AES* у режиму рада *ECB* паралелно обрађују блокове података дужине 16 бајта, односно парелно извршавају алгоритам *AES*. Док друге две паралелне имплементације паралелно обрађују најмањи независни скуп података у оквиру једне трансформације (искључиво дешифровање кад је у питању паралелна имплементација у режиму рада *CBC*), односно паралелизују извршавање појединачних корака алгоритма *AES*. На пример, у трансформацији *ZamenaBajtova* описаној у тачки 2.4.2. најмањи независни скуп података је величине један бајт, тако да језгарна функција која њој одговара у имплементацији паралелно обрађује податке величине једног бајта. Слично томе, језгарна функција која одговара трансформацији *IzmenaKolona* описаној у тачки 2.4.4. паралелно обрађује податке величине четири бајта, пошто је у датој трансформацији најмањи независни скуп података величине четири бајта.

Паралелна имплементација у режиму рада *ECB* је занимљива за поређење са паралелном имплементацијом појединачних корака у режиму рада *ECB* јер показује да ли долази до убрзања када се обрађују различите величине података у језгарним функцијама.

Зарад прегледности, у парелелној имплементацији акције шифровања и дешифровања су издвојене у посебне функције. У наставку су издвојени најинтересантнији делови паралелних имплементација:

- Распоред кључева и табеле супституције су смештени у константну меморију *GPU* јер их језгарне функције користе у сваком пролазу. Програмски код декларисања константне меморије налази се у додатку А3.

- Текст који се шифрује, односно дешифрује је представљен низом хексадекадних вредности. Низ се копира у глобалну меморију *GPU* да би језгарне функције могле да га обрађују. У додатку А4 се налази програмски код алокације меморије *GPU* и копирања података из меморије домаћина у глобалну меморију *GPU.* Код паралелне имплементације у режиму рада *CBC* алоцира се и копира додатни низ на *GPU*. На првој позицији низа се налази иницијализациони вектор *IV,* а након тога шифрат скраћен за последњу реч. Овај низ је неопходан да би се извршила ексклузивна дисјункција која је последњи корак дешифровања у режиму рада *CBC.*
- У додатку А5 се налази програмски код у којем се дефинише број блокова и нити у зависности од величине улазног низа. Пошто један блок графичке картице коришћене за тестирање садржи највише 1024 нити, у случају да улазни низ садржи мање од 1024 речи нема потребе да се користи више од једног блока *GPU*. У супротном случају број блокова се поставља на одговарајући и број нити на димензију блока.
- За сваку језгарну функцију у паралелној имплементацији алгоритма *AES* у режиму рада *ECB* користи се максимално једна димензија решетке. Код паралелних имплементација појединачних корака алгоритма *AES* језгарне функције за исти улаз извршавају већи број нити од језгарних функција паралелне имплементације алгоритма *AES* у режиму рада *ECB*, зато што не раде искључиво са блоковима података величине 16 бајта већ са мањим целинама, и из тог разлога се користи више димензија решетке. У додатку А6 се налази програмски код како се одређује идентификатор нити код коришћења само једне димензије решетке и код коришћења више димензија решетке.

6.2. Помоћна функција проширивања кључа

Операција проширивања кључа се обавља функцијом *calculateNextKey*. Кључеви се формирају један по један у складу са начином описаним у тачки 2.3. Пошто се проширивање кључа извршава само једном нема потребе да се та акција одвија на *GPU*, тако да је проширивање кључа имплементирано на исти начин за све имплементације, уз помоћ функције *calculateNextKey* која враћа проширени кључ.

Функција *calculateNextKey* прима два параметра, *keys* - низ који садржи све до сад направљене кључеве и *i* – број рунде. Пре уласка у функцију поставља се почетни кључ на прво место у низу *keys*. Након тога функција се позива десет пута и формира се сваки следећи кључ на основу комбинације претходног кључа и броја рунде, тако да се на крају проширени кључ налази у низу *keys.*

Програмски код функције се налази у додатку А7.

6.3. Помоћна функција *GMul*

Као интересантна функција издваја се помоћна функција *GMul. GMul* је помоћна функција функцији која извршава трансформацију *IzmenaKolona* описаној у тачки 2.4.4. односно функцији која извршава трансформацију Трансформација *InverzIzmenaKolona*. Она омогућује множење у пољу GF(2^8). Програмски код функције за паралелне имплементације се налази у додатку А8.

Множење је засновано на методи описаној у тачки 2.1. Променљиве *a* и *b* представљају елементе множења, док се променљива *p* користи за израчунавање производа. Променљива *a* представља одговарајућу вредност из матрице из тачке 2.4.4. за шифровање, односно из тачке 2.5.3. за дешифровање, док променљива *b* представља једно слово матрице стања.

Множење се обавља у осам итерација у којима се извршава следећи низ операција. Прво се проверава да ли је постављен скроз десни бит променљиве која се множи (*b*), ако јесте формира се производ тако што се полимијално саберу променљиве *a* и *p.* Пази се да ли је скроз леви бит (који одговара коефицијенту x^7) променљиве *a* једнак 1, након тога се *a* помери за једно место улево, чиме се вредност скроз левог бита губи, а вредност скроз десног поставља на 0. Ова акција је идентична акцији множења полиномом *x,* описаној раније. Променљива *b* се дели полиномом *x*, односно, помера се удесно за једно место, чиме се одбацује скроз десни бит (који је искоришћен за формирање производа), а скроз леви бит се поставља на нулу. Последњи део сваког корака је провера да ли је било преноса, у случају да јесте било преноса, променљива *a* се полиномијално сабира са дефинишућим полиномом m(x) алгоритма *AES,* описаним у тачки 2.1.

У неким имплементацијама функције *Gmul,* користи се условни оператор *if*, који би требало избегавати у оквиру језгарних функција зато што може да успори извршавање функције [28]. У овој имплементацији коришћење условног оператора *if* је избегнуто применом битских оператора.

6.4. Тестирање

Спецификација персоналног рачунара коришћеног за тестирање се налази у додатку Б.

Отворени текст који се користи у тестирању се састоји од низа блокова хексадекадних вредности. Основна карактерстика отвореног текста је његова дужина и због тога је уведена следећа нотација. Отворени текст се назива *OTn*, где *n* представља броју слова у тексту. На пример, *OT16* садржи шеснаест слова, *OT64* садржи шездесет четири слова и тако даље. Отворени текст је подељен на блокове, где је сваки блок величине шеснаест бајта, односно шеснаест слова. Број блокова у сваком тексту је 4^y, где је $0 \leq y \leq 12$. OTn_i представња *i*-ти блок отвореног текста *OTn*. $AES_K(z)$ представља акцију шифровања блока *z* алгоритмом *AES* са кључем *K*.

Отворени текст се генерише на следећи начин, први блок у тексту је једнак почетном блоку *x*, а сваки следећи блок текста је једнак примени шифровања алгоритма *AES* на претходни блок:

$$OTn_1 = x,$$

$$OTn_i = AES(OTn_{i-1}) \text{ , где је } 1 \leq i \leq 4^y,\ n = 16 * 4^y,\ 1 \leq y \leq 12.$$

Коришћени кључ *K* и почетни блок *x* су дефинисани у додатку В.

Тест инстанце су текстуалне датотеке форматиране тако да у првом реду се налази број *n*, а потом отворени текст који се састоји од *n* слова. Тест инстанце се означавају *pn,* где *n* представља број слова отвореног текста у датотеци, на пример *p16*, *p64*, и тако даље. Број слова отвореног текста *n* се користи за откривање грешке при учитавању отвореног текста. Укупно је генерисано 13 тест инстанци. Програм за генерисање инстанци је написан у програмском језику *C.* Програмски код функције за генерисање инстанци се налази у додатку А9.

6.5. Позивање програма

Свака од имплементација се компајлира у извршну датотеку под називом *aes.exe*. Програми се извршавају следећим позивом:

aes.exe arg1 arg2 arg3

Први аргумент командне линије *arg1* резервисан је за ознаку акције која ће се извршити. За шифровање се користи аргумент „-e“, док се за дешифровање користи аргумент „-d”. Улазни подаци који се шифрују/дешифрују су задати у виду текстуалне датотеке која се прослеђује као други аргумент командне линије *arg2*. Улазни подаци у текстуалној датотеци морају да задовоље формат задат у тачки 6.4. Излазни подаци се формирају у виду текстуалне датотеке која се прослеђује као трећи аргумент командне линије *arg3*.

У додатку Г се налазе имена библиотека, односно скупа библиотека неопходних за извршење паралелних имплементација.

6.6. Резултати

У овој тачки су представљени резултати имплементација при коришћењу тест инстанци описаних у тачки 6.4.

6.6.1. Резултати у режиму рада *ECB*

У табелама 3 и 4 се налазе резултати шифровања, односно дешифровања, тест инстанци дефинисаних у тачки 6.4. следећих имплементација алгоритма *AES*:

- Секвенцијална имплементација у режиму рада *ECB;*
- Паралелна имплементација у режиму рада *ECB*;
- Паралелна имплементација појединачних корака у режиму рада *ECB*.

У првој колони табела 3 и 4 приказано је име инстанце на којој су рађени тестови. Затим је приказано време извршавања секвенцијалне, паралелне и паралелне имплементације појединачних корака у режиму рада *ECB* изражено у секундама.

Табела 3. Резултати шифровања у режиму рада ECB

Инстанца	Секвенцијална импл. *ECB* [с]	Паралелна импл. *ECB* [с]	Паралелна импл. појединачних корака *ECB* [с]
p16	<0,001	0,093	0,110
p64	<0,001	0,094	0,115
p256	<0,001	0,093	0,104
p1024	0,002	0,094	0,154
p4096	0,006	0,098	0,094
p16384	0,024	0,097	0,108
p65536	0,098	0,099	0,103
p262144	0,381	0,098	0,191
p1048576	1,565	0,103	0,104
p4194304	6,468	0,176	0,101
p16777216	24,564	0,290	0,219
p67108864	98,129	0,477	0,364
p268435456	392,751	1,323	0,843

Табела 4: Резултати дешифровања у режиму рада ECB

Инстанца	Секвенцијална импл. *ECB* [с]	Паралелна импл. *ECB* [с]	Паралелна импл. појединачних корака *ECB* [с]
p16	<0,001	0,092	0,101
p64	<0,001	0,093	0,122
p256	<0,001	0,093	0,101
p1024	0,002	0,105	0,099
p4096	0,006	0,085	0,101
p16384	0,024	0,097	0,109
p65536	0,094	0,095	0,105
p262144	0,391	0,096	0,140
p1048576	1,540	0,106	0,104
p4194304	6,266	0,120	0,115
p16777216	23,925	0,232	0,227
p67108864	95,229	0,399	0,352
p268435456	381,037	1,195	0,819

На графику 1 се налази графичка репрезентација вредности резултата табеле 3, а на графику 2 графичка репрезентација вредности резултата табеле 4. Због боље прегледности вредности резултата из табела су на графицима представљене као логоритам основе десет вредности изражених у милисекундама.

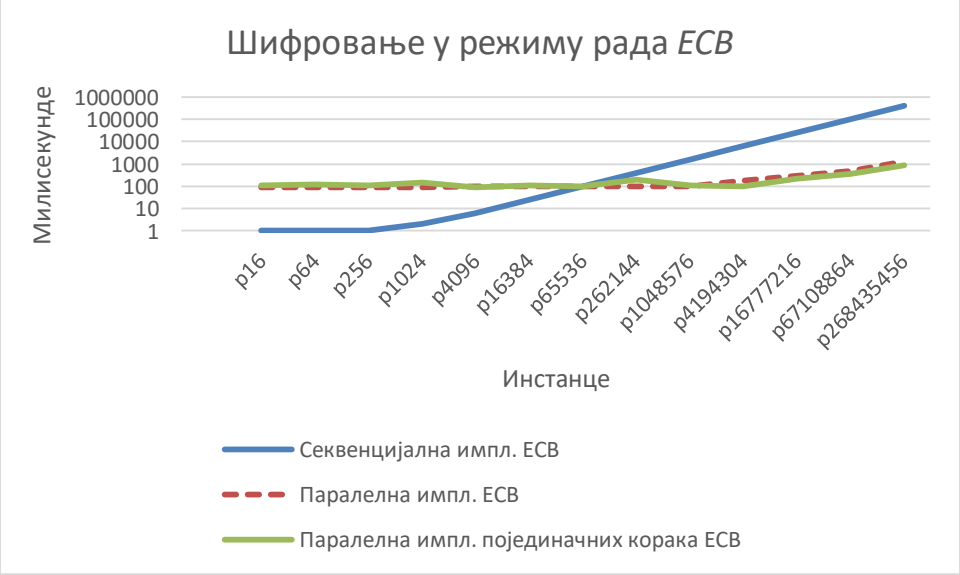


График 1: Шифровање у режиму рада ECB

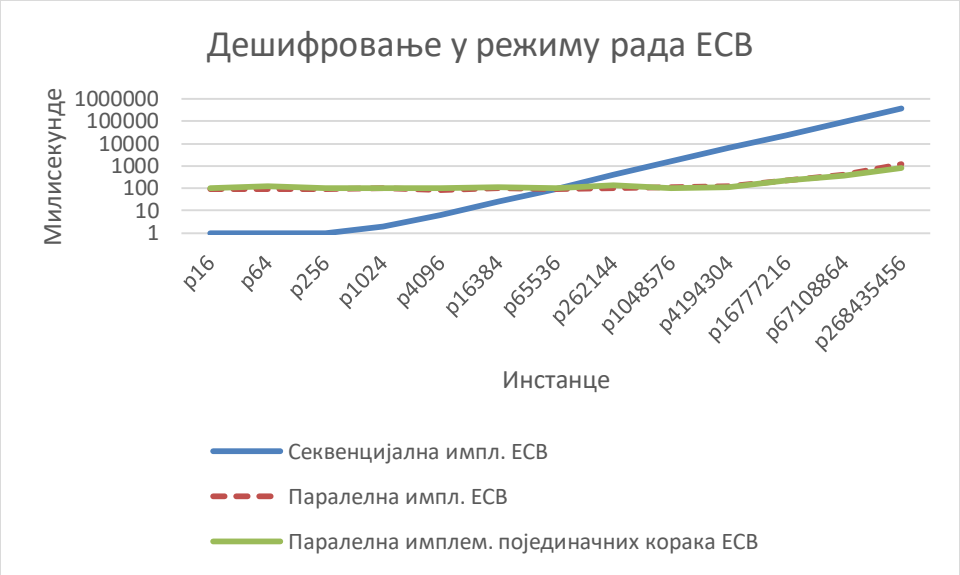


График 2: Дешифровање у режиму рада ECB

Вредности резултата код паралелних имплементација, за тест инстанце од *p16* до *p1048576,* су јако сличне зато што се за те тест инстанце користи само један блок нити. Са графика се види да је секвенцијална имплементација бржа у почетку, зато што паралелне имплементације за све инстанце мањих величина извршавају сличан процес (обрада једног блока), али за веће инстанце брзина извршавања паралелних имплементација јако споро расте, док код секвенцијалних имплементација долази до експоненцијалног успорења у односу на паралелне имплементације.

Што се више повећава димензија тест инстанци паралелне имплементације се све брже и брже извршавају у односу на секвенцијалну. Од инстанце *p262144* време извршавања секвенцијалне имплементације почиње значајно да расте. Већ од следеће инстанце паралелне имплементације се извршавају за ред величине брже од секвенцијалне. За последње тестирану инстанцу, *p268435456*, паралелне имплементације се извршавају скоро 400 пута брже од секвенцијалне.

6.6.2. Резултати дешифровања у режиму рада *CBC*

У табели 5 се налазе резултати дешифровања тест инстанци дефинисаних у тачки 6.4. следећих имплементација алгоритма *AES*:

- Секвенцијална имплементација у режиму рада *CBC;*
- Паралелна имплементација појединачних корака дешифровања у режиму рада *CBC.*

У првој колони табеле 5 приказано је име инстанце на којој су рађени тестови. Затим је приказано време извршавања дешифровања секвенцијалном имплементацијом у режиму рада *CBC* и након тога дешифровања паралелном имплементацијом појединачних корака у режиму рада *CBC*, изражено у секундама.

Табела 5: Резултати дешифровања у режиму рада CBC

Инстанца	Секвенцијална импл. *CBC* [с]	Паралелна импл. појединачних корака *CBC* [с]
p16	<0,001	0,201
p64	<0,001	0,162
p256	<0,001	0,161
p1024	0,002	0,126
p4096	0,006	0,191
p16384	0,024	0,216
p65536	0,094	0,159
p262144	0,382	0,213
p1048576	1,517	0,256
p4194304	6,160	0,375
p16777216	23,143	0,796
p67108864	91,564	2,516
p268435456	366,993	9,397

На графику 3 се налази графичка репрезентација вредности резултата табеле 5. Због боље прегледности вредности резултата из табеле су на графику представљене као логоритам основе десет вредности изражених у милисекундама.

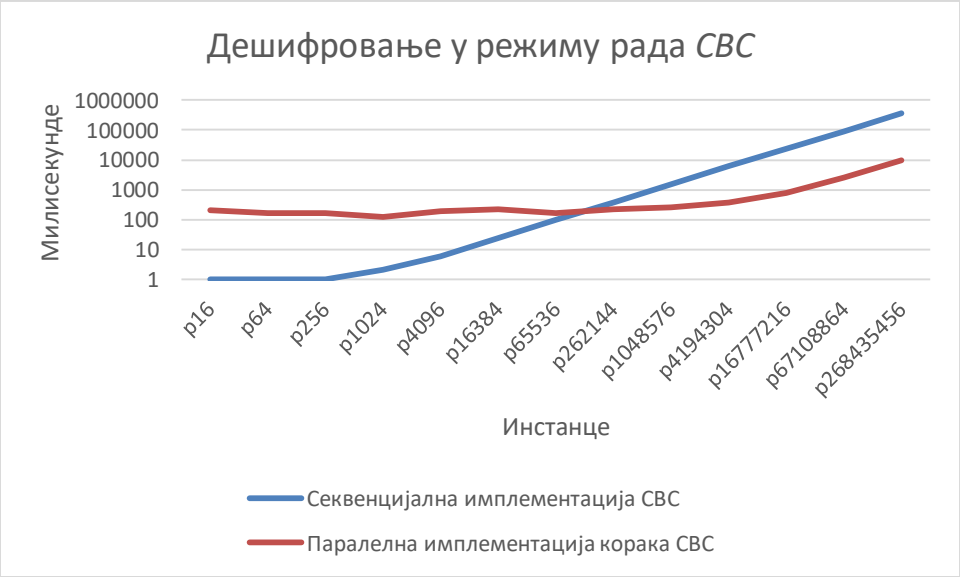


График 3: Дешифровање у режиму рада CBC

Слично као у тачки 6.6.1. секвенцијална имплементација се у почетку доста брже извршава у односу на паралелну имплементацију, пошто паралелна имплементација увек извршава обраду једног блока, али за веће тест инстанце резултати секвенцијалне имплементације су доста спорији од резултата паралелне имплементације.

На тест инстанци *p262144* паралелна имплементација се први пут изврши неколико пута брже од секвенцијалне (око 4 пута брже). Што се више повећава димензија улаза паралелна имплементација је све бржа од секвенцијалне. Последње тестирану инстанцу, *p268435456*, паралелна имплементација извршава чак 40 пута брже. Разлика између секвенцијалне и паралелне имплементације дешифровања у режиму рада *CBC* је мања када се упореди са разликом између секвенцијалне и паралелних имплементација дешифровања у режиму рада *ECB*, али режим рада *CBC* је безбеднији и добијено убрзање паралелизацијом појединачних корака је значајно велико.

6.6.3. Упоредна анализа дешифровања у паралелним имплементацијама

У овој тачки је приказано поређење између паралелних имплементација с циљем да се изведу закључци у односу између њих.

У табели 6 се налазе резултати дешифровања тест инстанци дефинисаних у тачки 6.4. следећих имплементација алгоритма *AES*:

- Паралелна имплементација у режиму рада *ECB*;
- Паралелна имплементација појединачних корака у режиму рада *ECB;*
- Паралелна имплементација појединачних корака дешифровања у режиму рада *CBC.*

У првој колони табеле 6 приказано је име инстанце на којој су рађени тестови. Затим су приказана времена извршавања паралелних имплементација, редом, у режиму рада *ECB*, имплементација појединачних корака у режиму рада *ECB* и имплементација појединачних корака дешифровања у режиму рада *CBC,* изражено у секундама.

Табела 6: Упоредни резултати дешифровања паралелних импелментација

Инстанца	Паралелна импл. *ECB* [с]	Паралелна импл. појединачних корака *ECB* [с]	Паралелна импл. појединачних корака *CBC* [с]
p16	0,092	0,101	0,201
p64	0,093	0,122	0,162
p256	0,093	0,101	0,161
p1024	0,105	0,099	0,126
p4096	0,085	0,101	0,191
p16384	0,097	0,109	0,216
p65536	0,095	0,105	0,159
p262144	0,096	0,140	0,213
p1048576	0,106	0,104	0,256
p4194304	0,120	0,115	0,375
p16777216	0,232	0,227	0,796
p67108864	0,399	0,352	2,516
p268435456	1,195	0,819	9,397

На графику 4 се налази графичка репрезентација вредности резултата из табеле 6. Због боље прегледности вредности резултата из табеле су на графику представљене као логоритам основе десет вредности изражених у милисекундама.

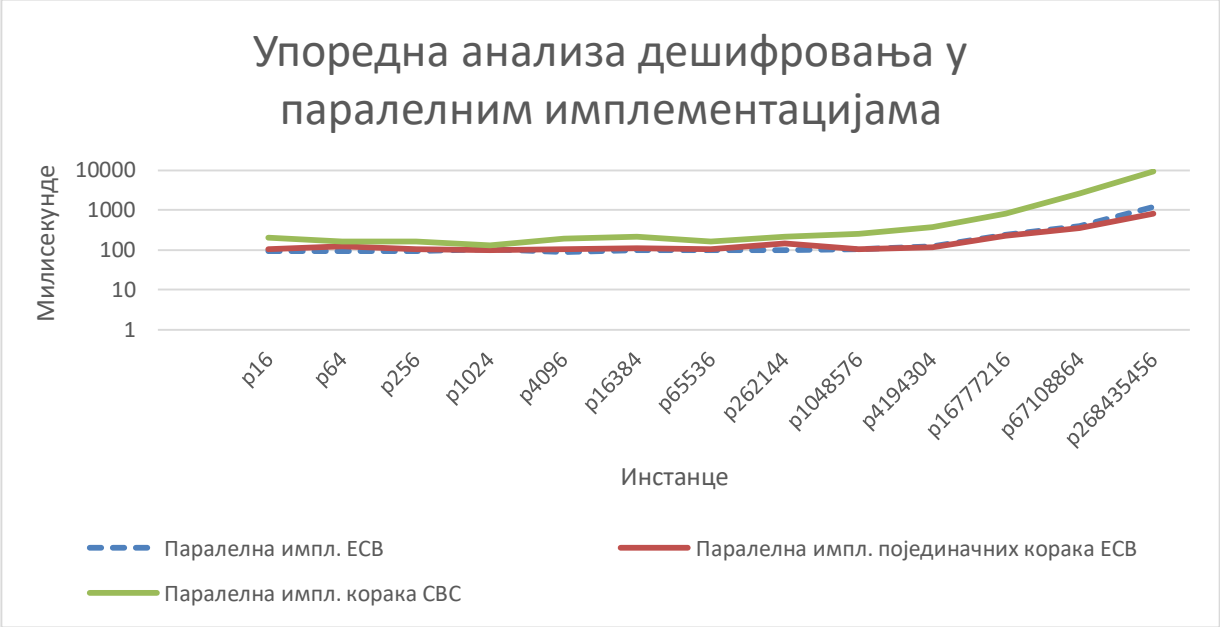


График 4: Упоредни резултати дешифровања паралелних импелментација

Очекивано, убедљиво најспорија је паралелна имплементација појединачних корака алгоритма *AES* у режиму рада *CBC*, пошто користи додатну функцију и додатни низ података у односу на остале.

Паралелна имлементација у режиму рада *ECB* и паралелна имплементација појединачних корака у режиму рада *ECB* извршавају се сличном брзином, са неким разликама, поготово за веће тест инстанце. Од тест инстанце *p4194304* паралелне имплементације почињу да користе више од једног блока нити. Почевши од те инстанце паралелна имплементација у режиму рада *ECB* се извршава све спорије у односу на паралелну имплементацију појединачних корака у режиму рада *ECB.* За првих пар тест инстанци та разлика је у пар процената да би за највећу тест инстанцу достигла 30%. Из овога се може закључити да паралелизацијом појединачних корака алгоритма *AES* се постиже убрзање, тј. да проблеми са великим скупом улазних података који се решавају паралелизацијом ће се брже извршавати ако се улазни подаци језгарних функција разбију на најмање независне целине, наравно под условом да је то могуће за дати проблем.

6.6.4. Ефикасност паралелизације

Убрзање паралелизације се рачуна као количник времена извршавања паралелног алгоритма и времена извршавања одговарајућег секвенцијалног алгоритма. Ефикасност паралелизације се даље рачуна као количник убрзања паралелизације и броја употребљених процесора (језгара), те ефикасност представља степен повећања убрзања приликом додавања новог процесора.

У табели 7 се налазе убрзања и ефикасности паралелизације дешифровања у режиму рада *EBC*. У првој колони табеле 7 приказано је име инстанце, а затим су приказани број искоришћених процесора, убрзање и ефикасност паралелизације. Приказани су резултати за највећих 6 тест инстанци, пошто су ефекти убрзања паралелизације почели да постају видљиви тек на тим инстанцама.

Табела 7: Ефикасност паралелизације дешифровања алгоритма AES у режиму рада EBC

Инстанца	Број искоришћених процесора	Убрзање *EBC*	Ефикасност *EBC*
p16384	16384	0,895238	0,0000009
p65536	65536	2,792857	0,0000007
p262144	262144	14,80769	0,0000009
p1048576	1048576	54,48696	0,0000008
p4194304	4194304	105,3965	0,0000004
p16777216	16777216	270,5369	0,0000003

График 5 описује функцију ефикасности парелелизације дешифровања алгоритма *AES* у режиму рада *EBC*, на основу табеле 7.

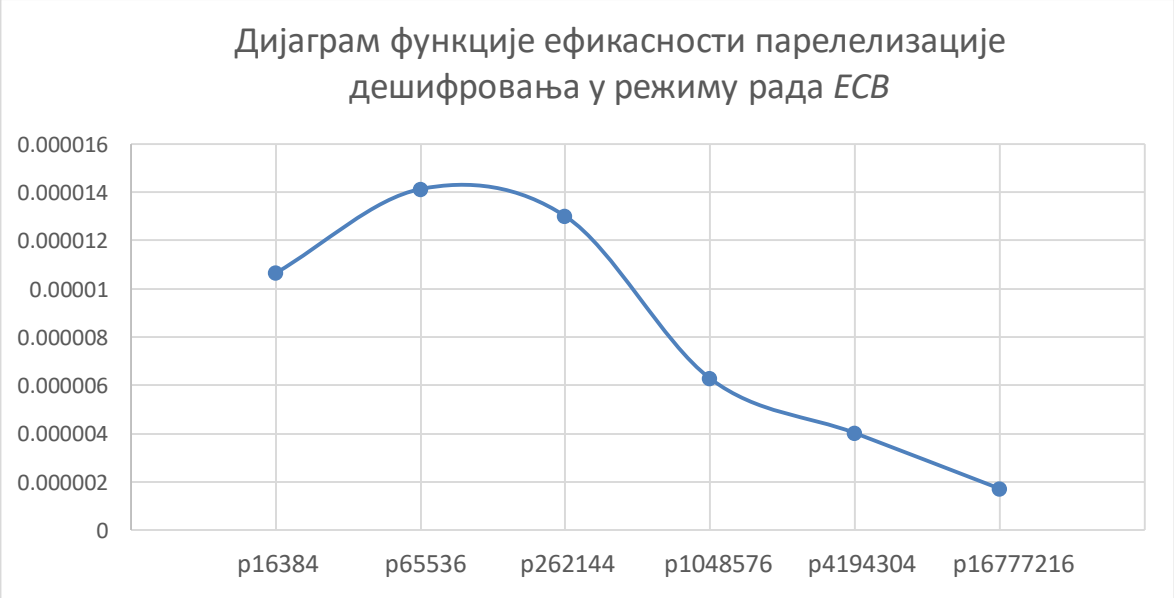


График 5: Дијаграм функције ефикасности паралелизације дешифровања у режиму рада ECB

На основу графика 5 се може закључити да ефикасност након неког броја процесора почиње да опада. То значи да се и релативно убрзање паралелног алгоритма додавањем нових процесора смањује па је самим тим и убрзање паралелизације ограничено одозго.

У табели 8 се налазе резултати ефикасности паралелизације дешифровања у режиму рада *CBC*. У првој колони табеле 8 приказано је име инстанце, а затим су приказани број искоришћених процесора, убрзање и минимална ефикасност парелелизације. Слично као и код режима рада *ECB*, и овде су приказани резултати само за највећих 6 инстанци. Разлика у односу на *ECB* режим је то што различите језгарне функције користе различите степене паралелизације, тако да негде је блок од четири бајта распоређен на један процесор, а негде је сваки појединачан бајт распоређен на појединачан процесор. То у неку руку отежава анализу зависности од употребљених процесора, јер се њихов број мења кроз различите језгарне функције. У табели 8 приказан је најмањи број процесора коришћен у некој језгарној функцији по инстанци, односно ефикасност у најгорем случају, ради лакше упоредиве анализе. Из аспекта рачунања ефикасности паралелизације ово поједностављење не нарушава облик криве ефикасности, јер вредности убрзања за сваку инстанцу су подељене са истим константним фактором.

Табела 8: Минимална ефикасност парелелизације дешифровања алгоритма AES у режиму рада CBC

Инстанца	Број искоришћених процесора	Убрзање *CBC*	Ефикасност *CBC*
p262144	4096	1.793427	0.00000171
p1048576	16384	5.925781	0.00000141
p4194304	65536	16.42667	0.00000098
p16777216	262144	29.07412	0.00000043
p67108864	1048576	36.39269	0.00000013
p268435456	4194304	39.05427	0.00000003

График 6 описује функцију ефикасности парелелне имплементације појединачних корака дешифровања алгоритма *AES* у у режиму рада *CBC*, на основу табеле 8.

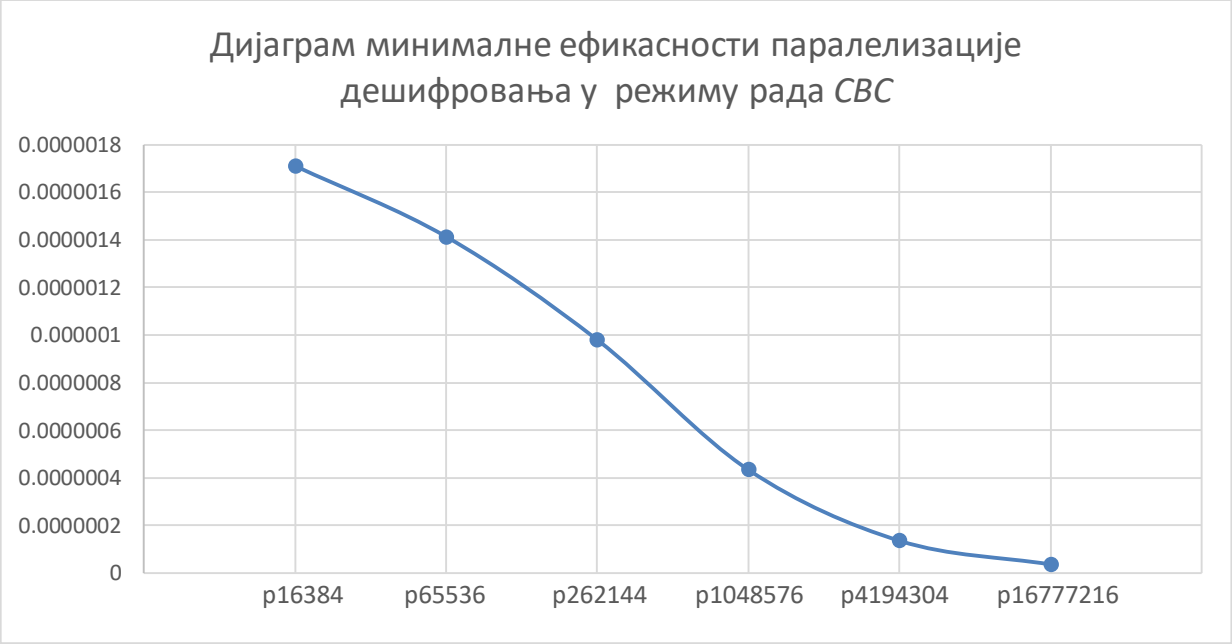


График 6: Дијаграм минималне ефикасности паралелизације дешифровања у режиму рада CBC

Као и у режиму рада *ECB*, може се закључити да постоји горња граница убрзања, јер је функција минималне ефикасности опадајућа, почев од неког броја процесора.

7. Закључак

У овом раду је разматран проблем ефикасности шифровања и дешифровања података на графичкој процесорској јединици, у различитим режимима рада. За акције шифровања и дешифровања коришћен је индустријски алгоритам *AES*, у режимима рада *ECB* и *CBC*. Приказане су његове паралелне верзије на платформи *CUDA* које користе различите режиме рада, као и различит начин паралелизације алгоритма. У једној верзији се паралелизује извршавање алгоритма, а у друге две извршавање појединачних корака алгоритма. Такође је приказано и неколико одговарајућих секвенцијалних верзија алгоритма, зарад поређења.

На инстанцама малих димензија, секвенцијални алгоритми су очекивано били упоредиви по питању ефикасности. Међутим, како је димензија проблема расла, паралелне верзије су почеле да се извршавају вишеструко пута брже у односу на секвенцијалне. Што се више повећавала димензија тест инстанци, то се и убрзање повећавало и достигло фактор 400 у режиму рада *ECB*, односно фактор 40 у режиму рада *CBC*, на највећој анализираној инстанци.

Међутим ефикасност паралелизације (убрзање/број процесора) почиње да се смањује након увођења неког броја нових процесора с обзиром на то да се додатно време троши на синхронизацију процесора. То значи да тренд раста убрзања због смањења ефикасноти паралелизације успорава, тако да постоји горња граница убрзања. Тачка опадања ефикасности паралелизације би могла да се помери бољим хардвером који има већу процесорску моћ, већу

ширину меморијске магистрале и већи проток података у секунди (на пример на *nVidia Tesla* графичка картица [29]).

Постигнути резултати показују да је за проблеме који немају превише комплексне операције израчунавања, а имају велики број улазних података ефикасније решавати паралелно на графичким процесорским јединицама, односно на платформи *CUDA*.

Тестирана је и разлика између паралелне верзије која паралелизује алгоритам *AES* и паралелне верзије која паралелизује кораке алгоритма *AES.* Утврђено је да паралелизацијом појединачних корака алгоритма долази до убрзања у односу на паралелизацију алгоритма, односно да решавање паралелизацијом проблема са великим скупом улазних података који немају превише комплексне операције, израчунавања се брже извршавају ако је могуће сам проблем разбити на мање подпроблеме и решавати их као мање целине.

8. Референце

1. Schaefer, Edward. "An introduction to cryptography and cryptanalysis." *California's Silicon Valley: Santa Clara University* (2009), стране 4-5.
2. Kahn, David. *The Codebreakers: The comprehensive history of secret communication from ancient times to the internet*. Simon and Schuster, 1996.
3. Singh, Simon. *The code book: the evolution of secrecy from Mary, Queen of Scots, to quantum cryptography*. Doubleday, 1999. стране 14-20.
4. Tuchman, Walter. "A brief history of the data encryption standard." *Internet besieged*. ACM Press/Addison-Wesley Publishing Co. (1997) стране 275-280.
5. Standard, N. F. "Data Encryption Standard (*DES*)." *Federal Information Processing Standards Publication* (1999).
6. Menezes, Alfred J., Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996..
7. B Barker, William C., and Elaine Barker. "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher: *NIST* Special Publication 800-67, Revision 2." (2012).
8. Richards, Mitchell C. "AES: The Making of a New Encryption Standard." (2001).
9. Daemen, Joan, and Vincent Rijmen. "AES proposal: Rijndael." Document version 2 (1999).
10. Daemen, Joan, and Vincent Rijmen. "AES proposal: Rijndael." National Institute of Standards and Technology (1998).
11. Standard, N. F. "Advanced Encryption Standard (AES)." *Federal Information Processing Standards Publication* (2001).
12. Миодраг Живковић. „Криптографија“ *Математички факултет* (2009) стране 30-31.
13. Bogdanov, Andrey, Dmitry Khovratovich, and Christian Rechberger. „Biclique cryptanalysis of the full AES.“ *International Conference on the Theory and Application of Cryptology and Information Security*. Springer Berlin Heidelberg, 2011., страна 1.

14. Dworkin, Morris. *Recommendation for block cipher modes of operation. methods and techniques*. No. *NIST*-SP-800-38A. NATIONAL INST OF STANDARDS AND TECHNOLOGY GAITHERSBURG MD COMPUTER SECURITY DIV, 2001.
15. "WolfSSL - Embedded SSL Library for Applications, Devices, IoT, and the Cloud." *WolfSSL - Embedded SSL Library for Applications, Devices, IoT, and the Cloud*. N.p., n.d. Web. 14 Sept. 2016.
16. "Gnutls.org." *GnuTLS*. N.p., n.d. Web. 14 Sept. 2016.
17. Oracle Corporation. "Chapter 13 Solaris Cryptographic Framework." *System Administration Guide: Security Services*. Oracle (2010).
18. "Botan." *: Crypto and TLS for C 11* —. N.p., n.d. Web. 14 Sept. 2016.
19. "Java Cryptography Architecture Oracle Providers Documentation for Java Platform Standard Edition 7." *Java Cryptography Architecture Oracle Providers Documentation*. N.p., n.d. Web. 14 Sept. 2016.
20. "Encrypting File System." *Wikipedia*. Wikimedia Foundation, n.d. Web. 14 Sept. 2016.
21. Cam-Winget, Nancy, et al. "IEEE 802.11 i Overview." *NIST 802.11 Wireless LAN Security Workshop*. 2002.
22. Manral, Vishwas. "Cryptographic algorithm implementation requirements for encapsulating security payload (esp) and authentication header (ah)." (2007).
23. @bestVPN_com. "Secure Instant Messaging with Pidgin plus OTR - BestVPN.com." *BestVPNcom Secure Instant Messaging with Pidgin plus OTR Comments*. N.p., 2014. Web. 14 Sept. 2016.
24. Li, Qinjian, et al. "Implementation and analysis of AES encryption on GPU."*High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE, 2012.
25. "Parallel Programming and Computing Platform.“ nVidia CUDA N.p., n.d. Web. 14 Sept. 2016.
26. "CUDA C Programming Guide Version 4.2. ", nVidia CUDA (2012).
27. Fung, James, and Steve Mann. "Computer vision signal processing on graphics processing units." *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*. Vol. 5. IEEE, 2004. стране 93-96.
28. "Avoiding If Condition in CUDA - BurnIgnorance." *BurnIgnorance*, N.p., 2015. Web. 14 Sept. 2016.
29. "Gigabyte GeForce GTX 970 WindForce 3X OC vs Nvidia Tesla K40 | Compare Graphics Cards." *VERSUS*. N.p., n.d. Web. 22 Sept. 2016.

Додатак А – програмски кодови

А1. Функција ексклузивне дисјункције код паралелне имплементације појединачних корака дешифровања

Табела 9: Програмски код језгарне функције експлозивне дисјункције паралелне имплементације корака дешифровања

```
__global__ void applyCBC(unsigned char *state, unsigned char *dev_original_cipher)
{
	int threadsPerBlock = blockDim.x * blockDim.y;
	int threadNumInBlock = threadIdx.x + blockDim.x * threadIdx.y;
	int blockNumInGrid = blockIdx.x + gridDim.x * blockIdx.y;

	int tid = blockNumInGrid * threadsPerBlock + threadNumInBlock;
	state[tid]^=dev_original_cipher[tid];

	__syncthreads();
}
```

А2. Функција трансформације *DodavanjeKључаRunde*

Табела 8: Програмски код језгарне функције DodavanjeKључаRunde у паралелној имплементацији појединачних корака

```
__global__ void cudaAddRoundKey(unsigned char *state, unsigned int keyOrder)
{
	int threadsPerBlock = blockDim.x * blockDim.y;
	int threadNumInBlock = threadIdx.x + blockDim.x * threadIdx.y;
	int blockNumInGrid = blockIdx.x + gridDim.x * blockIdx.y;

	int tid = blockNumInGrid * threadsPerBlock + threadNumInBlock;

	int keyofset =keyOrder*KEY_BYTE_SIZE+tid%KEY_BYTE_SIZE;
		state[tid]^=dev_keys[keyofset];

	__syncthreads();
}
```

Табела 9:: Програмски код позива језгарне функције DodavanjeKључаRunde за паралелну имплементацију појединачних корака

```
cudaAddRoundKey<<<partitionBlocks(blocks*KEY_BYTE_SIZE),threads>>>(dev_result,0);
if(cudaCheck("cudaAddRoundKey"))
	goto Error;
```

А3. Константна меморија на *GPU*

Табела 10: Програмски код декларације константне меморије на GPU

```
__constant__ unsigned char dev_S_BOX[256];
__constant__ unsigned char dev_S_BOX_INVERSE[256];
__constant__ unsigned char dev_keys[11*KEY_BYTE_SIZE];
```

А4. Алокација меморије и копирање података на *GPU*

Табела 11: Програмски код алокације меморије на GPU и копирања података са домаћина на GPU

```
cudaStatus = cudaMalloc((void**)&dev_result, wordCount * KEY_BYTE_SIZE* sizeof(unsigned char));
if (cudaStatus != cudaSuccess)
        {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
        }

cudaStatus = cudaMemcpy(dev_result, result, wordCount * KEY_BYTE_SIZE* sizeof(unsigned char),cudaMemcpyHostToDevice);
if (cudaStatus != cudaSuccess)
        {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
        }
```

А5. Дефинисање броја блокова и нити

Табела 12: Програмски код дефинисања броја блокова и нити потребних за обрађивање улазног низа

```
        If(wordCount<1024)
        {
                blocks=1;
                threads=wordCount;
        }
        else
        {
                blocks=wordCount/1024;
                threads=1024;
        }
```

А6. Разлике у коришћењу једне и више димензија решетке

Табела 103: Програмски код за одређивање идентификатора нити у случају коришћења једне или више решетки

Коришћење само једне димензије решетке	int tid = blockIdx.x*blockDim.x + threadIdx.x;

Коришћење више димензија решетке	int threadsPerBlock = blockDim.x * blockDim.y;
	int threadNumInBlock = threadIdx.x + blockDim.x * threadIdx.y;
	int blockNumInGrid = blockIdx.x + gridDim.x * blockIdx.y;
	
	int tid = blockNumInGrid * threadsPerBlock + threadNumInBlock;

А7. Функција проширивање кључа *calculateNextKey*

Табела 11: Програмски код C функције calculateNextKey

```
void calculateNextKey(unsigned char *keys, int rcon_i)
{
	unsigned char keyResult[KEY_BYTE_SIZE];
	unsigned char keyPrevious[KEY_BYTE_SIZE];
	int i,j;

	for(i=0; i<KEY_BYTE_SIZE; i++)
		keyPrevious[i]=keys[(rcon_i-1)*KEY_BYTE_SIZE+i];

	unsigned char **matrixKey = createMatrix(keyPrevious);
	unsigned char **matrixResult = allocateMatrix(4,4);

	rcon_i--;
	matrixResult[0][0] = matrixKey[0][0];
	matrixResult[0][0] ^= S_BOX[matrixKey[1][3]];
	matrixResult[0][0] ^= RCON[rcon_i];

	matrixResult[1][0] = matrixKey[1][0];
	matrixResult[1][0] ^= S_BOX[matrixKey[2][3]];

	matrixResult[2][0] = matrixKey[2][0];
	matrixResult[2][0] ^= S_BOX[matrixKey[3][3]];

	matrixResult[3][0] = matrixKey[3][0];
	matrixResult[3][0] ^= S_BOX[matrixKey[0][3]];

	for (i = 1; i < 4; i++)
		for (j = 0; j < 4; j++)
```

```
                {
                        matrixResult[j][i] = matrixResult[j][i - 1];
                        matrixResult[j][i] ^= matrixKey[j][i];
                }
        for(i=0; i<KEY_BYTE_SIZE; i++)
                keys[(rcon_i+1)*KEY_BYTE_SIZE+i]=matrixResult[i%4][i/4];
}
```

A8. Помоћна функција *GMul*

Табела 12: Програмски код језгарне функције GMul у паралелним имплементацијама

```
__device__ unsigned char cudaGMul(unsigned char a, unsigned char b)
{
        unsigned char p = 0;
        unsigned char counter;
        unsigned char carry;
        for (counter = 0; counter < 8; counter++)
        {
                p ^= a & (((b & 1) ^ 1) - 1);
                carry = (unsigned char)(a >> 7);
                a <<= 1;
                a ^= 0x1B & ((carry ^ 1) - 1);
                b >>= 1;
        }
        return p;
}
```

A9. Функција за генерисање тест инстанци

Табела 13: Програмски код C функције generateData за генерисање тест инстанци

```
void generateData() {
        int startdim = 16000;
        int inc = 4;
        int num = 13;
        int i, j, k;
        int n = 16;
        FILE *f;
        FILE *exef;
        char name[100];
        unsigned char origState[] =
```

```
{0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0x8d, 0x31, 0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34};
	unsigned char state[16];
	unsigned int stateLength;

	exef = fopen("pokreni.cmd", "w");
	if (exef == NULL)
	{
	printf("Greska.\n");
	exit(EXIT_FAILURE);
	}
	for (i = 0; i<num; i++)
	{
	fprintf(exef, "aes.exe -e p%d.txt p%d.e.txt\naes.exe -d p%d.e.txt p%d.e.d.txt\n\n", n, n);
	n *= inc;
	}
	fclose(exef);

	stateLength = sizeof(origState) / sizeof(origState[0]);

	n = 16;
	for (i = 0; i<num; i++) {
		sprintf(name, "p%d.txt", n);
		printf("Pravim %s\n", name);
		f = fopen(name, "w");
		if (f == NULL) {
			printf("Greska.\n");
			exit(EXIT_FAILURE);
		}
		memcpy(state, origState, sizeof(origState));
		fprintf(f, "%d\n", n);
		for (j = 0; j<n / 16; j++) {
			for (k = 0; k < stateLength; k++) {
				fprintf(f, "0x%x ", state[k]);
			}
			fprintf(f, "\n");
			encryptData(state, stateLength/KEY_BYTE_SIZE);
		}
		fclose(f);
		n *= inc;
	}
}
```

Додатак Б. Спецификација тест рачунара

Табела 14: Подаци о рачунару коришћеном за тестирање

Тип компоненте	Компонента
Процесор	*Intel Core i7-4790K*, такта од 4.00 GHz са 4 језгра
РАМ меморија	16GB
Графичка картица	*Gigabyte GeForce GTX 970 WINDFORCE,* такта од 1178 MHz, 4024 MB меморије и 1664 *CUDA* језгара

Додатак В. Улазне вредности

Табела 15: Вредности улазних хексадекадних низова

Појам	Вредност
Кључ *K*	{2b}, {7e}, {15}, {16}, {28}, {ae}, {d2}, {a6}, {ab}, {f7}, {15}, {88}, {09}, {cf}, {4f}, {3c}
Почетни блок *X*	{32}, {43}, {f6}, {a8}, {88}, {5a}, {30}, {8d}, {31}, {31}, {98}, {a2}, {e0}, {37}, {07}, {34}
Иницијализациони вектор *IV*	{01}, {01}, {01}, {01}, {01}, {03}, {01}, {45}, {01}, {01},{05}, {01}, {01}, {02}, {01}, {42}

Додатак Г. Библиотеке неопходне за извршавање паралелних имплементација

Табела 16: Списак библиотека неопдходних за извршавање паралелних имплементација

Име	Опис
cudart32_40_17.dll	Библиотека за извршење инструкција на графичкој картици
Microsoft Visual C++ 2010 Redistributable Package	Скуп библиотека програмерског окружења *Microsoft Visual C++ 2010*