Универзитет у Београду

Електротехнички факултет

Решавање проблема поновљивости израчунавања са бројевима у покретном зарезу на централном процесору

Дипломски рад

Ментор:

доц. др Марко Мишић

Кандидат:

Јован Николов 2016/0040

Београд, септембар 2020.

САДРЖАЈ

1. Увод

Нумеричка израчунавања каква се користе у научним истраживањима ослањају се на аритметику бројева у покретном зарезу. Ради постизања преносивости програмског кода, најчешће коришћени стандард за представљање бројева у покретном зарезу јесте IEEE-754 стандард [1]. Овај стандард уводи униформну семантику операција за широк спектар имплементација ове аритметике.

Стандард IEEE-754 дефинише исправно понашање за све операције, као и неопходне начине заокруживања. Мантиса ограничене прецизности и ограничени опсег експонената, које овај стандард дефинише, захтевају заокруживање међурезултата код већих аритметичких калкулација, као на пример код великих сума. Управо ова ограничена прецизност чини да операције са овим бројевима не буду асоцијативне.

Растуће перформансе савремених рачунара омогућавају решавање све комплекснијих проблема који захтевају све већи број операција са бројевима у покретном зарезу. Савремене вишејезгарне (*multi-core*), многојезгарне (*many-core*) и хетерогене (*heterogeneous*) архитектуре захтевају паралелизацију програма за потпуно искоришћење доступних ресурса. Због неасоцијативности ових операција, паралелна израчунавања које ови проблеми извршавају, а посебно редукције, могу постати недетерминистичка, а стога и непоновљива [2].

Иако постоје разне технике за превазилажење или смањивање ефеката овог проблема, оне углавном захтевају неке измене програма или повећавају број основних операција те знатно утичу на перформансе програма. Стога се ограничена прецизност ових бројева и неасоцијативност стандардом дефинисаних операција углавном прихвата као разуман компромис за постизање добрих перформанси програма.

Услед непоновљивости резултата преносивост ових програма је нарушена. Резултат зависи од параметра система на ком се калкулација извршава. Зато је готово немогуће паралелизовати операције са бројевима у покретном зарезу, а да се притом не наруши семантика коју дефинише IEEE стандард. Осим тога, непоновљивост резултата додатно отежава тестирање паралелних програма, јер се проблематичне ситуације (нпр. погрешни резултати и слично) не могу лако репродуковати.

Примарни циљ рада је анализа проблема поновљивости и имплементација једног решења овог проблема. У раду су изложени основни концепти неопходни за разумевање проблема поновљивости и постојећих решења. За имплементацију је одабрана хардверска техника која услед велике цене није прихваћена у савременим архитектурама, али се може једноставно софтверски имплементирати и представља добар основ за даље истраживање.

У питању је техника која користи широки акумулатор (*long accumulator*) за смештање међурезултата израчунавања у облику фиксног зареза, те на тај начин обезбеђује асоцијативност која представља основу за постизање поновљивости. Приказана је имплементација у облику C++ класе која се може једноставно користити у постојећим програмима. Демонстрирана је употреба ове класе у неколико репрезентативних примера, у секвенцијалном и паралелном окружењу због анализе поновљивости и перформанси.

У другом поглављу дат је историјат IEEE-754 стандарда, детаљно описан начин представљања бројева у покретном зарезу као и операције од интереса. У трећем поглављу дата је теоријска основа проблема и практичан пример. Четврто поглавље дефинише постојећа решења, мотивацију за избор имплементираног решења и основну идеју.

У петом поглављу дат је осврт на паралелизацију на системима са дељеном меморијом који су коришћени за тестирање и опис коришћених библиотека (POSIX Threads и OpenMP). Шесто поглавље садржи детаље имплементације класе широког акумулатора и референтног примера који је коришћен за тестирање.

Седмо поглавље се бави методологијом анализе, развојном и хардверском платформом, као и описом тест окружења. У осмом поглављу је дата анализа добијених резултата са аспекта поновљивости и са аспекта перформанси програма. Девето поглавље садржи закључак рада, предлоге за унапређење изложене имплементације и могућности за даље истраживање.

2. АРИТМЕТИКА У ПОКРЕТНОМ ЗАРЕЗУ

У овом поглављу биће изложен историјат IEEE-754 стандарда, начин представљања бројева у покретном зарезу, начини заокруживања и пример операције од интереса.

2.1. Историјат

Развојем рачунара настала је потреба за извршавање нумеричких калкулација које на неки начин користе реалне бројеве. Како је количина меморије у рачунарским системима ограничена, није могуће приказати сваки реални број са максималном прецизношћу. Разлог је чињеница да реалних бројева има бесконачно много.

Стога је било нужно на неки начин представити скуп реалних бројева са смањеном прецизношћу, али тако да је резултат извршених израчунавања и даље употребив. Како би било могуће пресликати што већи део скупа реалних бројева, пројектанти процесора и копроцесора су се одлучили да бројеве представе у покретном зарезу.

Основа репрезентације реалних бројева у покретном зарезу јесте научна нотација за представљање реалних бројева. Разлика између научне нотације и репрезентације броја у покретном зарезу јесте ограничена прецизност мантисе и ограничени опсег експонента. Управо ова ограничења омогућавају једноставну репрезентацију реалних бројева у рачунару.

Настале су разне имплементације аритметике са бројевима у покретном зарезу и то је онемогућило писање преносивог кода, односно програма који се могу извршавати на различитим архитектурама. Постало је нужно дефинисати стандард који ће бити усвојен од стране свих произвођача микропроцесора како би се проблем преносивости кода, али и поузданости израчунавања, решио.

Прва верзија овог стандарда [3] настала је 1985. године и дефинисала је репрезентацију бројева у покретном зарезу у бинарном облику. Дефинисане су репрезентације са четири нивоа прецизности од којих су најчешће коришћене: једнострука (*single*) 32-битна репрезентација и двострука (*double*) 64-битна репрезентација.

Такође, дефинисане су и репрезентације позитивне и негативне бесконачности, „негативна нула“ као и пет изузетака за обраду грешки као што су дељење нулом. Посебне вредности назване NaN (*Not a Number*) се користе представљање тих изузетака. Осим тога,

дефинисани су и субнормални (*subnormal*) бројеви који се користе за представљање бројева блиских нули, као и четири начина заокруживања.

Друга верзија овог стандарда [4] настала је 2008. године и представља проширење стандарда тако да обухвата и репрезентацију бројева у покретном зарезу у децималном облику. Ова верзија стандарда заменила је претходни, али и стандард IEEE-854 [5] дефинисан 1987. године који је дефинисао репрезентације бројева у покретном зарезу у основама различитим од 2 (бинарне).

Бинарни формати из оригиналног стандарда укључени су и у новом стандарду заједно са три нова основна формата: један бинарни и два децимална. Овај стандард уводи нове називе за бинарне репрезентације, па се претходно названа једнострука репрезентација сада зове *binary32*, а двострука *binary64*.

Најновија верзија овог стандарда [1] настала је 2019. године и представља мању ревизију стандарда из 2008. године. Овим стандардом су исправљени одређени недостаци претходно дефинисаних операција и дефинисане су нове операције које су препоручене за употребу, а потенцијално могу постати обавезне у новијим верзијама стандарда.

2.2. Формати бројева у покретном зарезу

IEEE-754 стандард дефинише неколико формата бројева у покретном зарезу који се користе за представљање коначног подскупа скупа реалних бројева. Формати су дефинисани својом основом (*radix*), прецизношћу и опсегом експонената и сваки формат представља јединствени скуп бројева у покретном зарезу. Постоји пет основних формата:

- три бинарна формата са кодовима ширине 32, 64 и 128 бита;
- два децимална формата са кодовима ширине 64 и 128 бита.

Осим ових формата дефинисани су и проширени формати који омогућавају представљање бројева веће прецизности. Скуп бројева у покретном зарезу који се могу представити одређеним форматом дефинисан је следећим целобројним параметрима:

- b = основа (*radix*), 2 или 10;
- p = ширина мантисе (прецизност);
- *emax* = максимални експонент e;
- *emin* = минимални експонент e (*emin* увек узима вредност 1 – *emax* за сваки формат).

Свим форматима се могу представити следећи бројеви у покретном зарезу:

- означена нула и сви ненулти бројеви облика $(-1)^s \times b^e \times m$, где је:
  - s или 0 или 1;
  - e било који цео број $emin \leq e \leq emax$;
  - m број представљен низом цифара облика $d_0 \cdot d_1 d_2 \ldots d_{p-1}$ где је d_i цифра $0 \leq d_i < b$ (стога је и $0 \leq m < b$);
- две бесконачности, $+\infty$ и $-\infty$;
- две NaN вредности, qNaN (*quiet*) и sNaN (*signaling*).

Понекад је корисно представити мантису у целобројном облику уместо у научној нотацији, са тачком одмах након прве цифре. У том случају је представа коначних бројева у покретном зарезу описана на следећи начин:

- означена нула и сви ненулти бројеви облика $(-1)^s \times b^q \times c$, где је:
  - s или 0 или 1;
  - q било који цео број $emin \leq q + p - 1 \leq emax$;
  - c број представљен низом цифара облика $d_0 d_1 d_2 \ldots d_{p-1}$ где је d_i цифра $0 \leq d_i < b$ (стога је c цео број за који важи $0 \leq c < b^p$).

Представљање мантисе у целобројном облику c са одговарајућим експонентом q описује идентичан скуп бројева у покретном зарезу као и представа у научној нотацији. За коначне бројеве у покретном зарезу важе једнакости: $e = q + p - 1$ и $m = c \times b^{1-p}$.

Најмањи позитиван нормалан број дефинисан форматом је b^{emin}, а највећи $b^{emax} \times (b - b^{1-p})$. Ненулти бројеви са апсолутном вредношћу мањом од b^{emin} називају се субнормални бројеви. Како би постојала подршка за ове бројеве у предложеним форматима, они увек имају мање од p значајних цифара. Сваки коначан број у покретном зарезу је целобројни умножак најмање позитивне субнормалне вредности $b^{emin} \times b^{1-p}$.

Знак s може носити додатне информације за број који има вредност 0. Иако сви формати имају посебне репрезентације позитивне $(+0)$ и негативне нуле (-0), знак нуле је некад значајан као на пример приликом дељења нулом, али не увек.

Табела 2.2.1. Основни формат бројева у покретном зарезу и параметри[1]

	Бинарни формат (b = 2)			**Децимални формат** (b = 10)	
параметар	**binary32**	**binary64**	**binary128**	**decimal64**	**decimal128**
p, број цифара	24	53	113	16	34
emax	+127	+1023	+16383	+384	+6144

У табели 2.2.1 представљени су основни формати бројева у покретном зарезу дефинисани овим стандардом као и параметри који их дефинишу. Параметар *emin* се рачуна коришћењем формуле: $emin = 1 - emax$.

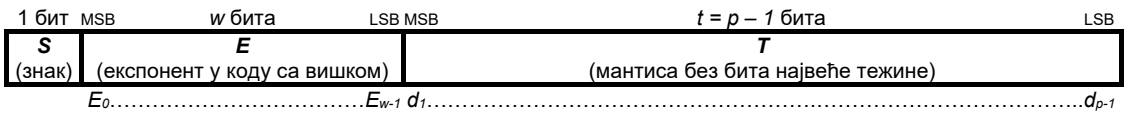
2.3. Кодирање бинарног формата бројева у покретном зарезу

Сваки број у покретном зарезу има тачно један начин кодирања у бинарном формату. Да би кодирање било јединствено у погледу параметара описаних у потпоглављу 2.2, вредност мантисе m се максимизује смањивањем експонента e све док e не постане *emin* или m узме вредност већу или једнаку 1 ($m \geq 1$).

Након што је овај процес завршен, уколико је $e = emin$ или $0 < m < 1$ онда је број у покретном зарезу субнормалан. Субнормални бројеви (и нула) се кодирају помоћу резервисане вредности експонента у коду са вишком (*biased exponent*) – 0.

Јединствени бинарни кодови бројева у покретном зарезу су ширине *k* бита и садрже три поља у редоследу приказаном на слици 2.3.1:

- 1 бит знака *S*;
- *w* бита експонент у коду са вишком $E = e + bias$;
- $t = p - 1$ бита најмање тежине мантисе, тј. низ цифара $T = d_1 d_2 \ldots d_{p-1}$ где је водећи бит мантисе d_0 имплицитно кодиран експонентом у коду са вишком *E*.



Слика 2.3.1. Начин кодирања бинарног формата бројева у покретном зарезу [1]

[1] Подршка за *binary32* и *binary64* формате постоји у свим новијим програмским језицима и они се често означавају са *float* и *double*, респективно. С друге стране, *binary128* формат није увек подржан, а у језицима C и C++ се означава са *long double*.

Опсег експонента E у коду са вишком обухвата:

- све целобројне вредности између 1 и $2^w - 2$ за кодирање нормалних бројева;
- резервисану вредност 0 за кодирање ± 0 и субнормалних бројева;
- резервисану вредност $2^w - 1$ за кодирање $\pm\infty$ и NaN вредности.

Репрезентација r и вредност v кодираног броја у покретном зарезу се на основу поља одређују на начин приказан у табели 2.3.1.

Табела 2.3.1. Начин одређивања репрезентације и вредности кодираног броја у покретном зарезу

E	T	r	v
$2^w - 1$	$T \neq 0$	$(-1)^S \times (+\infty)$	$(-1)^S \times (+\infty)$
$2^w - 1$	$T = 0$	qNaN или sNaN	NaN независно од знака S
$1 \leq E \leq 2^w - 2$	небитно	$(S, (E - bias), (1 + 2^{1-p} \times T))$	$(-1)^S \times 2^{E-bias} \times (1 + 2^{1-p} \times T)$[2]
0	$T \neq 0$	$(S, emin, (0 + 2^{1-p} \times T))$	$(-1)^S \times 2^{emin} \times (0 + 2^{1-p} \times T)$[3]
0	$T = 0$	$(S, emin, 0)$	$(-1)^S \times (+0)$[4]

У табели 2.3.2 дати су параметри за кодирање бројева у покретном зарезу за различите бинарне формате. За овај рад је од интереса формат *binary32* који користи 1 бит за знак, 8 бита за експонент у коду са вишком и 23 бита за кодирање мантисе без бита највеће тежине.

Табела 2.3.2. Параметри кодирања бројева у покретном зарезу за бинарне формате [1]

	Бинарни формат (b = 2)				
Параметар[5]	**binary16**	**binary32**	**binary64**	**binary128**	**binary{*k*}** ($k \geq 128$)
k, ширина формата	16	32	64	128	умножак 32
p, прецизност	11	24	53	113	$k - round(4 \times \log_2(k)) + 13$
emax, највећи експонент e	15	127	1023	16383	$2^{k-p-1} - 1$
Параметри кодирања					
bias, $E - e$	15	127	1023	16383	*emax*
бит знака	1	1	1	1	1
w, ширина поља експонента	5	8	11	15	$round(4 \times \log_2(k)) - 13$
t, ширина поља мантисе без бита највеће тежине	10	23	52	112	$k - w - 1$
k, ширина кода	16	32	64	128	$1 + w + t$

[2] Нормални бројеви имају бит 1 имплицитно као бит највеће тежине (вредност мантисе је $1.T$).
[3] Субнормални бројеви имају бит 0 имплицитно као бит највеће тежине (вредност мантисе је $0.T$).
[4] Означена нула.
[5] Вредности параметара су дате у битима.

2.4. Начини заокруживања бројева у покретном зарезу

Алгоритам заокруживања узима број који се сматра бесконачно прецизним и по потреби га модификује да би се уклопио у формат одредишта. У случају грешке сигнализира изузетак нетачности, недостатак (*underflow*) или прекорачење (*overflow*). Осим ако је другачије наведено, свака операција се извршава као да је прво произвела међурезултат бесконачне прецизности, а затим се извршава заокруживање међурезултата на основу примењеног атрибута заокруживања.

IEEE-754 стандард дефинише концепт атрибута за модификацију нумеричке семантике и семантике изузетака. Имплементације стандарда морају да обезбеде подршку за атрибуте тако да је кориснику омогућено да језиком дефинисаним елементима, као на пример компајлерским директивама, селектује атрибуте који се примењују у одређеном делу кода. Атрибути који су од интереса за овај рад јесу атрибути начина заокруживања.

Атрибут начина заокруживања утиче на све рачунске операције које могу бити непрецизне. Непрецизни нумерички резултати у покретном зарезу увек имају исти знак као незаокружени резултат. Постоје две групе атрибута начина заокруживања:

1) атрибути заокруживања на најближу вредност:

   Бесконачно прецизни резултат апсолутне вредности најмање $b^{emax} \times (b - \frac{1}{2}\, b^{1-p})$ увек се заокружује на ∞ без промене знака. Уколико је његова апсолутна вредност мања од поменуте, резултат заокруживања је број у покретном зарезу који му је најближи. Уколико су два најближа броја у покретном зарезу која окружују бесконачно прецизан резултат подједнако удаљена, разликујемо два атрибута:

   a) *roundTiesToEven* – узеће се број са **парном цифром најмање тежине**; уколико то није могуће, узеће се број **веће апсолутне вредности**[6];

   b) *roundTiesToAway* – узеће се број **веће апсолутне вредности**.

2) усмерени атрибути заокруживања:

   a) *roundTowardPositive* – резултат заокруживања је број који се може представити селектованим форматом (укључујући +∞) који **није мањи** од бесконачно прецизног резултата;

[6] Измена у стандарду из 2019. године. За више информација погледати одељак 4.3.1 измењеног стандарда [1].

b) *roundTowardNegative* – резултат заокруживања је број који се може представити селектованим форматом (укључујући $+\infty$) који **није већи** од бесконачно прецизног резултата;

c) *roundTowardZero* – резултат заокруживања је број који се може представити селектованим форматом (укључујући $+\infty$) који **нема апсолутну вредност већу** од апсолутне вредности бесконачно прецизног резултата.

2.4.1. Начин заокруживања примењен у раду

За потребе овог рада користи се бинарни формат *binary32* и *roundTiesToEven* атрибут заокруживања који се на једноставнији начин може објаснити следећим правилима:

1) $|R| \geq 2^{emax} \times (2 - 2^{-p}) \quad \rightarrow \pm\infty$
2) $x.xx|0yyy \ldots y \quad \rightarrow x.xx$
3) $x.xx|1yyy \ldots y \ (\exists y \neq 0) \quad \rightarrow x.xx + 0.01$
4) $x.xx|1000 \ldots 0$ разликујемо два случаја у зависности од бита x непосредно пре црте:
   a) $x.x0|1000 \ldots 0 \quad \rightarrow x.x0$
   b) $x.x1|1000 \ldots 0 \quad \rightarrow x.x1 + 0.01$

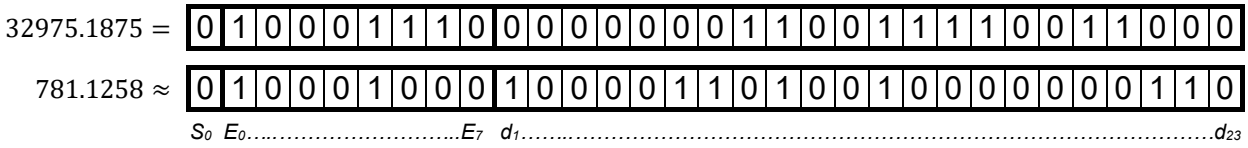
2.4.2. Пример израчунавања збира два броја са заокруживањем

Нека је потребно израчунати збир бројева 32975.1875 и 781.1258. Репрезентација броја 32975.1875 је потпуно тачна (грешка приликом смештања у *binary32* формат је 0). Са друге стране, репрезентација броја 781.1258 није тачна, јер мантиса *binary32* од 24 бита није довољно широка за представљање овог броја већ је заправо кодирани број 781.12579345703125.

Алгоритам израчунавања збира два броја у покретном зарезу је следећи:

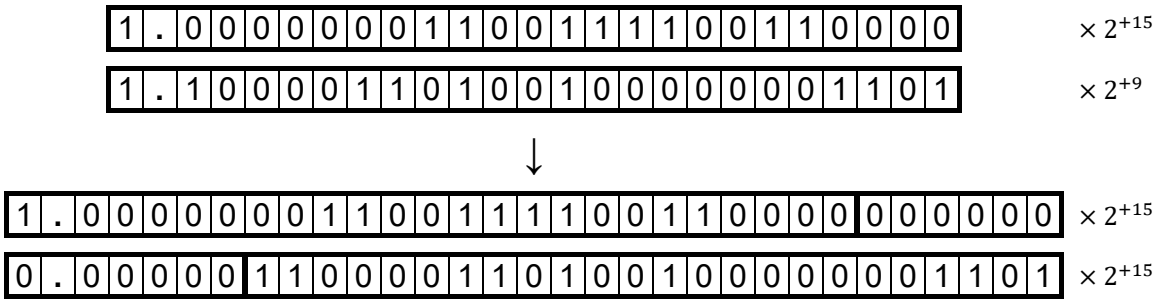
1) У општем случају, потребно је број са мањим експонентом довести на ред величине броја са већим експонентом.
2) Сада се врши сабирање бројева по правилима сабирања бројева у основи 2.
3) Након сабирања новодобијену мантису M_{A+B} је потребно нормализовати свођењем на облик описан у одељку 2.2 ($d_0 \cdot d_1 d_2 \ldots d_{23} \ldots$).
4) Уколико је мантиса веће ширине од 24 бита (постоје бити $d_{24}, d_{25}, \ldots$), потребно је резултат заокружити по правилу описаном у одељку 2.4.1.

На слици 2.4.2.1 дата је репрезентација ових бројева у *binary32* формату. Имплицитни бит највеће тежине мантиса оба броја је 1. Експонент броја 32975.1875 представљеног у *binary32* формату је +15, док је експонент броја 781.1258 у истом формату +9. То значи да постоји потреба да се број 781.1258 доведе на исти ред величине као број 32975.1875.



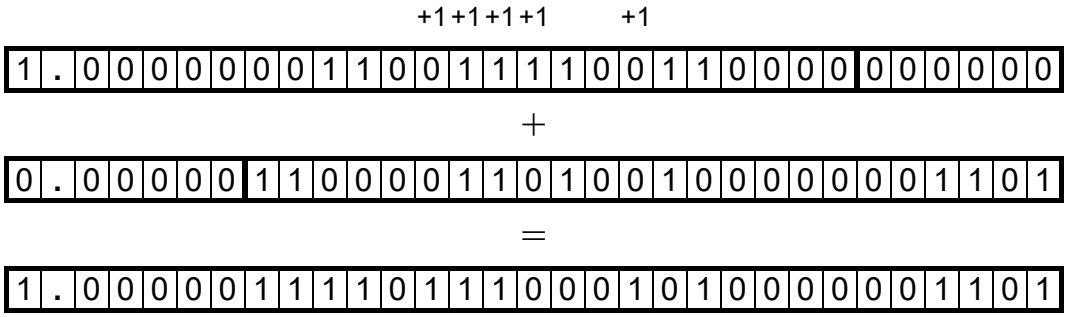
Слика 2.4.2.1. Репрезентација датих бројева у *binary32* формату

Поравнавање мантиса ова два броја је приказано на слици 2.4.2.2. Треба назначити да је овде извршено релативно померање мантисе другог броја у односу на мантису првог броја тако да експоненти више нису од значаја за извршавање операције сабирања.



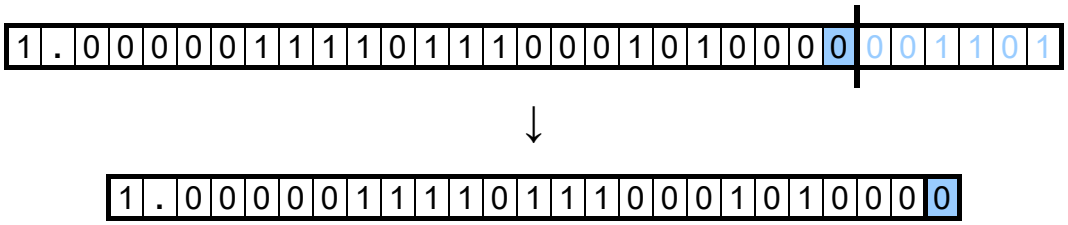
Слика 2.4.2.2. Поравнавање мантиса датих бројева (1. корак)

Сада је потребно извршити сумирање две поравнате мантисе по стандардним правилима за сумирање два бинарна броја. Ово је представљено на слици 2.4.2.3.



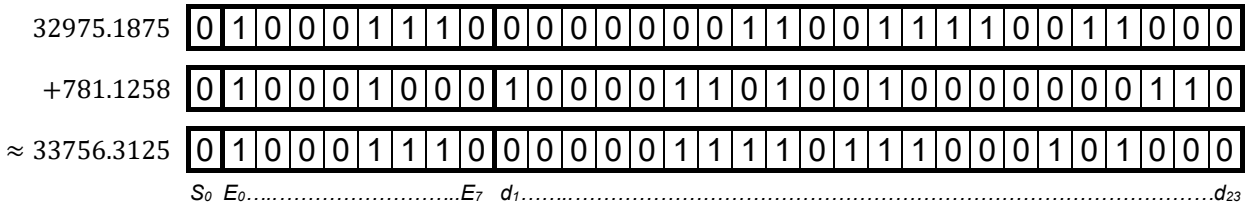
Слика 2.4.2.3. Сабирање поравнаних мантиса (2. (и 3.) корак)

Као резултат сабирања две поравнане мантисе добија се мантиса ширине 30 бита што је веће него 24 бита које могу да се репрезентују форматом *binary32*. Треба прво приметити да је збир ове две мантисе већ нормализован, па је заправо и 3. корак већ извршен. Сада остаје да се резултат заокружи тако да може да стане у 24 бита који се могу кодирати у датом формату.



Слика 2.4.2.4. Заокруживање резултата (4. корак)

Заокруживање је извршено по другом правилу из одељка 2.4.1 и приказано је на слици 2.4.2.4. Како је бит најмање тежине који се може сместити у 24-битну мантису 0 (осенчен на слици) резултат не зависи од преосталих бита са десне стране. Коначно, резултат представљен у *binary32* формату приказан је на слици 2.4.2.5.



Слика 2.4.2.5. Репрезентација коначног резултата датог израза

Тачан и очекивани резултат датог израза је 33756.3133. Међутим, рачунајући овај израз коришћењем *binary32* формата за представљање датих бројева, добили смо резултат 33756.3125. Стога смо добили апсолутну нумеричку грешку приликом израчунавања овог израза која износи -0.0008.

Пропагацијом оваквих грешака укупна апсолутна грешка расте, па је зато нужно одабрати одговарајућу прецизност аргумената да бисмо добили одговарајућу прецизност резултата. У овом случају, да смо користили *binary64* формат, добили бисмо тачан резултат.

3. О ПРОБЛЕМУ ПОНОВЉИВОСТИ ИЗРАЧУНАВАЊА

У овом поглављу биће детаљније објашњен проблем поновљивости резултата код израчунавања које користе бројеве у покретном зарезу. Осим тога, биће објашњени начини за постизање поновљивости зарад разумевања имплементације одабране технике и детаљније ће бити образложени циљеви рада.

3.1. Мотивација за поновљивост израчунавања

Поновљивост резултата израчунавања, које користе бројеве у покретном зарезу, од интереса је из неколико разлога. Први разлог је чињеница да је тестирање програма тешко уколико извршавања која доводе до грешке нису поновљива. С друге стране, поновљивост је некада неопходна због поређења резултата израчунавања где уговорене стране морају да се сложе око добијених резултата (нпр. симулације света у играма са више играча) [6].

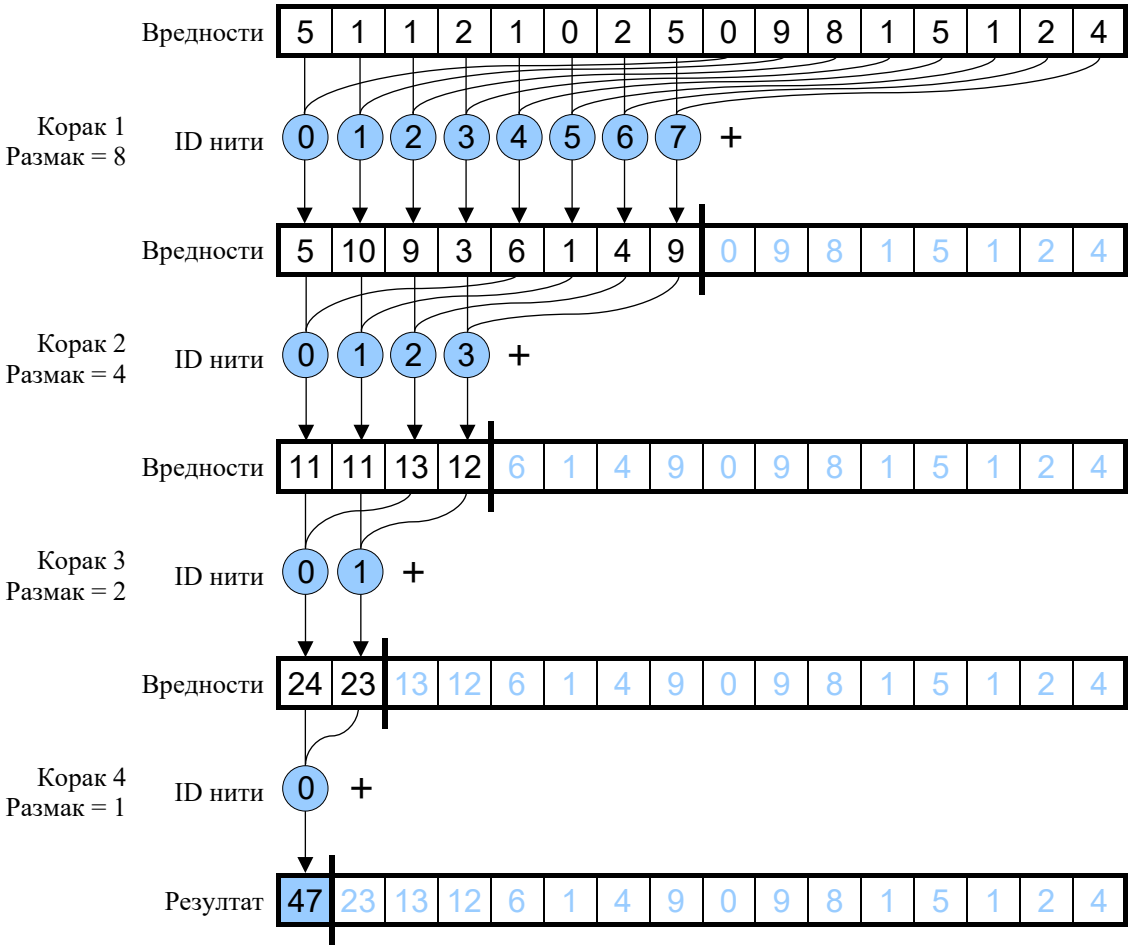
Већина данашњих научних истраживања захтева велике и комплексне нумеричке симулације за постизање нових открића. Медицина, биомедицина и остале бионауке, климатологија, дизајн и израда напредних материјала, геологија, астрономија, хемија, физика и финансијски системи, само су неке од научних грана које користе овакве симулације [7].

Постоје студије које проучавају проблем поновљивости код неких симулација као што су симулације климатских промена [8], N-body симулације, динамика атома и молекула [9], [10] и динамика флуида [11]. Све ове симулације имају заједничку особину – нестабилност, односно за њих важи да мале промене међурезултата доводе до значајних промена у каснијим деловима симулације. Стога су њихови резултати непоновљиви и често нетачни.

Готово увек се овакве симулације извршавају на паралелним рачунарима због великог потенцијала за паралелизацију различитих елемента симулације. Паралелно процесирање, које дистрибуира ова израчунавања на неколико процесних елемената (нити, процесора), често је коришћено за постизање прихватљивих перформанси симулација. Међутим, ово за последицу уводи додатне проблеме по питању поновљивости резултата.

Паралелизација производи недетерминистичке догађаје који утичу на поновљивост резултата. Редослед комуникације, број процесних елемената и локација података у меморији варира, те варирају и парцијалне суме. Паралелни програми најчешће имплементирају операцију редукције као последњи корак свог извршавања.

Редукција се састоји од примене одређене операције $(+,\times,max,\ldots)$ на све парцијалне резултате које производе процесни елементи да би се израчунао коначни резултат. Стога она омогућава поделу посла и тако боље искоришћење доступних ресурса. Међутим, у зависности од имплементације, редукција може изазвати недетерминистизам при пропагацији грешки што директно утиче на поновљивост резултата [12].



Слика 3.1.1. Редукциона шема за израчунавање суме елемената низа

На слици 3.1.1 дат је пример редукционе шеме која се користи за сумирање елемената низа. У овом случају редослед извршавања операција је детерминистички, али то за последицу има слабе перформансе зато што се у сваком наредном кораку користи дупло мање нити.

Чак су и секвенцијални делови програма, који су у сагласности са IEEE-754 стандардом, нумерички веома осетљиви на многе особине окружења. Неке од тих особина су: особине аритметичке јединице ниског нивоа (регистри промењљиве прецизности, спојени оператори), компајлерске оптимизације, мане програмског језика или верзије библиотека. Све ово утиче на смањење нумеричке поновљивости и преносивости нумеричких израчунавања [13].

3.2. Узроци непоновљивости

У претходном поглављу дат је начин репрезентације бројева у покретном зарезу у рачунарским системима као и пример операције сабирања два оваква броја. Сама архитектура аритметике са овим бројевима уводи ограничења која доводе до проблема поновљивости.

Главни узрок непоновљивости израчунавања са бројевима у покретном зарезу јесу грешке при заокруживању које настају услед ограничене прецизности (ширине мантисе) бинарне представе ових бројева. Најгори случајеви ових грешака су апсорпција (*absorption*) и поништавање (*cancellation*). Друге грешке се могу десити у случајевима прекорачења (*overflow*) и заокруживања на нулу (*underflow*) [12].

3.2.1. Утицај прецизности на појаву грешки при заокруживању

Неки реални бројеви имају бесконачну представу у формату са покретним зарезом. Стога се не могу тачно представити мантисом ограничене прецизности. Један од таквих бројева је број 0.1 који има само две значајне цифре, али се не може представити ни у једном бинарном формату, јер увек захтева заокруживање.

С друге стране, број 0.499755859375 има 12 значајних цифара, а може се потпуно тачно представити форматом *binary32*. Из ова два примера може се закључити да број значајних цифара броја, представљеног у децималном систему, није директно повезан са прецизношћу бинарног формата који се користи за његову представу. Међутим, понекад је корисно размотрити однос ове две величине због лакшег одабира бинарног формата.

Посматрајмо сада целобројну аритметику како бисмо одредили неку функцију ове две величине. Највећи неозначени цео број који се може представити бинарним форматом ширине p бита је $2^p - 1$. Применом логаритма са основом 10 се може одредити број значајних цифара који неки цео број у децималном систему има.

Како мантиса у ствари представља цео број, логаритам је користан за поређење бинарних формата за представу бројева у покретном зарезу. Формула:

$$n = \log_{10}(2^p - 1) \tag{3.2.1}$$

даје процену броја значајних цифара који се могу представити бинарним форматом са прецизношћу (ширином мантисе) p.

У случају *binary32* формата ширина мантисе је $p = 24$ бита па је процена броја значајних цифара који се овим форматом могу представити ~7.225. *Binary64* формат има мантису ширине $p = 53$ бита, па је процена броја значајних цифара ~15.955. Уколико нам је потребан одређени број значајних цифара за неко израчунавање, можемо користити инверзну формулу формуле (3.2.1) за одређивање адекватног бинарног формата:

$$p = \log_2(10^n + 1). \tag{3.2.2}$$

Ако бисмо хтели да представимо бројеве са, на пример, 20 значајних цифара, било би нам потребно најмање 67 бита мантисе у бинарном формату. Лако се може приметити да ни *binary32* ни *binary64* формати не могу представити ове бројеве. У овом случају би било потребно користити формате проширене прецизности [1] или рецимо *binary128* формат.

3.2.2. Прекорачење и заокруживање на нулу

У претходном одељку смо видели да скуп реалних бројева који се могу тачно представити бинарним форматом у покретном зарезу није униформан. За све ограничене формате за представљање бројева важи да могу представити само оне бројеве у границама између минималног и максималног броја. Ако нека операција за резултат има вредност која је већа од максималне могуће вредности која се може представити коришћеним форматом, резултат ће бити специјална вредност – бесконачност.

Немогућност представљања резултата, који је већи од максималне могуће вредности, назива се прекорачење (*overflow*). Проблем настаје када се овај резултат даље користи у израчунавању, јер ће све остале операције које користе ову вредност вратити специјалну вредност: бесконачно или NaN (у случају множења нулом).

Аналогно, заокруживање на нулу (*underflow*) настаје када резултат има вредност мању од најмање могуће вредности која се може представити коришћеним форматом. Заправо, заокруживање на нулу је једна од могућих последица када се деси описана ситуација и зависи од тренутно изабраног начина заокруживања. Ако дође до оваквог заокруживања, може доћи и до каскадног поништавања извршавањем сукцесивних операција или враћање специјалне вредности: бесконачно (у случају дељења нулом).

3.2.3. Апсорпција и поништавање

Апсорпција настаје услед заокруживања резултата сабирања два броја у покретном зарезу који имају знатно различите експоненте. Нека су *a* и *b* два броја у покретном зарезу

таква да је a много веће од b. Део b ће бити изгубљен у збиру $(a + b)$ услед заокруживања на мантису ограничене ширине.

Ако су сви бити броја b изгубљени приликом заокруживања резултата збира ова два броја, каже се да је апсорпција катастрофална (*catastrophic absorption*). Катастрофална апсорпција је илустрована примером у одељку 3.2.4.

Поништавање настаје услед одузимања два веома блиска броја истог знака (или услед сабирања два броја истог типа различитог знака). Резултат ове операције се може веома разликовати од тачног резултата и може изазвати огромну релативну грешку. Катастрофално поништавање настаје када само неколико значајних цифара остану у резултату.

3.2.4. Неасоцијативност операција са бројевима у покретном зарезу

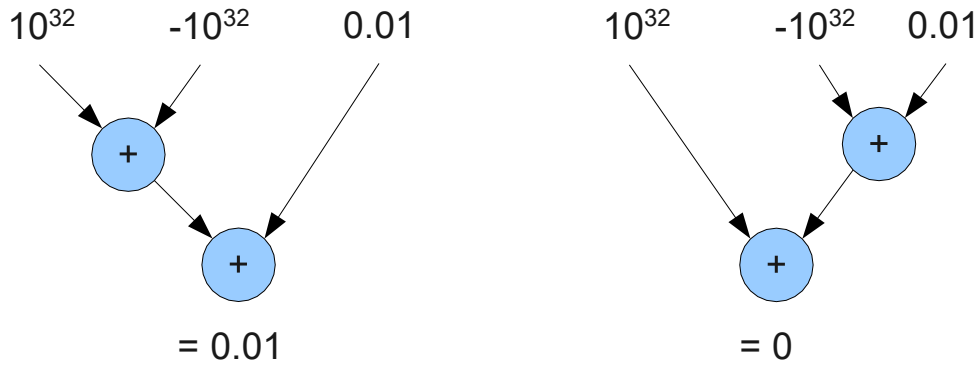
Операције са бројевима у покретном зарезу нису асоцијативне услед грешака при заокруживању које настају због ограничене прецизности и опсега представе бројева у покретном зарезу. Промене редоследа извршавања операција мењају утицај различитих нумеричких грешака на коначни резултат.

Неасоцијативност може произвести веома различите резултате за различите редоследе извршавања операција. Понекад, може чак изазвати прекорачење иако другачији редослед извршавања може добити тачан резултат. Математички идентични изрази:

$$(MAX - MAX) + MAX \leftrightarrow (MAX + MAX) - MAX$$

се различито евалуирају: израз са леве стране не производи прекорачење нити грешке при заокруживању, док до прекорачења јасно долази у изразу са десне стране. Константа *MAX* представља максималну вредност која се може представити у коришћеном формату.

Размотримо следећи пример збира три броја са веома различитим експонентима:



Слика 3.2.4.1. Пример неасоцијативности сабирања бројева у покретном зарезу [14]

Зарад лакшег разумевања овог примера биће представљено израчунавање ова два збира коришћењем алгоритама описаних у одељцима 2.4.1 и 2.4.2. Овде неће бити приказани сви кораци зарад концизности, а уколико постоји нејасноћа погледати пример у одељку 2.4.2.

На слици 3.2.4.2 је дата репрезентација ових бројева у *binary32* формату. На слици 3.2.4.3 је дат алгоритам израчунавања израза:

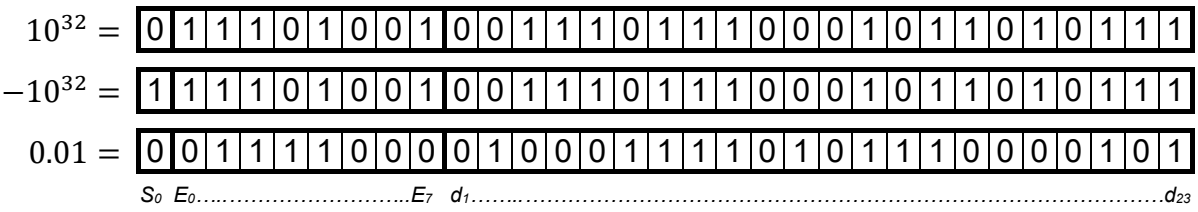
$$(10^{32} + (-10^{32})) + 0.01 \qquad (3.2.4.1)$$

илустрованог на слици 3.2.4.1 лево, док је на слици 3.2.4.4 дат алгоритам израчунавања израза:

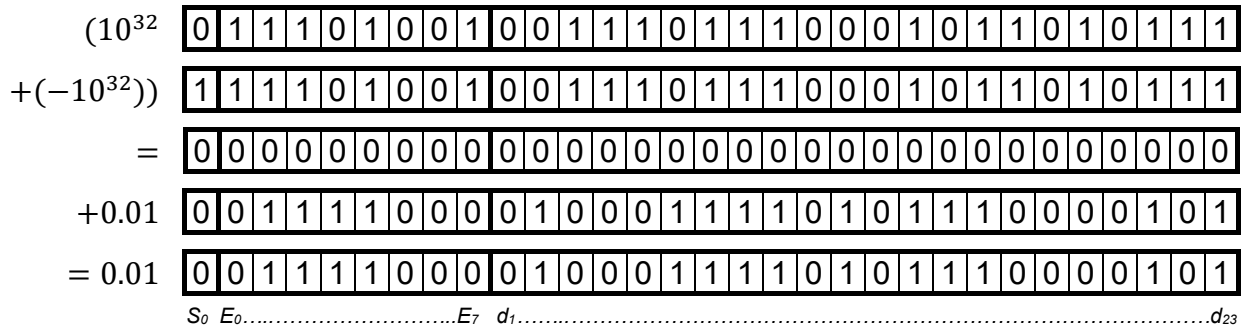
$$10^{32} + ((-10^{32}) + 0.01) \qquad (3.2.4.2)$$

илустрованог на слици 3.2.4.1 десно.

Може се приметити да приликом израчунавања израза (3.2.4.2) долази до катастрофалне апсорпције услед велике разлике у опсегу бројева (-10^{32}) и 0.01. Сви бити броја 0.01 се игноришу услед ограничене прецизности мантисе (24 бита).

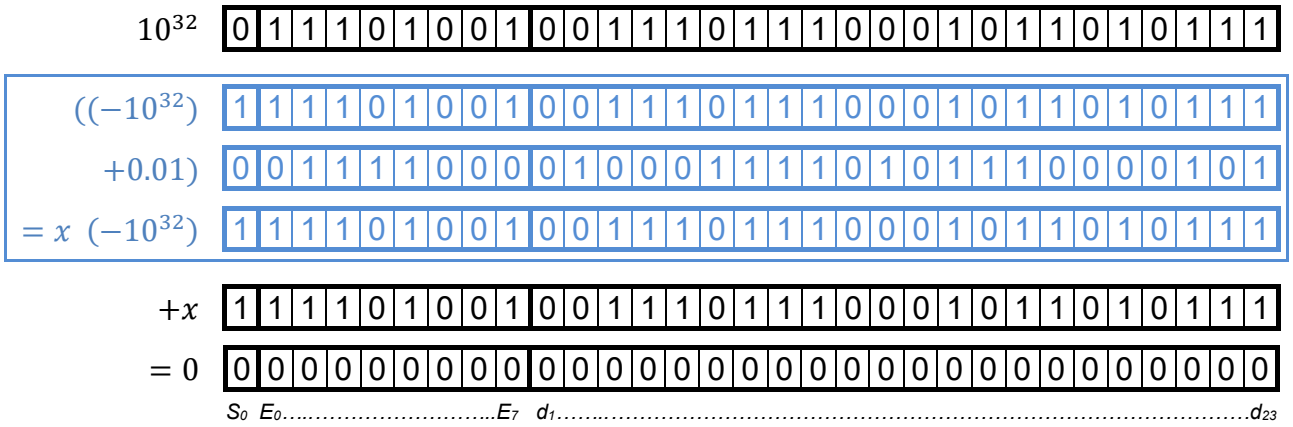


Слика 3.2.4.2. Репрезентација датих бројева у *binary32* формату



Слика 3.2.4.3. Поступак израчунавања израза (3.2.4.1)

За израчунавање израза (3.2.4.1) прво се рачуна збир $10^{32} + (-10^{32})$ чији је резултат 0 (вредности се пониште), а затим се на добијени међурезултат додаје вредност 0.01. Стога је коначни резултат 0.01.



Слика 3.2.4.4. Поступак израчунавања израза (3.2.4.2)

Израз (3.2.4.2) се израчунава тако што се прво израчуна збир: $(-10^{32}) + 0.01$ и добија међурезултат x. Вредност међурезултата x је (-10^{32}) зато што је мантиса недовољне ширине (24 бита) те вредност 0.01 не доприноси резултату[7]. Сада се рачуна збир: $10^{32} + x$ чији је резултат 0 што представља резултат израза (3.2.4.2).

Уколико бисмо користили један акумулатор за израчунавање вредности, израз се може добити и додавањем вредности: (-10^{32}), 0.01 и 10^{32}, тим редом. Заправо, добија се израз: $\left((-10^{32}) + 0.01\right) + 10^{32}$ који је еквивалентан датом изразу (3.2.4.2) зато што је операција сабирања бројева у покретном зарезу комутативна. Више детаља је могуће пронаћи у главним референцама које се односе на аритметику бројева у покретном зарезу [12], [13] и [15].

3.3. Врсте нумеричке поновљивости

У савременим имплементацијама аритметичких јединица за рад са бројевима у покретном зарезу постоје одређене технике којима се обезбеђује делимична асоцијативност за мали број аргумената. Међутим, ове хардверске технике нису довољне да се проблем поновљивости у потпуности реши зато што не гарантују асоцијативност за велики број аргумената, односно велики број поновљених операција.

Може се примети да чак да смо у претходном примеру користили *binary128* формат за представљање ових бројева дошло би до одређене грешке. Добијени резултати би се разликовали за ~0.005625, јер би се у првом случају поново добио 0.01, а у другом ~0.015625.

[7] Да је ширина мантисе у репрезентацији ових бројева била ~100 или више бита онда би се добила вредност нешто већа од (-10^{32}). Међурезултат се заокружује и стога долази до грешке при заокруживању.

Стога, у општем случају, просто повећање прецизности не гарантује асоцијативност па тако ни поновљивост.

Поставља се питање да ли је могуће програмски фиксирати редослед операција како би се заобишла неасоцијативност ових операција. Уколико је могуће контролисати редослед извршавања операција у програму, нпр. рачунање суме елемената низа – редукциони алгоритми, тада свако извршавање програма може генерисати исте резултате.

Серијализација операција на описани начин, међутим, онемогућава паралелизацију, што у погледу савремених рачунарских система није прихватљиво. Осим тога, оваква имплементација програма некад није могућа, јер је природа проблема који програм решава недетерминистичка. Тада је немогуће фиксирати одређени редослед извршавања операција.

У зависности од тога да ли примењена техника обезбеђује поновљивост која зависи од параметара система[8], разликујемо две врсте поновљивости [16]:

1) **слабу нумеричку поновљивост** (*weak numerical reproducibility*) – резултати два узастопна извршавања су идентични ако су **параметри система идентични**;
2) **јаку нумеричку поновљивост** (*strong numerical reproducibility*) – резултати два узастопна извршавања су идентични **независно од параметара система или архитектуре**. Технике које пружају јаку нумеричку поновљивост се деле на:
   a) технике које производе **исправно заокружене резултате** [2];
   b) технике које **не гарантују тачност** [6].

Слаба нумеричка поновљивост се може постићи форсирањем редоследа операција како би се заобишао проблем неасоцијативности операција. На овај начин се, за одређене параметре система, операције увек извршавају у истом редоследу, па је и резултат поновљив. Променом параметара система мењају се и редоследи операција, па се тако мењају и резултати.

Јака нумеричка поновљивост се може постићи избегавањем грешака при заокруживању међурезултата што ефективно обезбеђује и асоцијативност операција. Представљањем међурезултата у облику фиксног зареза, са ширином таквом да покрива читав скуп бројева у покретном зарезу које покрива и коришћени формат, заокруживање постаје непотребно.

[8] У контексту паралелног програма, параметри система су број паралелних нити које се извршавају и слично.

Овако описана техника **гарантује тачност** резултата зато што су нумеричке грешке елиминисане, а сама прецизност резултата је еквивалентна прецизности коришћеног формата. Ова техника се назива **широки акумулатор** (*long accumulator*). За *binary32* формат потребно је ~280 бита како би се добила потпуна тачност и избегле све грешке при заокруживању. За *binary64* формат потребно је ~2100 бита да би се постигао исти ефекат.

3.4. Циљеви рада

Раст броја језгара на централним процесорима, развијање акцелератора и других специјалних чипова за убрзање одређених операција погодује коришћењу паралелизације на савременим архитектурама. Рачунарство високих перформанси (*High Performance Computing – HPC*) за масивне нумеричке симулације ослања се на паралелизам за ефикасно искоришћење хардверских ресурса и добијање задовољавајућих перформанси. Паралелизација, стога, представља кључни апарат за убрзање захтевних израчунавања и побољшање ефикасности.

Нажалост, побољшана ефикасност и перформансе програма се добијају по цени поновљивости израчунавања. Паралелизација инхерентно уноси насумичност у редослед извршавања операција, а услед неасоцијативности операција са бројевима у покретном зарезу насумичност мења начин пропагације грешки при заокруживању. Услед недетерминистичке пропагације грешака при заокруживању резултати израчунавања нису поновљиви.

Примарни циљ овог рада је да читаоцу прикаже проблем нумеричке поновљивости, опише нека предложена решења и кроз анализу имплементације једног конкретног решења да увид у цену поновљивости. У раду је приказана софтверска имплементација добро познатог хардверског решења овог проблема: широког акумулатора. Ово решење гарантује поновљивост по цену меморијске и временске сложености софтверске имплементације.

Широки акумулатор је имплементиран у облику C++ класе која, као таква, може једноставно да се примени у постојећим програмима. Анализа имплементираног решења извршена је на референтном тест примеру који има могућност подешавања разних параметара ради боље анализе поновљивости и перформанси. Осим тога, имплементирано решење је примењено на пар карактеристичних програма ради демонстрирања поновљивости у реалним паралелним програмима.

4. ТЕХНИКЕ ЗА ПОСТИЗАЊЕ ПОНОВЉИВОСТИ

У овом поглављу биће представљена два приступа за постизање поновљивости. Биће дати примери техника који користе ова два приступа и образложена идеја технике чија је имплементација разматрана у раду.

4.1. Поновљивост услед побољшања тачности резултата

У аритметици бројева у покретном зарезу свака елементарна операција може да изазове грешке при закруживању које се пропагирају током израчунавања. Ове грешке имају кумулативно дејство и смањују тачност резултата. Довољним побољшањем тачности резултата укупна апсолутна грешка може постати занемарљива, па се добијени резултати могу сматрати поновљивим.

За побољшање тачности често се користе компензације, експанзије или библиотеке које имплементирају аритметику арбитрарне прецизности. Примена ових техника на адекватан начин може омогућити добијање поновљивих резултата. Базиране су на коришћењу *error-free* трансформација које омогућавају рачунање грешки при заокруживању које генеришу основне операције са бројевима у покретном зарезу.

4.1.1. Error-free transformations (EFT)

Основна идеја ових трансформација јесте да за елементарну операцију $op \in \{+, -, *\}$ примењену на два броја у покретном зарезу $\hat{a}$ и $\hat{b}$ постоје два броја у покретном зарезу $\hat{x}$ и $\hat{y}$ за које важи израз: $\hat{a}\ op\ \hat{b} = \hat{x} + \hat{y}$. У овом изразу $\hat{x}$ представља заокружени део резултата, а $\hat{y}$ генерисану грешку при заокруживању. У литератури се често ова два броја називају деловима резултата високог (*high-order*) и ниског реда (*low-order*), респективно.

Најчешће коришћене *error-free* трансформације су $[x, y] = 2Sum(a, b)$ за операције сабирања и одузимања и $[x, y] = 2Product(a, b)$ за операцију множења. Осим ових трансформација постоје и друге трансформације које имају специфичну примену као што су трансформације за спојену операцију множења и акумулирања (FMA). У овом раду даље разматрање ових трансформација није од значаја и више детаља постоји у главним изворима за овај део рада [12], [13].

4.1.2. Компензациони алгоритми

Компензациони алгоритми пружају k пута бољу тачност приликом израчунавања у датој прецизности u. Идеја ових алгоритама јесте акумулирање грешки сваке елементарне операције у току израчунавања у један терм грешке (*error term*). На крају израчунавања овај терм грешке се компензује на заокружени резултат да би се добио тачан (тачнији) резултат.

Ови алгоритми користе претходно поменуте *error-free* трансформације за рачунање грешке приликом израчунавања међурезултата и на тај начин постижу тачније резултате. Дефинисани су алгоритми за постизање два пута тачнијег резултата суме елемената низа a: $res = Sum2(a)$ и скаларног производа два низа a и b: $res = Dot2(a, b)$. Такође, дефинисани су и алгоритми за постизање k пута тачнијег резултата истих операција: $res = SumK(a, k)$ и $res = DotK(a, b, k)$.

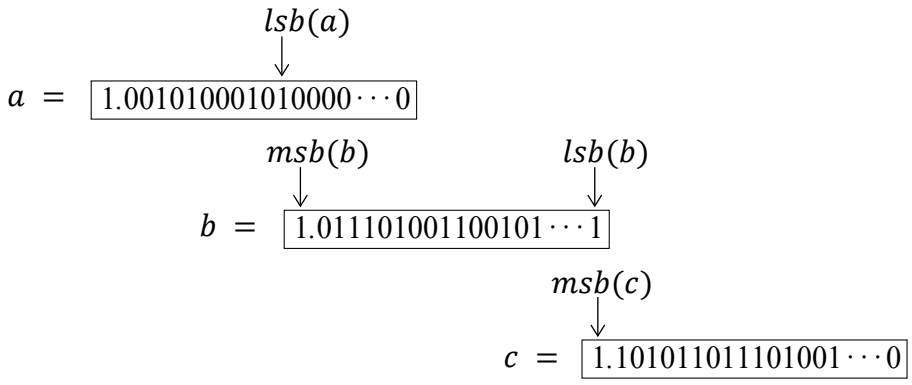
4.1.3. Експанзије бројева у покретном зарезу

Експанзије представљају реалне бројеве као збирове од n компонената које су представљене у покретном зарезу. На овај начин су омогућена израчунавања веће тачности, јер се неке грешке при заокруживању могу избећи тако што се вредност упише у компоненту која има одговарајући опсег.

Прецизније, експанзија је репрезентација реалног броја x секвенцом од n компоненти $(x_1, x_2, \dots, x_n)$ представљених у покретном зарезу за коју важи:

- вредност експанзије броја x је тачан, незаокружени, број који представља суму компонената експанзије;
- компонента x_i може бити једнака нули;
- компоненте су поређане по динамичком опсегу (експонентима);
- подсеквенца ненултих компоненти се не сме преклапати. Две компоненте у покретном зарезу x_i, x_{i+1} се не преклапају ако је најмање значајан (најмлађи) **ненулти** бит компоненте x_i значајнији (старији) од најзначајнијег **ненултог** бита компоненте x_{i+1}.

На слици 4.1.3.1 је приказан пример компоненти које се не преклапају a и b, као и пример компоненти које се преклапају b и c. Функције компоненти *lsb* и *msb* односе се на најмање значајан **ненулти** бит и најзначајнији **ненулти** бит компоненте, респективно. Услов да се две компоненте x_i и x_{i+1} не преклапају је стога: $lsb(x_i) > msb(x_{i+1})$.



Слика 4.1.3.1. Пример непреклопљених (a и b) и преклопљених (b и c) компоненти експанзије [12]

Са повећањем броја компонената експанзије тачност репрезентације расте, али расте и цена имплементације операција над експанзијом. Често се користи експанзија која се назива *double-double*, а представља пар (x_h, x_l) компоненти представљених у *binary64* формату. Детаљи ове имплементације су доступни у [12].

4.2. Поновљивост независно од тачности резултата

Осим претходно поменутих техника за постизање поновљивости, постоје технике који се не ослањају на побољшање тачности резултата. Ове технике елиминишу мане паралелизације које доводе до нумеричке непоновљивости. Као што је претходно поменуто, главне мане паралелизације су недетерминистичка паралелна редукција и различита пропагација грешака при заокруживању услед промене параметара система.

Једна од оваквих техника јесте 1-редукција аутора Џејмса Демела и Хонг Диеп Енгајена коју су 2015. године представили у свом раду [6]. Ова техника користи постојеће *error-free* трансформације и *a priori* заокруживања (*pre-rounding*) за акумулирање вредности. У паралелном окружењу користи само једну редукцију за сумирање парцијалних сума те је на тај начин ефикаснија од претходних.

Целобројна аритметика је асоцијативна, па се у одређеним ситуацијама може искористити да симулира операције са бројевима у покретном зарезу. Бројеви у покретном зарезу V се трансформишу у целе бројеве IV тако што се помноже одговарајућим фактором Q: $IV = Q \times V$. Фактор Q не сме бити превелики како се не би прекорачио опсег целих бројева који се може представити, али не сме бити ни премали како се број не би пресликао у нулу. У општем случају, коришћење целобројне аритметике на овај начин може повећати укупну апсолутну грешку услед смањења прецизности приликом трансформације.

У потпоглављу 3.3 поменута је техника форсирања редоследа извршавања операција за постизање слабе поновљивости. Детерминистичка паралелна редукција се ослања на балансирање посла на расположивим процесорима и фиксирање редоследа израчунавања. Пример ове имплементације се може наћи у [16].

Фиксирани редослед извршавања доводи до слабе поновљивости, јер се редослед фиксира за одређене параметре система (број процесних елемената – нити, процесора). Променом параметара система мења се редослед извршавања, па се мења и пропагација нумеричких грешака те тако и резултат. Стога је ова техника прихватљива само у ситуацијама када јака поновљивост није неопходна.

4.3. Широки акумулатор

У претходна два потпоглавља поменуте су различите технике које се користе за постизање поновљивости. Осим ових техника, често коришћена техника је широки акумулатор (*long accumulator*). Она обезбеђује потпуну тачност резултата, јер елиминише грешке при заокруживању на одређени формат, а самим тим елиминише и утицаје недетерминистичке паралелне редукције јер гарантује асоцијативност операција.

Немачки математичар Улрих Кулиш, један од пионира интервалне аритметике, предложио је употребу акумулатора довољно широког да покрије читав опсег експонената броја представљеног у покретном зарезу [17]. На тај начин, суме и скаларни производи постају тачне операције. Представио је три хардверске имплементације широког акумулатора и дао предлоге за разне оптимизације ове технике.

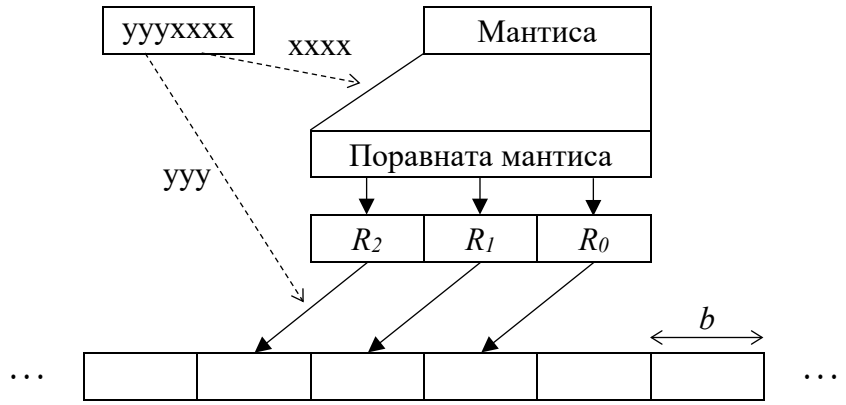
Међутим, како је хардверска имплементација ове технике често недовољно брза или скупа по питању површине на чипу, она није усвојена у општим архитектурама као што је x86. Иако хардверска имплементација није усвојена, ова техника се може имплементирати и софтверски на готово свим савременим архитектурама и то тако да даје прихватљиве перформансе, а гарантује поновљивост и тачност израчунавања.

Основна идеја софтверске имплементације ове технике је употреба одређеног броја целобројних величина за представљање реалних бројева у фиксном зарезу. Како би обезбедили компатибилност са постојећим форматима бројева у покретном зарезу, неопходно је да ширина акумулатора, w_a, буде довољно велика тако да се сваки број у покретном зарезу из посматраног формата може сместити у акумулатор.

4.3.1. Особине софтверске имплементације

Широки акумулатор стога делимо на речи ширине b бита. Оваквих речи има $N = \lceil w_a \div b \rceil$. Мантиса броја у покретном зарезу ширине $w_f + 1$ (1 скривени бит) се помера у зависности од вредности експонента и поравнава на речи акумулатора [18].

Померање се реализује из два дела: прво се селектују речи акумулатора на које се додаје мантиса, па се онда делови мантисе померају унутар речи. Уколико је b степен двојке ($b = 2^k$), померај унутар речи се добија као доњих k бита експонента, док се адреса речи добија као $w_e - k$ (w_e је ширина експонента) горњих бита. Ово је илустровано на слици 4.3.1.1.



Слика 4.3.1.1. Илустрација поравнавања мантисе на речи широког акумулатора [18]

Поравната мантиса је обично поравната на две или више речи. Тачније, након максималног помераја који износи $b - 1$, поравната мантиса је ширине $w_f + b$ бита и простире се на $S = \left\lceil \frac{w_f + b}{b} \right\rceil$ речи. Подела широког акумулатора на више речи има две предности:

- два дела померања се могу извршавати у паралели;
- акумулирање броја захтева само додавање на S речи.

Међутим, постоје и два проблема која произилазе из овакве имплементације. Први је пропагација преноса из једне речи акумулатора у другу што потенцијално може да захтева ажурирање свих речи изнад S циљаних речи. Други је знак мантисе, јер мантиса може бити и позитивна и негативна, па постоји могућност за појаву позајмица и пропагација истих.

За решавање проблема пропагација преноса могу се преклопити речи акумулатора и искористити неке од *carry-save* шема [19]. На овај начин се пропагација преноса одлаже и тако смањују вероватноће пропагације преноса кроз читав акумулатор. С друге стране ово захтева употребу већег броја речи као и пропагацију свих преноса на крају извршавања.

За решавање проблема знака мантисе акумулатор се може глобално посматрати као означени цео број представљен у комплементу двојке. На тај начин се користи постојећа целобројна аритметика. Последица овакве представе акумулатора је потреба за финалном негацијом негативне вредности у акумулатору како би се она представила у неком од формата покретног зареза.

4.3.2. Предности и мане широког акумулатора

У претходним потпоглављима описане су разне технике за постизање поновљивости. Компензације и експанзије дефинисане у одељцима 4.1.2 и 4.1.3, респективно, могу довести до резултата исте тачности. Разлика је у томе што су компензације доста брже од експанзија, али су слабије применљиве због потребе за акумулирањем грешке.

Неке од техника описане у потпоглављу 4.2 користе детерминистичку паралелну редукцију за постизање поновљивости. Међутим, недетерминизам се може јавити у пропагацији грешки при заокруживању приликом израчунавања локалних (парцијалних) сума, па ове технике не гарантују, нужно, поновљиве резултате.

Главне предности широког акумулатора јесу гарантована тачност израчунавања, јер не постоје заокруживања међурезултата, као и гарантована асоцијативност операција услед непостојања грешке при заокруживању. Софтверска имплементација ове технике, међутим, захтева знатне меморијске ресурсе и услед тога скупоцене меморијске приступе. Стога су перформансе ове технике знатно мање од претходно описаних и то увек треба имати на уму.

У овом раду је изабран управо широки акумулатор за имплементацију услед његове једноставности и једноставне примене у постојећим програмима. Стога је представљена имплементација широког акумулатора у виду C++ класе која се може лако применити и надоградити. Жеља аутора је да на овај начин омогући једноставан начин за постизање поновљивости како би тестирање програма било олакшано, као и да прикаже начин имплементације једне од поменутих техника.

5. ПРЕГЛЕД КОРИШЋЕНИХ ТЕХНОЛОГИЈА

У овом поглављу биће дат кратак увод у паралелно рачунарство уз осврт на модел дељене меморије који је примењен у раду. Осим овога, биће описане технологије коришћене за имплементацију примера за тестирање широког акумулатора.

5.1. Паралелно рачунарство

Претходно је у раду поменут појам паралелизма који се користи за имплементацију нумеричких алгоритама који се извршавају на савременим хетерогеним архитектурама. Основна идеја паралелизма је декомпозиција домена, где сваки процесни елемент (у даљем тексту процесор) добија поддомен проблема за решавање. На тај начин, могуће је ефикасно искористити савремене трендове у рачунарству (паралелне и хетерогене архитектуре) за постизање знатно бољих перформанси извршавања.

Паралелне рачунарске архитектуре се најпре разликују на основу начина организације меморије и приступа меморији од стране процесора. Меморија у паралелним рачунарским системима може бити дељена (*shared-memory*) или дистрибуирана (*distributed-memory*). Осим ових, постоји и хибридна, дистрибуирана дељена меморија (*distributed-shared-memory*) [20].

За извршавање нумерички захтевних алгоритама поменутих у раду најчешће се користе рачунарски кластери који су конфигурисани у режиму дистрибуиране дељене меморије. Међутим, аутору су приликом израде рада били доступни само рачунарски системи са дељеном меморијском архитектуром, па је на њих овде стављен акценат.

5.1.1. Дељена меморијска архитектура

Рачунарски системи који користе дељену меморијску архитектуру се, на основу времена приступа меморији (*latency*), класификују на [20]:

- UMA (*Uniform Memory Access*) – сви процесори имају исто време приступа свим меморијским локацијама и нема разлике приликом приступа некешираним локацијама. Користи се код реализације SMP (*Symmetric Multi-Processor*) машина.
- NUMA (*Non-Uniform Memory Access*) – машине овог типа се најчешће реализују повезивањем више SMP машина. За разлику од UMA архитектуре, бреме приступа локалној меморији је краће од времена приступа меморији другог SMP-а.

Дељена меморијска архитектура дозвољава приступ свим меморијским локацијама од стране свих процесора. Више процесора може радити независно у паралели и делити исте меморијске ресурсе. Ово знатно олакшава писање програма, јер нема потребе за експлицитном комуникацијом између процесора, осим за међусобно искључење приступа за коректно извршавање програма.

Главна мана ове меморијске архитектуре је слаба скалабилност између меморије и процесора. Додавањем процесора се геометријски повећава саобраћај на путањи између дељене меморије и процесора, а у случају система са кохерентним кеш меморијама, геометријски се повећава саобраћај за одржавање кохеренције кеш меморија. Осим тога, програмер је одговоран за синхронизацију приступа дељеној меморији како би гарантовао исправно извршавање програма.

5.1.2. Модел дељене меморије

Програмски модели представљају скуп програмских апстракција које програмеру омогућавају да на једноставан и транспарентан начин види софтвер и хардвер. Најчешће коришћени паралелни програмски модел на рачунарима са дељеном меморијом је модел дељене меморије [20]. У зависности од нивоа на ком је постигнут паралелизам разликују се:

1) паралелизација на нивоу задатака (*task-level parallelism*) – процеси (задаци) деле заједнички адресни простор којем асинхроно могу да приступају са захтевима за читање и упис. Механизми, као што су браве (*lock*) и семафори (*semaphore*), користе се за спречавање утркивања (*race condition*) и мртвих закључавања (*deadlock*).
2) паралелизација на нивоу нити (*thread-level parallelism*) – један „тежак“ (*heavy weight*) процес може имати више „лаких“ (*light weight*), конкурентних путања извршавања. У овом случају један програм се може поделити на више нити (декомпозиција домена) које деле меморију процеса и комуницирају међусобно кроз ажурирање дељене меморије. Механизми синхронизације су слични онима код паралелизације на нивоу задатака, с тим што се овај вид паралелизма обично имплементира неким библиотекама, а не на нивоу оперативног система.

Основна разлика ова два модела јесте ниво на ком се имплементира паралелизација. Паралелизација на нивоу процеса је скоро увек имплементирана на нивоу оперативног система и као таква има додатне режијске трошкове. С друге стране, паралелизација на нивоу нити има

доста мање режијске трошкове, јер се често имплементира на нивоу програмског језика, а нити једног процеса деле заједничке параметре, што код паралелних процеса није случај.

У рачунарству, паралелизација на нивоу нити је дуго позната и коришћена. Произвођачи хардвера су историјски имплементирали своје, власничке (*proprietary*), имплементације нити. Ове имплементације су се знатно разликовале што је отежавало писање преносивих паралелних програма.

Независни напори за стандардизацију имплементације нити произвели су две веома различите имплементације: POSIX Threads и OpenMP. О овим имплементацијама ће бити више речи у наредна два потпоглавља. Осим ових имплементација постоје и друге имплементације нити као што TBB (*Thread Building Blocks*), Microsoft нити, Java нити итд. [20].

5.2. POSIX Threads

POSIX (*Portable Operating System Interface*) представља фамилију IEEE стандарда 1003.*n* (*n* означава број стандарда) за одржавање компатибилности међу оперативним системима. POSIX дефинише апликативни програмски интерфејс (API), окружења командне линије и алате за софтверску компатибилност са различитим варијантама UNIX и других оперативних система.

POSIX Threads, или скраћено PThreads, паралелни је извршни модел који постоји независно од програмског језика. Дозвољава програму да контролише више различитих путања извршавања које се преклапају у времену. Свака путања се назива нит (*thread*) и креирање и контрола ових путањи је омогућена позивима које дефинише POSIX Threads API.

Најновија верзија POSIX.1 стандарда који дефинише и POSIX Threads API је IEEE Std. POSIX.1-2017 [21]. Стандардом је дефинисан скуп типова, функција и константи за програмски језик C. Имплементиран је у **pthread.h** заглављу и библиотеци **libpthread**.

Постоји око 100 имплементираних функција, све са префиксом **pthread_** и могу се поделити у четири групе:

- функције за управљање нитима – креирање, спајање нити и слично;
- функције за имплементацију међусобног искључења (*mutex*);
- функције за имплементацију условних променљивих (*condition variables*);
- функције за синхронизацију нити коришћењем брава (*locks*) и баријера (*barriers*).

5.3. OpenMP

OpenMP (*Open Multi-Processing*) је API вишег нивоа дизајниран за програмирање паралелних рачунарских система са дељеном меморијом. OpenMP чини скуп директива, рутина извршне библиотеке и променљива окружења. Представља индустријски стандард, дефинисан и подржан од стране групе главних хардверских и софтверских компанија, организација и индивидуа. Најновија верзија OpenMP стандарда је 5.0 [22] из 2018. године. API је дефинисан за програмске језике C, C++ и Fortran. Преносивост кода је обезбеђена имплементацијама на већини оперативних система: Solaris, UNIX (Linux и macOS) и Windows.

У језицима C/C++ OpenMP директива почиње са **#pragma omp**. Уколико се програм који садржи OpenMP директиве преводи као обичан секвенцијални код (без потребних OpenMP опција), директиве се просто игноришу. Паралелизација коришћењем OpenMP програмског модела је имплицитна, односно програмер није задужен за управљање нитима већ једноставно користи пружене директиве за паралелизацију.

Основни алат OpenMP модела јесте паралелни регион. Секвенцијални код написан у паралелном региону ће бити извршен у паралели. Унутар паралелног региона раде тимови нити. У оквиру директиве за означавање паралелног региона, **#pragma omp parallel**, неопходно је поставити одговарајуће параметре.

У параметре паралелног региона спада организација промењивих које се користе унутар региона. Промењиве се могу означити као дељене (*shared*) или приватне (*private*). Ове ознаке утичу на начин извршавања програма, јер у случају да више нити приступа истој промењивој постоји потреба за синхронизацијом за очување коректности програма.

Стога, осим паралелног региона, OpenMP дефинише и синхронизационе концепте, а то су баријере, критични региони, атомично ажурирање меморије и *single* региони. Ове синхронизационе примитиве се користе имплицитно и експлицитно. На крају сваког паралелног региона постоји баријера (која се може, опционо, елиминисати) која служи за синхронизацију нити зарад правилног извршавања остатка програма.

Паралелна редукција се може извршавати над промењивама унутар паралелног блока. На тај начин OpenMP пружа могућност за ефикасно рачунање суме, разлике, производа и количника елемената низа, а од верзије 3 омогућава и налажење минимума и максимума низа. Коначно, најбитнији концепт OpenMP модела су директиве за поделу посла. Најчешће коришћена директива за поделу посла је директива **for**.

6. ИМПЛЕМЕНТАЦИОНИ ДЕТАЉИ

У овом поглављу биће представљен начин имплементације широког акумулатора у виду C++ класе, као и делови имплементације репрезентативног тест примера.

6.1. Имплементација широког акумулатора у програмском језику C++

У 4. поглављу дата је архитектура широког акумулатора представљеног као низ целобројних величина. Дат је шематски приказ широког акумулатора и поступак акумулирања броја у покретном зарезу. Поменути су проблеми који могу настати приликом акумулирања и предложене идеје за решавање истих.

За потребе овог рада аутор је одлучио да широки акумулатор имплементира у облику C++ класе која има преклопљене све операторе од значаја и на тај начин се може користити уместо постојећег типа за *binary32* формат бројева у покретном зарезу – **float**. Оваква имплементација омогућава једноставну адаптацију постојећег програма имајући у виду повећане меморијске трошкове за имплементацију акумулатора. Стога је могуће једноставно постићи поновљивост и тачност резултата за тестирање исправности програма.

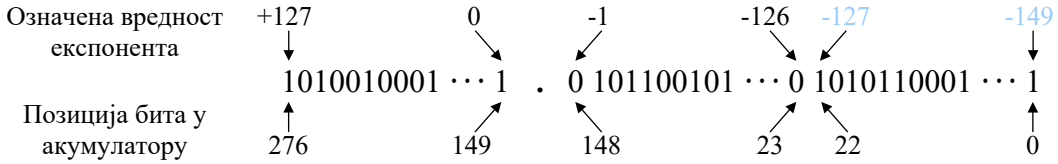
6.1.1. Одређивање параметара широког акумулатора

Потребно је, најпре, одредити ширину акумулатора за тачно представљање целог скупа бројева који се могу представити *binary32* форматом. У табели 2.3.2 дати су параметри различитих бинарних формата и у случају *binary32* формата ширина експонента у коду са вишком (E) је $w = 8$ бита, а ширина мантисе је $p = 24$ бита (23 бита у коду + 1 скривени бит). Постоје две резервисане вредности експонента у коду са вишком и то 0 (за кодирање нуле и субнормалних бројева) и 255 (за кодирање бесконачности и NaN вредности).

Вишак који се додаје на вредност експонента (e) износи 127 ($bias = 2^{w-1} - 1$). Сада се опсег означеног експонента може одредити према формули: $e = E - bias$. Са минималном вредношћу $E_{min} = 1$ и максималном вредношћу $E_{max} = 254$ имамо $e_{min} = E_{min} - bias = 1 - 127 = -126$ и $e_{max} = E_{max} - bias = 254 - 127 = 127$.

Сада на основу означеног опсега експонента и ширине мантисе можемо одредити број потребних бита лево и десно од тачке. Како означени експонент представља локацију бита највеће тежине мантисе у акумулатору (локацију лево од тачке почевши од 0), лако се може

закључити да је број потребних бита лево од тачке једнак $e_{max} - 0 + 1 = 127 + 1 = 128$. С друге стране, број потребних бита десно од тачке зависи и од ширине мантисе.



Слика 6.1.1.1. Позиције бита широког акумулатора у зависности од означене вредности експонента

Како је мантиса ширине 24 бита, а експонент дефинише локацију бита највеће тежине, за најмању вредност експонента $e_{min} = -126$ позиција бита највеће тежине мора бити 23 како би цела мантиса стала у акумулатор. Стога је број потребних бита десно од тачке једнак $-1 - e_{min} + 1 + w_f = -1 - (-126) + 1 + 23 = 126 + 23 = 149$ бита.

Коначно, сабирањем ове две вредности добијамо да је минимална потребна ширина акумулатора за смештање скупа бројева подржаних *binary32* форматом: $w_a = 128 + 149 = 277$ бита. Акумулатор ове ширине није могуће представити једном целобројном речју, па је потребно поделити акумулатор на више речи.

Сада је потребно одредити ширину једне речи која ће се користити за имплементацију акумулатора и на основу тога потребан број речи. Приликом бирања величине речи треба имати на уму однос величине речи и ширине мантисе зато што би употреба речи величине мање од величине мантисе захтевала преклапање мантисе са бар две суседне речи за акумулирање (слика 4.3.1.1). У одељку 4.3.1 дата је формула за рачунање потребног броја речи N на основу минималне ширине акумулатора w_a и ширине једне речи b: $N = \lceil w_a \div b \rceil$.

Број потребних речи величине 32 бита је: $N = \lceil 277 \div 32 \rceil = 9$, па би укупно меморијско заузеће било: $9 \times 32 = 288$ бита $= 36$ бајта. С друге стране, број потребних речи величине 64 бита је: $N = \lceil 277 \div 64 \rceil = 5$, па би акумулатор заузимао: $5 \times 64 = 320$ бита $= 40$ бајта. Може се закључити да је коришћење 32-битних целобројних величина меморијски ефикасније.

Стога је аутор за имплементацију решења користио 9 32-битних речи типа **uint32_t**. Треба приметити да при рачунању броја речи нису урачунати *carry-save* бити како би имплементација била нешто једноставнија. Међутим, као што је претходно речено, ова имплементација може бити и те како корисна у одређеним ситуацијама када програм изазива велики број пропагација преноса, односно позајмица.

Осим тога, како је број доступних бита већи од потребног, додатни бити се користе за представљање бројева већих од највећег броја који се може представити *binary32* форматом. На тај начин, нека израчунавања која доводе до прекорачења коришћењем **float** типа се могу успешно извршити коришћењем широког акумулатора. Наравно, уколико се резултат не може представити *binary32* форматом, приликом конверзије у **float**, добија се бесконачност.

Аутор је широки акумулатор имплементирао у облику другог комплемента због једноставнијих пропагација позајмица. Уколико настане потреба за позајмицом у највишој речи акумулатора, долази до инверзије знака акумулатора што ефикасно решава овај проблем. На тај начин, бит највеће тежине највише речи акумулатора одређује знак.

6.1.2. Имплементација операција сабирања и одузимања

Операција сабирања је основна операција која се може извршити над имплементираним типом. Преклопљени су оператори +=, −=, + и − за параметре типа **float**, као и за параметре типа константне класне референце. Ово омогућава коришћење класе на стандардни начин у изразима уместо типа **float**.

Први корак за додавање броја у покретном зарезу (**float**) у широки акумулатор је издвајање компоненти: знака, експонента и мантисе. За ту потребу креирана је структура **float_components** која чува ове компоненте и користи се у имплементацији разних функција.

```
typedef struct {
    bool negative;
    uint32_t exponent;
    uint32_t mantissa;
} float_components;
```

Код 6.1.2.1. Структура float_components

Две једноставне помоћне функције врше конверзију **float** у **float_components** и обратно, а њихове имплементације су дате у прилогу:

```
float_components extractComponents(float f);
float packToFloat(float_components components);
```

Код 6.1.2.2. Потписи помоћних функција extractComponents и packToFloat

Сада се, коришћењем ове структуре и ових метода једноставно издвајају компоненте броја:

```
float_components components = extractComponents(f);
```

Код 6.1.2.3. Издвајање компоненти броја

Други корак је одређивање речи акумулатора на које треба додати мантису на основу експонента. Усвојено је такво пресликавање где за експонент -126 (у коду са вишком = 1) мантиса заузима бите 23…0 најниже (нулте) речи акумулатора:

```
uint32_t k = (components.exponent + 22) >> 5;
```

Код 6.1.2.4. Одређивање индекса највише речи на коју треба додати део мантисе

Променљива **k** означава индекс највише речи на коју треба додати део мантисе. Међутим, мантиса потенцијално треба да се дода на две речи акумулатора, па се у зависности од вредности експонента она дели и додаје посебно на **k**-ту и **(k – 1)**-ву реч акумулатора:

```
bool split = ((components.exponent - 1) >> 5) < k;
```

Код 6.1.2.5. Одређивање да ли мантису треба додати на две суседне речи

Коначно, за додавање мантисе потребно је исту померити за одговарајући број бита унутар саме речи, па се позива помоћна метода **add** која врши сабирање, односно одузимање, и пропагацију преноса, односно позајмица.

```
uint32_t hi_width = (components.exponent - 9) & 0x1F;
if (!split) {
    add(k, components.mantissa << (hi_width - 24), components.negative);
} else {
    add(k - 1, components.mantissa << (8 + hi_width),
               components.negative);
    add(k, components.mantissa >> (24 - hi_width), components.negative);
}
```

Код 6.1.2.6. Позив методе add са одговарајућим параметрима

```
void LongAccumulator::add(uint32_t idx, uint32_t val, bool negative)
{
    while (idx < ACC_SIZE) {
        uint32_t old = acc[idx];
        if (!negative) {
            acc[idx] += val;
            if (acc[idx] < old) { // overflow
                ++idx;
                val = 1;
            } else break;
        } else {
            acc[idx] -= val;
            if (acc[idx] > old) { // underflow
                ++idx;
                val = 1;
            } else break;
        }
    }
}
```

Код 6.1.2.7. Имплементација помоћне методе add

Метода **add** је имплементирана итеративно, а у зависности од параметра **negative** на реч са индексом **idx** се додаје, односно одузима прослеђена вредност **val**. У случају прекорачења, инкрементира се **idx** и поставља се **val** на 1, а онда врши се пропагација преноса, тј. позајмица док је потребно, а максимално до највише речи акумулатора.

Остали оператори (−=, + и −) су имплементирани коришћењем оператора +=, а њихове имплементације, као и имплементације оператора са параметрима константних класних референци су тривијалне и нису од интереса за овај део рада.

6.1.3. Имплементација конверзије вредности назад у float

Након завршетка акумулације, а како би тип **LongAccumulator** могао да се користи на месту где је коришћен тип **float**, потребно је имплементирати конверзију и заокруживање вредности на величину типа **float**. Конверзија је имплементирана у облику оператора **()** чија је комплетна имплементација дата у прилогу. Као што је претходно поменуто, знак резултујућег броја је једнак највишем биту највише речи акумулатора:

```
components.negative = acc[ACC_SIZE - 1] & 0x80000000;
```

Код 6.1.3.1. Одређивање знака резултујућег броја

Како би имплементација била нешто једноставнија, користи се апсолутна вредност акумулатора која користи унарни оператор –. Имплементација овог оператора је тривијална и представља одузимање вредности акумулатора од нове инстанце акумулатора (вредност 0).

```
LongAccumulator absolute = components.negative ? -*this : *this;
```

Код 6.1.3.2. Одређивање апсолутне вредности широког акумулатора

За одређивање експонента и мантисе потребно је одредити позицију најзначајнијег бита у акумулатору, тј. индекс најзначајније речи акумулатора (**word_idx**) и индекс најзначајнијег бита у тој речи (**bit_idx**). На основу ових вредности се разматра да ли је вредност у акумулатору 0, субнормална вредност, нормална вредност или бесконачност. Специјални случајеви се обрађују посебно како би резултат био у складу са IEEE-754 стандардом.

```
int word_idx = ACC_SIZE - 1;
while (word_idx >= 0 && absolute.acc[word_idx] == 0) {
    word_idx--;
}
...
bitset<32> bits(absolute.acc[word_idx]);
int bit_idx = 31;
while (bit_idx >= 0 && !bits.test(bit_idx)) {
    bit_idx--;
}
```

Код 6.1.3.4. Одређивање индекса најзначајније речи акумулатора и најзначајнијег бита у речи

Сада се експонент на основу израчунатих индекса **word_idx** и **bit_idx** одређује пресликавањем инверзним пресликавању описаним у одељку 6.1.2:

```
components.exponent = (word_idx << 5) + bit_idx - 22;
```

Код 6.1.3.5. Одређивање вредности експонента у коду са вишком на основу одређених индекса

Први корак при одређивању мантисе је изоловање бита мантисе из најзначајније речи акумулатора помоћу одговарајуће маске:

```
uint32_t mask = ((uint64_t) 1 << (bit_idx + 1)) - 1;
components.mantissa = absolute.acc[word_idx] & mask;
```

Код 6.1.3.6. Изоловање дела мантисе из најзначајније речи акумулатора помоћу маске

Након изоловања бита из највише речи, врши се одговарајући померај, а у случају да је мантиса подељена, новом маском се изолују бити наредне речи. Коначно, позива се помоћна метода **round** која врши заокруживање мантисе са одговарајућим параметрима који одређују позицију првог бита у акумулатору који се налази иза мантисе.

```
if (bit_idx > 23) {
    components.mantissa >>= bit_idx - 23;
    round(absolute, components, word_idx, bit_idx - 24);
} else if (bit_idx == 23) {
    round(absolute, components, word_idx - 1, 31);
} else {
    components.mantissa <<= 23 - bit_idx;
    components.mantissa |= absolute.acc[word_idx - 1] >> (bit_idx + 9);
    round(absolute, components, word_idx - 1, bit_idx + 8);
}
```

Код 6.1.3.7. Изоловање осталих бита мантисе и позив методе round

Помоћна метода **round** мантису заокружује на основу постављеног параметра система за начин заокруживања бројева у покретном зарезу дефинисан језиком C++. Међутим, за потребе овог рада је од значаја само заокруживање на најближу вредност које је имплементирано на основу правила датих у одељку 2.4.1.

Како заокруживање у одређеним ситуацијама подразумева инкрементирање тренутне вредности мантисе, може доћи до прекорачења. У том случају, одбацује се најнижи бит мантисе, а експонент се инкрементира, па се проверава да ли је прекорачена вредност експонента. Уколико јесте, заокруживањем је добијена бесконачност.

```
void LongAccumulator::round(const LongAccumulator &acc,
                            float_components &components,
                            int word_idx, int bit_idx)
{
    ...
    bitset<32> bits(acc.acc[word_idx]);
    if (!bits.test(bit_idx)) {
        return; // case 1
    }
    bool allzero = true;
    for (int i = bit_idx - 1; allzero && i >= 0; --i) {
        if (bits.test(i)) {
            allzero = false;
        }
    }
    if (allzero) {
        for (int i = word_idx - 1; allzero && i >= 0; --i) {
            if (acc.acc[i] > 0) {
                allzero = false;
            }
        }
    }
    if (!allzero) {
        components.mantissa++; // case 2, check overflow
    } else if ((components.mantissa & 0x1) == 0) {
        return; // case 3a
    } else {
        components.mantissa++; // case 3b, check overflow
    }
    ...
    // Check if rounding caused overflow...
    if (components.mantissa > 0xFFFFFF) {
        components.mantissa >>= 1;
        components.exponent++;
        if (components.exponent > 0xFE) {
            components.exponent = 0xFF;
            components.mantissa = 0;
        }
    }
}
```

Код 6.1.3.8. Имплементација заокруживања на најближу вредност у помоћној методи round

6.1.4. Остали елементи имплементације

У претходна два одељка дати су најбитнији елементи имплементације: аритметички оператори и оператор конверзије **()**. Осим ових оператора имплементирани су и поредбени оператори као и оператор излаза, односно штампе на излазни ток.

Имплементација поредбених оператора је једноставна и базира се на имплементацији оператора < са параметром константне референце на класу **LongAccumulator**. Имплементација оператора < подразумева поређење знакова два операнда, а у случају да су знаци исти, пореде се апсолутне вредности. На тај начин је избегнуто поређење свих речи акумулатора сваки пут, те је постигнута мала оптимизација.

```
bool operator<(const LongAccumulator &l, const LongAccumulator &r)
{
    bool sign_l = l.acc[ACC_SIZE - 1] & 0x80000000;
    bool sign_r = r.acc[ACC_SIZE - 1] & 0x80000000;
    if (sign_l && !sign_r) {
        return true;
    } else if (!sign_l && sign_r) {
        return false;
    }
    LongAccumulator positive_l = sign_l ? -l : l;
    LongAccumulator positive_r = sign_r ? -r : r;
    int i = ACC_SIZE - 1;
    while (i >= 0 && positive_l.acc[i] == positive_r.acc[i]) {
        i--;
    }
    if (i < 0) {
        return false; // equal
    }
    return sign_l ? positive_l.acc[i] > positive_r.acc[i]
                  : positive_l.acc[i] < positive_r.acc[i];
}
```

Код 6.1.4.1. Имплементација поредбеног оператора: <

Оператор << штампа вредност у акумулатору у бинарном облику и то прво знак вредности па најзначајније бите апсолутне вредности. Овај оператор је првенствено коришћен приликом тестирања имплементације, али како је користан за демонстрацију рада акумулатора, његова имплементација је дата у прилогу.

6.2. Имплементација референтног тест примера

За валидацију имплементације широког акумулатора написан је једноставан тест пример који је базиран на суми насумично генерисаних елемената низа. Сумирање је имплементирано секвенцијално и паралелно (коришћењем PThreads библиотеке) у циљу анализе поновљивости и перформанси технике.

За генерисање насумичних елемената низа користи се „семе“ (*seed*) које се може проследити параметром програма. Подразумевана вредност „семена“ је: 1549813198. На овај начин је омогућено генерисање истих елемената за исте параметре програма.

Имплементација референтног тест примера користи генератор случајних бројева уграђен у језик од стандарда C++11: *Mersenne Twister 19937*. Овај генератор има добре карактеристике и користи се одвојено за генерисање знака, експонента и мантисе коришћењем униформне расподеле. Мана овог генератора су перформансе, јер како је се генерисање низа случајних бројева мора користити једна нит, извршавање ове функције траје доста дуго.

```
void generate_elements()
{
    mt19937 gen(seed);
    uniform_int_distribution<uint32_t> sign_dist(0, 1);
    uniform_int_distribution<uint32_t> exponent_dist(exponent_min +
            EXPONENT_BIAS, exponent_max + EXPONENT_BIAS);
    uniform_int_distribution<uint32_t> mantisa_dist(0, (1 << 23) - 1);
    elements = new vector<float>(element_count);
    for (int i = 0; i < element_count; ++i) {
        uint32_t bits = (sign_dist(gen) << 31) |
                        (exponent_dist(gen) << 23) |
                        (mantisa_dist(gen) << 0);
        static_assert(sizeof(uint32_t) == sizeof(float));
        float number;
        memcpy(&number, &bits, sizeof(uint32_t));
        (*elements)[i] = number;
    }
}
```

Код 6.2.1. Имплементација функције за генерисање насумичних елемената низа

За валидацију поновљивости, елементи низа се након сваке итерације насумично пермутују коришћењем уграђене функције **shuffle**. Паралелна имплементација поседује додатну насумичност при подели елемената низа свакој нити и редукцији парцијалних сума. На тај начин, сви ефекти који утичу на поновљивост и код секвенцијалне и код паралелне имплементације се могу тестирати.

Подразумевано су укључене све пермутације за анализу поновљивости. Међутим, за коректну анализу перформанси имплементираног решења потребно је обезбедити поделу елемената низа нитима на идентичан начин приликом сваког покретања.

За ову потребу је имплементирана опција која се прослеђује приликом покретања тест примера. Опцијама програма могуће је подесити и „семе“ генератора, број елемената, број нити, најнижу и највишу вредност генерисаних експонената и број итерација за анализу поновљивости.

7. МЕТОДОЛОГИЈА АНАЛИЗЕ

У овом поглављу је описана методологија анализе имплементације широког акумулатора. Детаљно је описано тест окружење и наведена како развојна, тако и хардверска платформа на којој је вршена евалуација имплементације широког акумулатора.

7.1. Развојна платформа

Комплетан развој имплементације класе **LongAccumulator**, референтног тест примера као и додатна два практична тест примера је обављен коришћењем Visual Studio Code едитора са проширењем за програмски језик C++ под оперативним системом Manjaro KDE Plasma и верзијом Linux кернела 5.8.6-1. Коришћени компајлер је GNU C++ компајлер верзије 10.2.0, а C++ стандард C++14. Поред овога, за развој су коришћене библиотеке **libpthread** и **libgomp** за имплементацију паралелизације у коришћеним примерима.

7.2. Хардверска платформа

Развојни рачунар чини Intel Core i7 6700HQ процесор са радним тактом од 3.1 GHz са четири физичка језгра и осам логичких коришћењем Intel Hyper-Threading технологије. Меморија коју рачунар поседује је капацитета 16 GB, а такт 2133 MHz. Овај рачунар је коришћен за иницијално тестирање имплементације.

За тестирање је коришћен други рачунар који ради под оперативним системом Ubuntu 18.04 LTS и верзијом Linux кернела 4.15.0-20. Верзија GNU C++ компајлера инсталираног на овом рачунару је 7.3.0, али имплементација користи C++14 стандард који овај компајлер подржава, те се стога ова верзија битно не разликује од верзије 10.2.0. Овај рачунар поседује Intel Core i7 4790K са четири физичка и осам логичких језгара радног такта 4.0 GHz. Рачунар такође поседује 16 GB RAM меморије.

7.3. Тест окружење

Осим референтног тест примера чија је функционалност описана у потпоглављу 6.2, за анализу поновљивости и перформанси имплементираног решења коришћена су и два програма: **kmeans** из *Rodinia benchmark* пакета [23] и **mri-gridding** из *Parboil benchmark* пакета [24]. Ови програми представљају реалне тест примере који поседују одређени степен непоновљивости резултата у својим паралелним имплементацијама.

Програм **kmeans** врши кластеризацију података методом *k*-средњих вредности (*k-means clustering*). То је метод који врши алоцирање *n* објеката у *k* кластера у којем сваки објекат припада кластеру са најближом средњом вредношћу. Објекат се састоји од низа вредности – особина (*features*). Поделом објеката у потскластере, алгоритам представља све објекте помоћу њихових средњих вредности (тзв. центроида потскластера).

Програм **mri-gridding** врши мапирање неуниформних података у 3D простору на регуларну мрежу у 3D простору. Свака тачка из неуниформног 3D простора доприноси суседним тачкама у регуларној мрежу у складу са *Kaiser-Bessel* функцијом за одређивање растојања. Овај програм је од значаја за MRI (*Magnetic Resonance Imaging*) те стога представља занимљив пример примене ове технике.

Паралелна имплементација програма **kmeans** дата у *Rodinia benchmark* пакету користи OpenMP библиотеку и демонстрира слабу поновљивост јер се променом параметара система (броја нити за извршавање) мењају резултати. Ова имплементација је једноставно измењена тако да користи имплементирани тип **LongAccumulator** уместо типа **float**.

С друге стране, аутор је користио сопствену паралелну имплементацију програма **mri-gridding** услед лоше (неоптимизоване) имплементације за централни процесор коју пружа *Parboil benchmark* пакет. Тестиране су две различите имплементације овог алгоритма од којих једна демонстрира потпуни недостатак поновљивости услед коришћења брава, док друга демонстрира слабу поновљивост услед коришћења паралелне редукције.

Прва имплементација овог алгоритма користи браве за обезбеђивање међусобног искључења приликом приступа елементима резултујућих низова **gridData** и **sampleDensity**. Како је вероватноћа приступа узастопним локацијама низа мала, једна брава штити неколико елемената резултујућих низова. Очигледно је да је приступ локацијама недетерминистички па се резултати разликују од једног до другог извршавања програма. Овај алгоритам има нешто слабије перформансе услед серијализације приступа меморијским локацијама.

Друга имплементација користи паралелну редукцију и засебне копије резултујућих низова за сваку нит. У овом случају свака нит ажурира локалне копије, а како је подела елемената (узорака) нитима идентична за свако извршавање то су и парцијалне суме (низови парцијалних сума) идентичне. Паралелна редукција која се овде користи има слабе перформансе услед приступа меморијски удаљеним подацима, па долази до честе инвалидације кешева. Овај проблем ће бити демонстриран у наредном поглављу.

8. РЕЗУЛТАТИ АНАЛИЗЕ И ДИСКУСИЈА

У овом поглављу дата је анализа поновљивости на основу резултата добијених помоћу референтног тест примера. Осим тога, анализирана је и цена поновљивости у секвенцијалним и паралелним окружењима у свим тест примерима.

8.1. Анализа поновљивости широког акумулатора

За анализу поновљивости коришћен је референтни тест пример описан у потпоглављу 6.2. Вариран је број генерисаних елемената низа, број понављања израчунавања и број нити коришћених у паралелној имплементацији. Добијени су резултати који униформно показују да је имплементирано решење поновљиво независно од параметара тест примера.

За број елемената коришћене су вредности у опсегу: 10^2 – 10^8, а број понављања је, услед великог времена извршавања, рачунат по формули: 10^8 / број елемената. На тај начин је, за већи број елемената низа, број понављања мањи, и обратно. За број нити су коришћене вредности: 1, 2, 4 и 8.

Стандардна секвенцијална и паралелна имплементација суме елемената низа није поновљива и већ после једног понављања производи различите резултате. С друге стране, имплементирано решење увек производи исте резултате и у секвенцијалној и у паралелној имплементацији. Стога је, применом овог решења, омогућено поређење резултата секвенцијалне и паралелне имплементације програма.

8.2. Анализа перформанси широког акумулатора

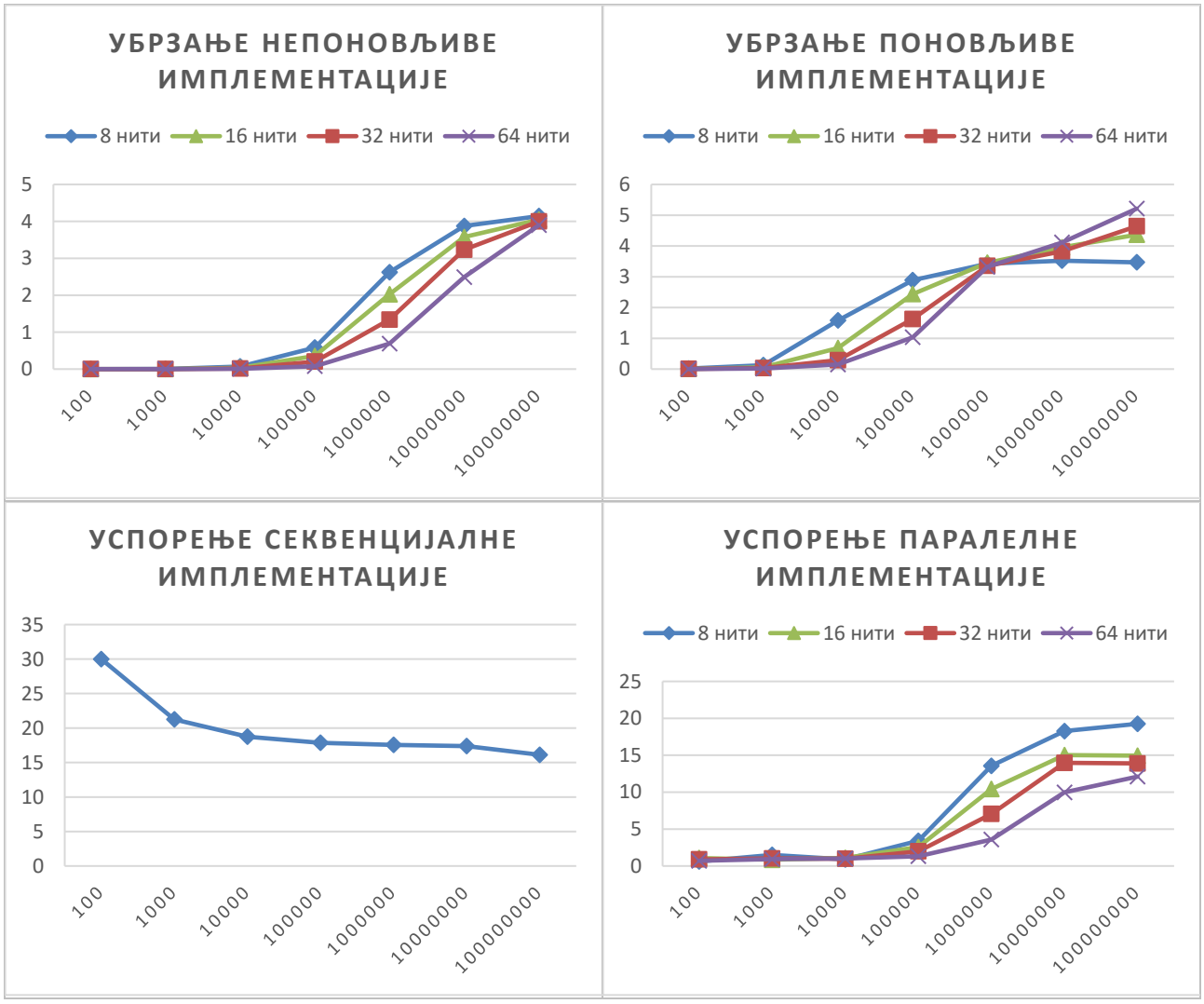
За анализу перформанси имплементираног решења, коришћена су сва три примера, па је ово потпоглавље на тај начин организовано. Сваки тест пример је компајлиран са укљученим оптимизацијама (-O2), јер се на тај начин може добити реалнија слика перформанси овог решења. Сви резултати представљају средњу вредност од 3 понављања.

8.2.1. Референтни тест пример

Коришћењем референтног тест примера могуће је, између осталог, варирати број елемената низа и опсег експонената генерисаних бројева. Варирање броја елемената низа је од интереса из разлога што директно утиче на време извршавања програма. С друге стране, опсег експонената је од интереса зато што утиче на број пропагација преноса/позајмица.

Анализа перформанси се може спровести из два угла посматрања. Могуће је разматрати убрзања паралелних имплементација пре и после примене широког акумулатора. На тај начин се може закључити да ли, и у којој мери, паралелизација повећава перформансе програма.

С друге стране, могуће је посматрати одвојено успорења секвенцијалне и паралелне имплементације након примене ове технике. На тај начин је могуће утврдити цену поновљивости што свакако даје комплетнију слику перформанси широког акумулатора.

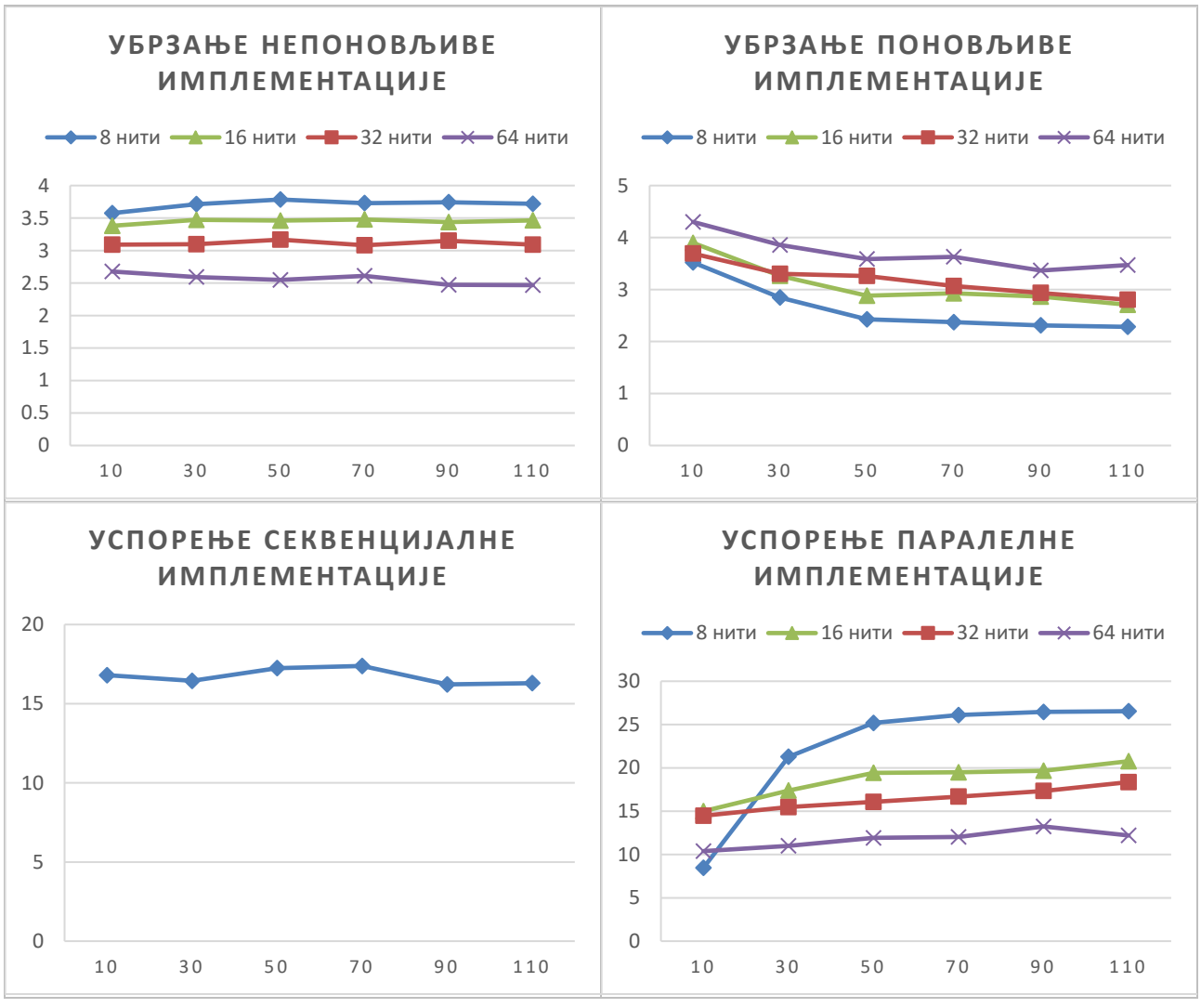


Слика 8.2.1.1. Перформансе у зависности од броја елемената низа

Треба нагласити да је опсег експонената коришћен за ову анализу: –10-10. Разматрајући приказана убрзања може се закључити да, пре примене широког акумулатора, повећањем броја нити преко 8 убрзање спорије расте. Разлог за то је чињеница да коришћени процесор (Intel Core i7-4790K) поседује само 4 физичка, тј. 8 логичких језгара. Креирањем већег броја нити се не постижу боља убрзања.

Са друге стране, након примене широког акумулатора, убрзање са 8 нити стагнира и достиже свој максимум за низ величине 10^6 елемената. Креирањем већег броја нити, у овом случају, убрзања настављају да расту. Потенцијални разлог за то јесте број меморијских приступа потребан за коришћење широког акумулатора. Нити већи део времена чекају читање/упис у меморију, па се повећавањем броја нити повећава ефикасност програма.

Разматрајући цену поновљивости, односно успорење програма након примене широког акумулатора, може се приметити да се, за секвенцијалну имплементацију, успорење креће у опсегу 15-30 пута. С друге стране, у случају паралелне имплементације, повећањем броја нити опада успорење и креће се од 0.1-20 пута.



Слика 8.2.1.2. Перформансе у зависности од максималне апсолутне вредности експонената

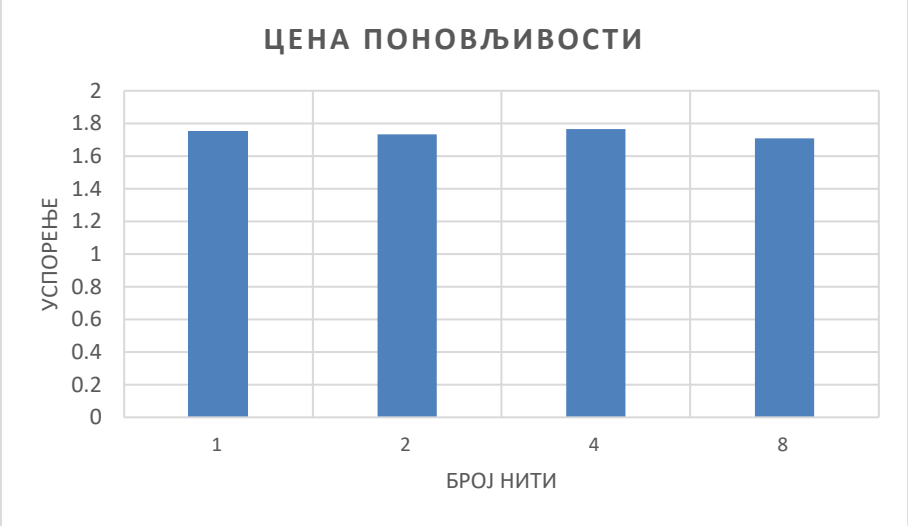
Треба назначити да је број елемената низа коришћен за ову анализу 10^7. Пре примене широког акумулатора, убрзања су готово независна од опсега експонената. Поново су

перформансе мање када је број нити већи од 8, али је убрзање готово константно. С друге стране, након примене широког акумулатора, убрзања опадају са растом опсега експонента. Један од разлога за ово понашање је повећан број пропагација преноса/позајмица.

Да је коришћена *carry-save* шема за имплементацију акумулатора, број пропагација био би смањен па би и перформансе биле нешто боље. Што се тиче успорења, тј. цене поновљивости, она је готово константна за секвенцијалну имплементацију и креће се у опсегу 16.2-17.4. Са друге стране, у случају паралелне имплементације поново примећујемо да повећање броја нити доводи до смањења цене поновљивости.

8.2.2. Програм kmeans

Као што је претходно речено, паралелна имплементација овог програма демонстрира слабу поновљивост. Променом броја нити које користи паралелна имплементација, мењају се резултати. Стога је, у овом случају, поновљивост од значаја за проверу исправности паралелне имплементације.



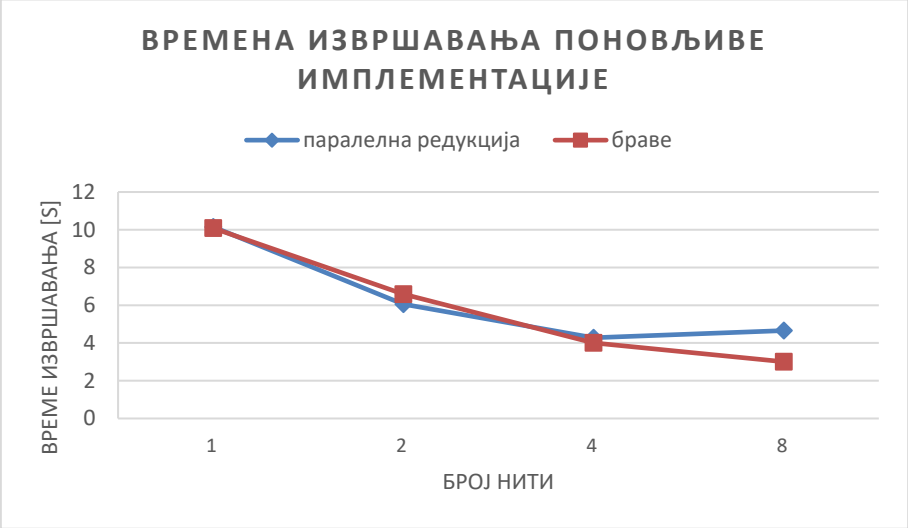
Слика 8.2.2.1. Цена поновљивости у зависности од броја нити

На слици 8.2.2.1 дат је хистограм цене поновљивости (успорења) у зависности од броја нити са којим се покреће паралелна имплементација. У овом случају, цена поновљивости је знатно мања него код референтног тест примера, што говори да је примена ове технике понекад оправдана у реалним програмима. Могући разлог за то је чињеница да је број сумирања у овом програму знатно мањи него код референтног тест примера. Референтни тест пример покренут са 8 нити и 10^7 елемената захтева ~10^6 операција сабирања по акумулатору (нити). С друге стране, овај програм захтева ~10^2 операција сабирања по акумулатору.

8.2.3. Програм mri-gridding

Коначно, коришћене паралелне имплементације овог програма демонстрирају и слабу поновљивост и потпуну непоновљивост. Пре него што размотримо цену поновљивости, треба размотрити времена извршавања поновљиве имплементације са паралелном редукцијом и бравама. На тај начин можемо закључити која је имплементација скалабилнија.

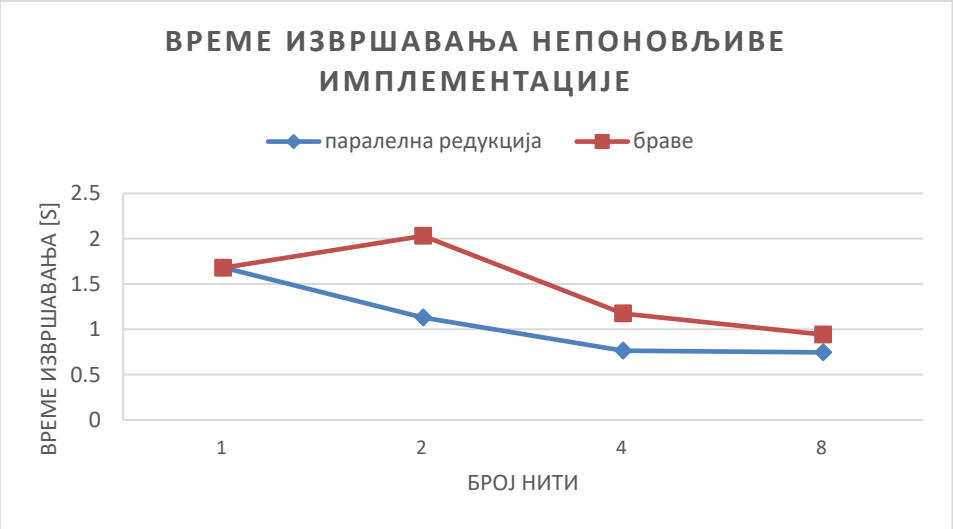
Наиме, повећањем броја нити у имплементацији са паралелном редукцијом се знатно повећава однос времена потребног за редукцију и времена потребног за рачунање парцијалних резултата. С друге стране, имплементација са бравама нема овај проблем јер је цена синхронизације готово фиксна, односно веома споро расте са порастом броја нити.



Слика 8.2.3.1. Времена извршавања поновљиве имплементације у зависности од броја нити

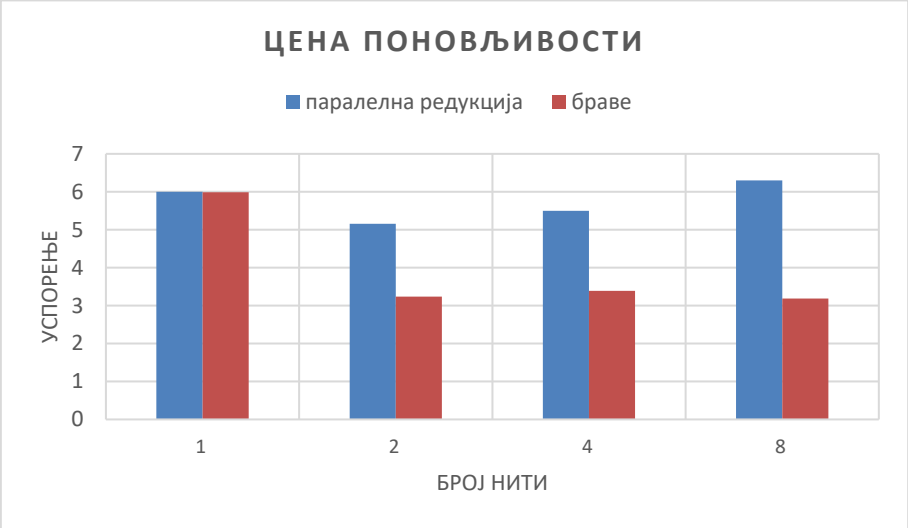
На слици 8.2.3.1 приказан је пораст времена извршавања у имплементацији са паралелном редукцијом приликом повећања броја нити са 4 на 8. С друге стране, имплементација са бравама нема овај пораст и сада је бржа од друге имплементације. Ово јасно показује да би даљим повећањем броја нити имплементација са бравама имала боље перформансе од имплементације са паралелном редукцијом.

Пре него што размотримо цену поновљивости, тј. успорење, потребно је прво размотрити однос времена извршавања непоновљивих имплементација. У овом случају, очекивано је да паралелна редукција има нешто боље перформансе услед мање цене редукције у односу на поновљиву имплементацију са широким акумулатором.



Слика 8.2.3.2. Времена извршавања непоновљиве имплементације у зависности од броја нити

Са графика на слици 8.2.3.2 јасно се види да су времена извршавања паралелне имплементације са бравама нешто већа од оне са паралелном редукцијом. Овај резултат је готово супротан резултату који смо имали за поновљиве имплементације. Разлог за то је мања цена паралелне редукције када се не користи широки акумулатор.



Слика 8.2.3.3. Цена поновљивости у зависности од броја нити

Са графика на слици 8.2.3.3 постоји јасан тренд пораста цене поновљивости у имплементацији са паралелном редукцијом. С друге стране, услед описаних односа времена извршавања непоновљиве и поновљиве имплементације са бравама, цена поновљивости је значајно мања и нема тренд раста какав има паралелна редукција. Ово још једном потврђује бољу скалабилност коју пружа имплементација са бравама.

9. Закључак

Експоненцијални развој савремених рачунарских система омогућио је развитак у разним научним пољима. Потреба за извршавање нумеричких алгоритама довела је до стандардизације аритметике у покретном зарезу у виду IEEE-754 стандарда. Међутим, услед ограниченог меморијског капацитета, постоји незаобилазна потреба за заокруживањем међурезултата израчунавања како би могли да буду смештени у рачунару.

Пропагација нумеричких грешака насталих при заокруживању међурезултата може имати знатан ефекат на тачност резултата. Тренд у рачунарству је развој паралелних програма које уносе недетерминизам у редослед извршавања операција. Недетерминизам у редоследу мења начин пропагације нумеричких грешака, те резултати постају непоновљиви.

Поновљивост је од значаја из неколико разлога. Најпре су битски идентични резултати корисни, а понекад и неопходни, за тестирање и исправљање грешака у програму. Осим тога, поновљивост је од значаја за извршавање одређених програма. Симулације, какве се користе у науци, су нестабилне, те су њихови резултати непоновљиви и често нетачни.

У овом раду представљена је имплементација широког акумулатора, технике која у потпуности елиминише непоновљивост резултата. За анализу перформанси овог решења коришћен је референтни тест пример који симулира најгори могући случај непоновљивости. Коришћена су и два паралелна програма за тестирање перформанси које се могу очекивати у реалним применама ове технике.

За истраживање и даљи развој, аутор предлаже усвајање *carry-save* шеме која би поправила перформансе у случајевима са великим опсегом експонената. Техника 1-редукције аутора Џејмса Демела [6] представља знатно бржи начин за постизање поновљивости, али по цену тачности резултата. Уколико је тачност, пак, неопходна, хијерархијска техника каква је имплементирана у раду [2] користи најбоље особине широког акумулатора и других поменутих техника за постизање и тачности и добрих перформанси.

Аутор се захваљује доц. др Марку Мишићу на предлогу ове теме, помоћи и саветима током израде рада. Такође, аутор се неизмерно захваљује својим родитељима, Борјани и Васку Николов, на подршци током целокупног школовања. Аутор се посебно захваљује свом брату Душану на консултацијама и тетки Радмили Милановић на подршци током студија.

Литература

[1] „IEEE Standard for Floating-Point Arithmetic“, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp.1-84, 22. јул 2019., doi: 10.1109/IEEESTD.2019.8766229

[2] S. Collange, D. Defour, S. Graillat, R. Iakymchuk, „Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures“, 10. септембар 2015., HAL Id: hal-00949355v4

[3] „IEEE Standard for Binary Floating-Point Arithmetic“, *ANSI/IEEE Std 754-1985*, pp.1-20, 12. октобар 1985., doi: 10.1109/IEEESTD.1985.82928

[4] „IEEE Standard for Floating-Point Arithmetic“, *IEEE Std 754-2008*, pp.1-70, 29. август 2008., doi: 10.1109/IEEESTD.2008.4610935

[5] „IEEE Standard for Radix-Independent Floating-Point Arithmetic“, *ANSI/IEEE Std 854-1987*, pp.1-19, 5. октобар 1987., doi: 10.1109/IEEESTD.1987.81037

[6] J. Demmel, H. D. Nguyen, „Parallel reproducible summation“, *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 2060-2070, 1. јул 2015., doi: 10.1109/TC.2014.2345391

[7] W. W. Smari, M. Bakhouya, S. Fiore, G. Aloisio, „New advances in High Performance Computing and simulation: parallel and distributed systems, algorithms, and applications“, *Concurrency and Computation: Practice and Experience*, vol. 28, no. 7, pp. 2024–2030, 19. фебруар 2016., doi: 10.1002/cpe.3774

[8] Y. He, C. H. Q. Ding, „Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications“, *The Journal of Supercomputing*, vol. 18, no. 3, pp. 259-277, март 2001., doi: 10.1023/A:1008153532043

[9] M. A. Cleveland, T. A. Brunner, N. A. Gentile, and J. A. Keasler, „Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle Monte Carlo simulations“, *Journal of Computational Physics*, vol. 251, pp. 223–236, 15. октобар 2013., doi: 10.1016/j.jcp.2013.05.041

[10] M. Taufer, O. Padron, P. Saponaro, S. Patel, „Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs“, *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, 2010, pp. 1-9, doi: 10.1109/IPDPS.2010.5470481

[11] R. W. Robey, J. M. Robey, R. Aulwes, „In search of numerical consistency in parallel programming“, *Parallel Computing*, vol. 37, no. 4-5, pp. 217–229, април 2011., doi: 10.1016/j.parco.2011.02.009

[12] R. Nheili, „How to improve the numerical reproducibility of hydrodynamics simulations: analysis and solutions for one open-source HPC software“, докторска дисертација, Université de Perpignan Via Domita, 2016, HAL Id: tel-01418384

[13] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, *Handbook of Floating-Point Arithmetic*, Birkhäuser, 2010

[14] O. Villa, D. Chavarría-Miranda, V. Gurumoorthi, A. Márquez, S. Krishnamoorthy, „Effects of floating-point non-associativity on numerical computations on massively multithreaded systems“, децембар 2010.

[15] N. J. Higham, *Accuracy and stability of numerical algorithms 2nd ed.*, Society for Industrial and Applied Mathematics (SIAM), 2002.

[16] D. Defour, S. Collange, „Reproducible floating-point atomic addition in data-parallel environment“, *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*, Lodz, 2015, pp. 721-728, doi: 10.15439/2015F86

[17] U. Kulisch, *Computer arithmetic and validity: Theory, implementation, and applications 2nd ed.*, De Gruyter, 2013.

[18] Y. Uguen, F. de Dinechin. „Design-space exploration for the Kulisch accumulator“, 20. март 2017., HAL Id: hal-01488916v2

[19] D. Defour, F. de Dinechin: „Software carry-save for fast multiple-precision algorithms“, *35th International Congress of Mathematical Software*, 2002.

[20] B. Barney, *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/, 2020.

[21] IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7“, *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp.1-3951, 31. јануар 2018., doi: 10.1109/IEEESTD.2018.8277153

[22] OpenMP Application Programming Interface Version 5.0 November 2018, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, 2020.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, K. Skadron, „Rodinia: A benchmark suite for heterogeneous computing“, *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, 2009, pp. 44-54, doi: 10.1109/IISWC.2009.5306797

[24] J. A. Stratton, C. I. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. Liu, W. Hwu. „Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing“, 2012.

Списак скраћеница

API	*Application Programming Interface*
EFT	*Error-Free Transformation*
FMA	*Fused Multiply and Accumulate*
HPC	*High Performance Computing*
IEEE	*Institute of Electrical and Electronics Engineers*
LSB	*Least Significant Bit*
MRI	*Magnetic Resonance Imaging*
MSB	*Most Significant Bit*
NUMA	*Non-Uniform Memory Access*
NaN	*Not-a-Number*
OpenMP	*Open Multi-Processing*
POSIX	*Portable Operating System Interface*
PThreads	*POSIX Threads*
SMP	*Symmetric Multi-Processor*
TBB	*Thread Building Blocks*
UMA	*Uniform Memory Access*
qNaN	*quiet NaN*
sNaN	*signaling NaN*

Списак слика

СПИСАК ТАБЕЛА

Списак кодова

A. Прилози

A.1. Програмски кодови разних елемената имплементације

```
float_components extractComponents(float f)
{
    float_components components;
    uint32_t tmp;
    static_assert(sizeof(uint32_t) == sizeof(float));
    memcpy(&tmp, &f, sizeof(float));
    components.negative = tmp & 0x80000000;
    components.exponent = (tmp >> 23) & 0xFF;
    components.mantissa = tmp & 0x7FFFFF;
    if (components.exponent != 0x00 && components.exponent != 0xFF) {
        components.mantissa |= 0x800000; // hidden bit is equal to 1
    }
    return components;
}
```

Код А.1. Имплементација помоћне функције extractComponents

```
float packToFloat(float_components components)
{
    uint32_t tmp = (components.negative ? 0x80000000 : 0x00000000) |
                   (components.exponent << 23) |
                   (components.mantissa & 0x7FFFFF);
    float f;
    static_assert(sizeof(uint32_t) == sizeof(float));
    memcpy(&f, &tmp, sizeof(uint32_t));
    return f;
}
```

Код А.2. Имплементација помоћне функције packToFloat

```
LongAccumulator &LongAccumulator::operator+=(float f)
{
    // Extract floating-point components - sign, exponent and mantissa
    float_components components = extractComponents(f);
    // Calculate the leftmost word that will be affected
    uint32_t k = (components.exponent + 22) >> 5;
    // Check if more than one word is affected (split mantissa)
    bool split = ((components.exponent - 1) >> 5) < k;
    // Calculate number of bits in the higher part of the mantissa
    uint32_t hi_width = (components.exponent - 9) & 0x1F;
    if (!split) {
        add(k, components.mantissa << (hi_width - 24), components.negative);
    } else {
        add(k - 1, components.mantissa << (8 + hi_width), components.negative);
        add(k, components.mantissa >> (24 - hi_width), components.negative);
    }
    return *this;
}
```

Код А.3. Имплементација оператора акумулирања: +=

```
float LongAccumulator::operator()()
{
    float_components components;
    components.negative = acc[ACC_SIZE - 1] & 0x80000000;
    components.exponent = 0;
    components.mantissa = 0;
    LongAccumulator absolute = components.negative ? -*this : *this;

    // Calculate the position of the most significant non-zero word...
    int word_idx = ACC_SIZE - 1;
    while (word_idx >= 0 && absolute.acc[word_idx] == 0) {
        word_idx--;
    }
    if (word_idx < 0) { // zero
        return packToFloat(components);
    }

    // Calculate the position of the most significant bit...
    bitset<32> bits(absolute.acc[word_idx]);
    int bit_idx = 31;
    while (bit_idx >= 0 && !bits.test(bit_idx)) {
        bit_idx--;
    }
    if (word_idx == 0 && bit_idx < 23) { // subnormal
        components.mantissa = absolute.acc[0];
        return packToFloat(components);
    }

    // Calculate the exponent using the inverse of the formula used
    // for "sliding" the mantissa into the accumulator.
    components.exponent = (word_idx << 5) + bit_idx - 22;
    if (components.exponent > 0xFE) { // infinity
        components.exponent = 0xFF;
        return packToFloat(components);
    }

    // Extract bits of mantissa from current word...
    uint32_t mask = ((uint64_t) 1 << (bit_idx + 1)) - 1;
    components.mantissa = absolute.acc[word_idx] & mask;

    // Extract mantissa and round according to currently selected
    // rounding mode...
    if (bit_idx > 23) {
        components.mantissa >>= bit_idx - 23;
        round(absolute, components, word_idx, bit_idx - 24);
    } else if (bit_idx == 23) {
        round(absolute, components, word_idx - 1, 31);
    } else {
        components.mantissa <<= 23 - bit_idx;
        components.mantissa |= absolute.acc[word_idx - 1] >> (bit_idx + 9);
        round(absolute, components, word_idx - 1, bit_idx + 8);
    }

    return packToFloat(components);
}
```

Код А.4. Имплементација оператора конверзије типа: ()

```
ostream &operator<<(ostream &out, const LongAccumulator &acc)
{
    bool sign = acc.acc[ACC_SIZE - 1] & 0x80000000;
    LongAccumulator positive_acc = sign ? -acc : acc;
    int startIdx = ACC_SIZE - 1, endIdx = 0;
    while (startIdx >= 0 && positive_acc.acc[startIdx] == 0) {
        startIdx--;
    }
    if (startIdx < 4) {
        startIdx = 4;
    }
    while (endIdx < ACC_SIZE && positive_acc.acc[endIdx] == 0) {
        endIdx++;
    }
    if (endIdx > 4) {
        endIdx = 4;
    }
    out << (sign ? "- " : "+ ");
    for (int idx = startIdx; idx >= endIdx; --idx) {
        bitset<32> bits(positive_acc.acc[idx]);
        int startBit = 31, endBit = 0;
        if (idx == startIdx) {
            while (startBit >= 0 && !bits.test(startBit)) {
                startBit--;
            }
        }
        if (idx == endIdx) {
            while (endBit < 32 && !bits.test(endBit)) {
                endBit++;
            }
        }
        if (idx == 4) {
            if (startBit < 21) {
                startBit = 21; // include first bit before .
            }
            if (endBit > 20) {
                endBit = 20; // include first bit after .
            }
            for (int i = startBit; i > 20; --i) {
                out << bits.test(i);
            }
            out << " . ";
            for (int i = 20; i >= endBit; --i) {
                out << bits.test(i);
            }
        } else {
            for (int i = startBit; i >= endBit; --i) {
                out << bits.test(i);
            }
        }
        if (idx > endIdx) {
            out << ' ';
        }
    }
    return out;
}
```

Код А.5. Имплементација оператора за штампу на излазни ток: <<