

TDK Dolgozat

Kelemen Zoltán

SIMD optimalizáció **alkalmazásterület-specifikus nyelvben**

Tdk dolgozat

Kelemen Zoltán
(2. évfolyam)

E-mail: kelemzol@elte.hu

Konzulens: Dévai Gergely
(Tanársegéd, PhD hallgató)

Programozási Nyelvek és Fordítóprogramok Tanszék, ELTE
Informatikai Kar



Támogatta a KMOP-1.1.2-08/1-2008-0002, Európai Regionális
Fejlesztési Alap (ERFA)

Tartalomjegyzék

1 Bevezetés.....	1
1.1 Feldspar.....	1
1.2 Technikai Háttér.....	2
1.2.1 Haskell.....	2
1.2.2 Feldspar.....	2
1.2.3 SIMD.....	3
1.3 Motiváció.....	4
1.4 SIMD Könyvtár.....	4
1.5 Példa: Skalárszorzás.....	5
1.6 Kapcsolódó Munkák.....	6
2 Fordítás.....	7
2.1 Fordítás menete.....	7
2.2 Mag nyelvi típus szerepe.....	11
2.3 Nyelvi rétegek.....	12
2.4 SIMD adatok megkülönböztetése.....	13
2.4 Átjárás a nyelvi rétegek között.....	15
2.5 Magnyelvi konstrukciók.....	16
3 SIMD könyvtár - Implementáció.....	17
3.1 Típusok.....	17
3.2 Új Feldspar Műveletek.....	18
3.3 Elemenkénti alaplóműveletek.....	20
3.4 Feltöltő, konstrukciós műveletek.....	22
3.4.1 Számliterálok túlterhelése.....	22
3.4.2 Közöséges feltöltő műveletek.....	23
3.4.3 Feltöltő operátorok.....	23
3.5 Indexelő műveletek.....	24
3.6 Összegző műveletek.....	26
4 Platform támogatás.....	27
4.1 Átírási szabályok.....	28
4.2 Rule plugin.....	29
4.3 Célplatform.....	31
5 Példa: Skalárszorzás.....	32
5.1 Feldspar interface.....	33
5.2 Technikai apróságok.....	33
5.3 Rule Plugin.....	34
5.4 Célplatform.....	35
5.5 Generált kódok.....	36
5.5.1 C platform tömb típusa.....	36
5.5.2 Generált kódok.....	37
5.6 Eredmények.....	40
6 Összefoglalás.....	42
7 Hivatkozások.....	43

1 Bevezetés

1.1 Feldspar

A Feldspar (Functional Embedded Language for Digital Signal Processing and Parallelism)[1] egy beágyazott, alkalmazásterület-specifikus funkcionális programozási nyelv, melyet digitális jelfeldolgozó algoritmusok megfelelően absztrakt leírására fejlesztettek ki.

A digitális jelfeldolgozás egy kritikus pontja a hatékonyság, ezért rendszerint az algoritmusokat C nyelven implementálják, egy adott platform speciális tulajdonságainak kihasználásával. Ennek hátránya, hogy az alacsony nyelvi szint miatt az algoritmusokat nehéz implementálni, nehéz karbantartani, az így keletkező C kódok csak kevéssé, vagy egyáltalán nem lesznek hordozhatóak a platformspecifikus típusok és függvények használata miatt.

A Feldspar, egy magas szintű, hardverfüggetlen programozási nyelv. Olyan eszközt ad a programozó kezébe, mellyel egy algoritmust csupán egyszer kell implementálnia a programozónak, és azt tetszőleges, a Feldspar által is támogatott platformra lefordíthatja. Ilyen támogatott platform például a C99 nyelv is.

Egy moduláris nyelvről van szó, melyet könnyedén ki lehet egészíteni új magas szintű könyvtárakkal, - amennyiben valamilyen speciális funkcióra lenne szüksége a programozónak - oly módon, hogy a nyelv vagy a fordítóprogram elemeit nem kell módosítanunk.

A Feldspar nyelvet és fordítóprogramját az ELTE Informatikai Kar Programozási Nyelvek és Fordítóprogramok Tanszékének Feldspar kutatócsoportja, a svédországi Ericsson és a göteborgi Chalmers egyetem kutatócsoportja közösen fejleszti.

1.2 Technikai Háttér

1.2.1 Haskell

A Haskell egy tisztán funkcionális, általános célú programozási nyelv, melyet kifejezőereje miatt gyakorta használnak beágyazott nyelvek fejlesztésére, gazda nyelvként. A Feldspar is egy Haskellbe ágyazott programozási nyelv. [5]

Beágyazott nyelvnek nevezzük azokat a programozási nyelveket, melyek egy másik nyelv szintaktikáját, és típusrendszerét kölcsönzik, azonban szemantikájukat némiképp módosítják. Ezt úgy valósítják meg, hogy az eredeti nyelven egy célspecifikus eszközkészletet fejlesztenek. Ezt az eszközkészletet használva érezheti a programozó úgy, hogy egy új nyelvvel van dolga.

1.2.2 Feldspar

A Feldsparnak, mint alkalmazásterület specifikus beágyazott nyelvnek a célja, hogy olyan felületet biztosítson a programozó számára, melyen megfelelően absztrakt módon írhat jelfeldolgozó algoritmusokat úgy, hogy azokból a fordító optimális kódot legyen képes generálni.

A Feldspar már számos magas szintű könyvtárral rendelkezik. Ezek közül néhány:

- A Vector könyvtár, a Haskell lista függvényeinek kényelmét hivatott biztosítani.
- A Matrix könyvtár, a Vector könyvtárból kiindulva nyújt támogatást két dimenziós mátrixok kezeléséhez.
- A BitVector könyvtár, olyan adatszerkezetet definiál, melyben minden bit különálló adatot képvisel. Ezen az adatszerkezeten végezhetőek logikai műveletek, melyek minden bitre külön hajtódhatnak végre.
- Stream könyvtár, egy olyan szekvenciális adattípust ad a programozó kezébe, ahol egy kezdő állapotot és egy állapotból egy olyan állapotban képező függvényt felhasználva, ahol a pár

első eleme egy új állapot a második pedig szekvenciális adatszerkezet egy eleme, folyamatosan iterál új állapotot és eredményt. Ez egy végtelen adatszerkezet.

- Seq könyvtár, a Stream könyvtár egy olyan speciális változata, mely véges.
- FixedPoint könyvtár, egy absztrakt lebegőpontos típust definiál, olyan platformok számára, ahol az ilyen típus nem támogatott.

1.2.3 SIMD

A SIMD a Single Instructions Multiple Data szavak rövidítése. Az adatfolyam olyan feldolgozásáról van szó, ahol egyetlen utasítás egyszerre több azonos típusú adaton hajtódik végre. Ennek a gyakorlati jelentősége a hatékonyságban keresendő. Olyan hardveres platformok, amelyek támogatnak SIMD műveleteket, azonos, vagy közel azonos idő alatt képesek bizonyos műveleteket végrehajtani egy illetve több adaton. SIMD intrinsic-eknek nevezzük azokat az alacsony szintű utasításokat, melyek közvetlenül egy processzor utasítást takarnak. SIMD műveletek használatakor jellemzően kis adatcsomagokra osztják fel az adatfolyamot (például 4, vagy 8 elem alkot egy csomagot), és ezeken egyenként hajtják végre a kívánt műveleteket. Egy algoritmus SIMD műveletekkel történő gyorsításakor az egyetlen nehézséget az adatfolyam megszervezése jelentheti [2].

A Feldspar programozási nyelvet kiegészítettük egy az előzőekhez hasonló könyvtárral, mely segítségével magas absztrakciós szinten használhat a programozó SIMD műveleteket. Ezzel az eszközzel egy hordozható algoritmus is tartalmazhat olyan erős optimalizációkat, mint egy platform specifikus kód.

A Feldspar jelenlegi stabil változatában (0.5.0.1), és az általam használt változatában (0.4.0.2), a C kód generálása támogatott. SIMD optimalizációra itt szükség lehet. Léteznek olyan platformok (pl. Intel x86), melyek alacsony szinten támogatnak SIMD művelet processzor szintű utasításokon keresztül. Az LLVM egy platformfüggetlen assembly nyelv, melynek a SIMD utasítások is szerves részét képezik [3].

1.3 Motiváció

A dolgozat célja, hogy egy olyan eszközkészlettel egészítse ki a Feldspar nyelvet, mely segítségével a programozó a nyelvre jellemző magas absztrakciós szint és áttekinthetőség megtartásával képes legyen SIMD optimalizálásra.

- Az eszközkészlet fejlesztése ne igényeljen nagyobb strukturális változtatásokat a Feldspar nyelvben, lehetőleg csak kiegészítésről legyen szó.
- Adjon jó eszközöket a programozó kezébe a SIMD optimalizáció minél széleskörűbb felhasználásához.
- Felhasználásával generált kódok legyenek hatékonyabbak az annak használatát mellőző verzióitól. Ez az hatékonyságbeli különbség legyen az ezekre kritikus kódszakaszokban jól mérető és kimutatható.

1.4 SIMD Könyvtár

A létrehozott SIMD könyvtár bevezet olyan típusokat melyek képesek tárolni kettő, négy illetve nyolc integer elemet, így alkotva egy SIMD vektort. Ezekhez a típusokhoz kínál műveleteket is, melyek úgy működnek, hogy a csomag minden elemén egymástól függetlenül végrehajthatódnak.

- Ilyen adatcsomagok építésére többféle konstrukciós lehetőség létezik.
- A csomagokra értelmezve van a páronkénti összeadás, szorzás, kivonás, abszolút érték, előjel függvény és negálás műveletek.
- Valamint értelmezve van egy olyan összegzés művelet, mely egy csomag elemeit adja össze.
- A csomagokat lehet indexelni.

A Feldspar Vector könyvtárának műveletei csomagokat tartalmazó vektorokra is megfelelően működnek.

1.5 Példa: Skalárszorzás

A dolgozatban be fogom mutatni egy példaprogram segítségével, hogyan megy végbe egy SIMD optimalizációval támogatott kód fordítása. Ez a példaprogram a skalárszorzás lesz.

Szerepeljen itt egy absztrakt leírása a skalárszorzásnak, Haskell nyelven. Ez jó referencia lesz a későbbiekben is:

```
scalarProd :: (Num a) => [a] -> [a] -> a
scalarProd a b = sum (zipWith (*) a b)
```

A lista elemeitől megköveteljük, hogy numerikus típusúak legyenek, hisz csak így végezhetünk rajtuk összeadás és szorzás műveleteket.

A `zipWith` függvény egy két operandusú függvényt, és két listát vár, majd a függvény alkalmazza a lista azonos indexű elemeiből képzett párokra, így képezve egy harmadik listát, mely az eredmény lesz.

A `sum` függvény egy vektort vár, melynek elemeire értelmezve vannak az alapvető numerikus műveletek, majd ezt a listát összegzi.

Ezek a függvények `Feldspar` változatai megtalálhatóak a `Vector` könyvtárban is. Ezek felhasználásával a skalárszorzás `Feldspar` implementációja:

```
scalarProd :: Numeric a =>
    Vector (Data a) -> Vector (Data a) -> Data a
scalarProd a b = sum (zipWith (*) a b)
```

Látható, hogy a két implementáció szinte teljesen azonos, csupán a függvények szignatúrájában látható eltérés. Mégis a `Feldspar` `Vector` könyvtárával implementált változatot lefordítva, hatékonyabb kód keletkezik, mint a Haskell-es változat esetén. Ennek oka, hogy a `Feldspar` kódot lefordíthatjuk pl. szabványos C99 nyelvre is, ami egy rendkívül tömör, és gyors kódot

eredményez.

Azonban ennél is hatékonyabb kódot kaphatunk, ha nem csak szabványos, hanem platformfüggő elemeket is használunk. Ilyen platformfüggő elemek lehetnek a speciális SIMD intrinsic-ek.

A dolgozatom célja, hogy az eddig látott magas absztrakciós szinten megadott kódon ne, vagy csak keveset változtatva használhasson a programozó platformfüggő SIMD intrinsic-eket. Például a következő, képen:

```
scalarProd :: Numeric a =>
    Vector (SIMD a) -> Vector (SIMD a) -> Data a
scalarProd a b = ssum (sum (zipWith (*) a b))
```

A SIMD a egy olyan típus, mely több a típusú adatot tárol, a fordító SIMD vektorként kezeli, és platformspecifikus kódot generál a segítségével.

A ssum függvény összegzi az egy SIMD vektorban tárolt elemeket.

1.6 Kapcsolódó Munkák

A MOUSE [4] Projekt keretein belül egy olyan eszközt fejlesztettek, mely Matlab forrásokból képes előállítani alacsony szintű C illetve assembly kódokat DSP célokra, SIMD támogatással.

Christopher Kumar Anand és Wolfram Kahl [5] egy Haskellbe ágyazott nyelvet fejlesztettek matematikai számításokhoz és nagy teljesítményű SIMD-párhuzamos algoritmusokhoz.

2 Fordítás

2.1 Fordítás menete

Egy Feldspar program fordítása, a következő lépésekből áll: A Feldspar, egy mélyen beágyazott nyelv tehát első lépésben a programozó, a Feldspar interface-n keresztül lényegében, egy Haskell adattípust épít. Ezt hívjuk magnyelvnek. Az adattípus neve `Data a`, ahol `a` a típusparaméter. A fordítás tehát, első lépésben egy adatszerkezetet hoz létre. Egy megfelelően absztrakt leírását, egy számításnak. Az `a` típusparaméter, a számítás eredményének típusát jelöli.

Például jelölje `x` a következő számítást:

```
scalarProd (vector [1,2]) (vector [3,4]) :: Data Int32
```

Ahol `vector` egy olyan függvény, mely listákból Feldspar `Vector`-okat épít.

Egyik módja egy `Data` számítás kiértékelésének, az `eval` függvény használata.

```
$ ghci
> :m Feldspar
...
> let x = scalarProd (vector [1,2])
              (vector [3,4]) :: Data Int32
> :t eval
eval :: Data a -> a

> eval x
11
```

Egy másik mód, hogy `x`-ből a fordítóprogram segítségével kódot generálunk.

Data a konstrukciói közé, többek között olyan fogalmak sorolhatóak, mint feltételes érték kiválasztás, szekvencia, tömb indexelés valamint kétféle ciklus. Az egyik egy vektor elemein végighaladva egyenként, a vektor elemeit egymástól független módon transzformálja. (Ily módon, ezt a ciklust olyan esetben amikor igény támad rá, könnyű párhuzamosítani.) A másik ciklus a tipikusabb, állapottal dolgozó verzió, ami azt jelenti, hogy a vektor elemeinek kiszámítása, nem feltétlen független egymástól, egy számítás hivatkozhat egy korábbi iteráció eredményére. Data a konstrukció még, az elemi függvény hívás. Ezek olyan függvények vagy primitívek, melyek már a fordítás végén is függvényhívásként vagy primitívként jelennek meg.

A fő fordítási folyamat az erősen funkcionális jellemzőket is tartalmazó, magnyelvi reprezentációról egy platformfüggetlen absztrakt imperatív reprezentációra transzformálja a programot. Ez az imperatív reprezentáció megfelel, egy nem túl bonyolult imperatív nyelv szintaxisfájának. [6]

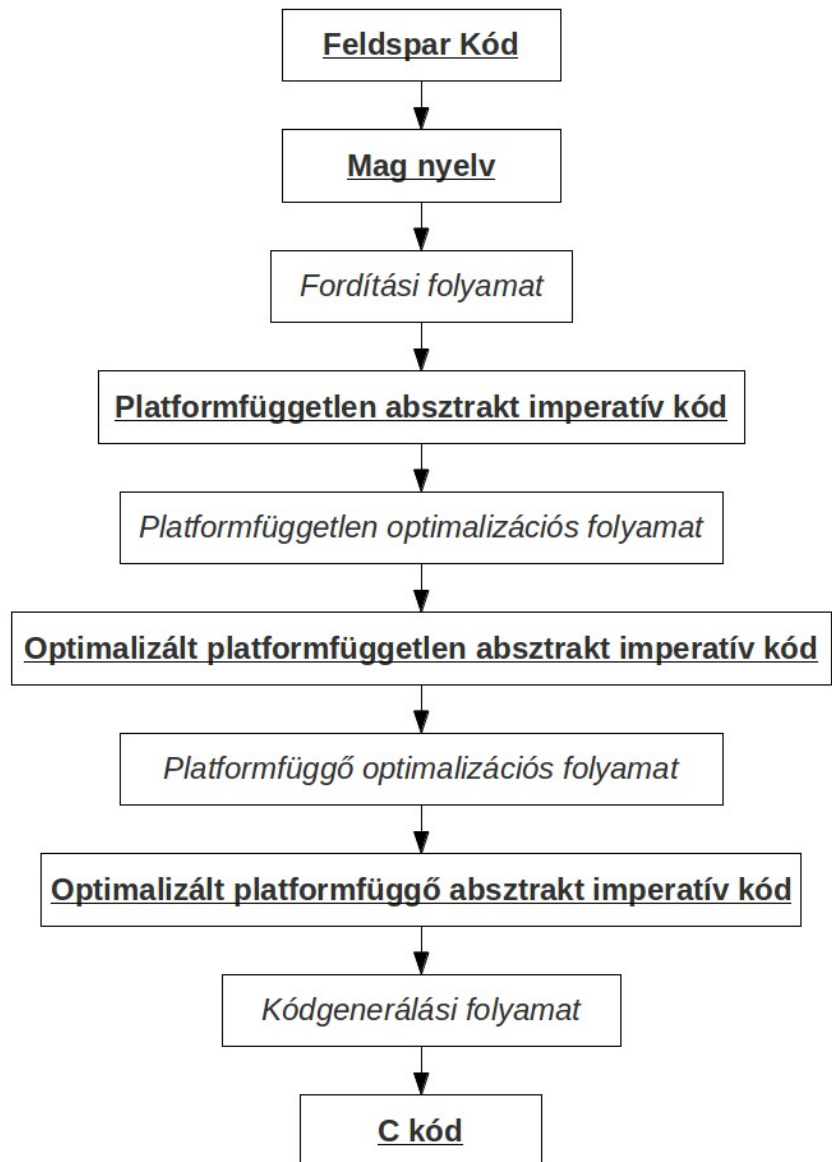
Az absztrakt imperatív reprezentáció elemeit már nem tartalmaz típusparamétereket, így a Haskell fordító nem tudja teljes körűen garantálni a reprezentáció helyességét. Így kézzel könnyedén leírhatunk benne olyan kifejezéseket, programokat, amik a mi szemantikánk szerint hibásak. Az, hogy ilyen adatszerkezetek mégsem generálódnak, a Feldspar magnyelvének, és interface-ének köszönhetően. Ezen két réteg erős típusossága garantálja a szemantikus helyes adatszerkezetek generálását.

Természetesen ez a reprezentáció sem tartalmaz SIMD specifikus típusokat, tehát nem találkozunk pl. négy darab Int16-ot tartalmazó SIMD vektorral. Ahogyan a magnyelvi reprezentációban, itt is más típusokba csomagolva tároljuk a SIMD vektorokat. Tehát csupán a típusokat tekintve, ezen a ponton már nem különböztethetők meg az egyszerű adatoktól. Többek között ennek a problémának a feloldására szolgálnak a bevezetett új Feldspar műveletek is. Ha ezt az adatszerkezetet vizsgálva olyan műveletekbe botlunk, ami SIMD specifikus, és SIMD típusú adatokat vár paraméterként, vagy azokat ad vissza, nem okozhat meglepetést, ha a bemeneti paraméter, vagy épp a visszakapott érték, az adatszerkezet szerint, nem SIMD vektor, hanem például nyolc bites integer.

Az így kapott reprezentációt lehetőség van platformfüggetlen, és platformfüggő optimalizációknak alávetni. Erre a célra az úgy nevezett Rule Plugin használható. Ez egy hatékony eszköz az imperatív reprezentáció manipulálására. A Rule Plugin a Feldspar fordító, transzformációs keretrendszerének része [6]. A Rule Plugin a programozó által megadott szabályok szerint cserél le program részleteket. A szabályok felírása intuitív módon történik.

A következő, kódgenerálási fázisban a platformfüggő elemeket is tartalmazható imperatív reprezentációt a fordító átalakítja platformfüggő kóddá. Például a pretty printer segítségével, ansi C kódot állít elő.

A fordítási folyamatot jól szemlélteti az *Ábra 1*.



Ábra 1

2.2 Mag nyelvi típus szerepe

A magnyelvi típusról, már volt szó, mint egy számításról. Azonban ezen felül, még sok jelentést, vagy szerepet tölthet be.

A Vektor könyvtárban, egy vektor (`Vector (Data a)`) magnyelvi reprezentációja `Data [a]`.

Az én dolgozatomban, a `Data a` egy olyan típus szerepét tölti be, mely egyfajta átjárást képez a nyelvi rétegek között. Egészen pontosan, a `Feldspar` magnyelve (és ezen keresztül a `Vector` könyvtár), és a `SIMD` könyvtár között. Ezen átjárhatóságot úgy valósítja meg, hogy a `SIMD` könyvtárban használt, `SIMD` utasításokhoz szánt típusokat külön-külön leképezzük egy magnyelvi típusra. Lássunk pár példát ilyen leképezésekre:

<code>Int16</code> , két elemet tartalmazó <code>SIMD</code> vektorra	<code>-></code> <code>Int32</code>
<code>Int16</code> , négy elemet tartalmazó <code>SIMD</code> vektorra	<code>-></code> <code>Int64</code>
<code>Int32</code> , két elemet tartalmazó <code>SIMD</code> vektorra	<code>-></code> <code>Int64</code>
<code>stb...</code>	

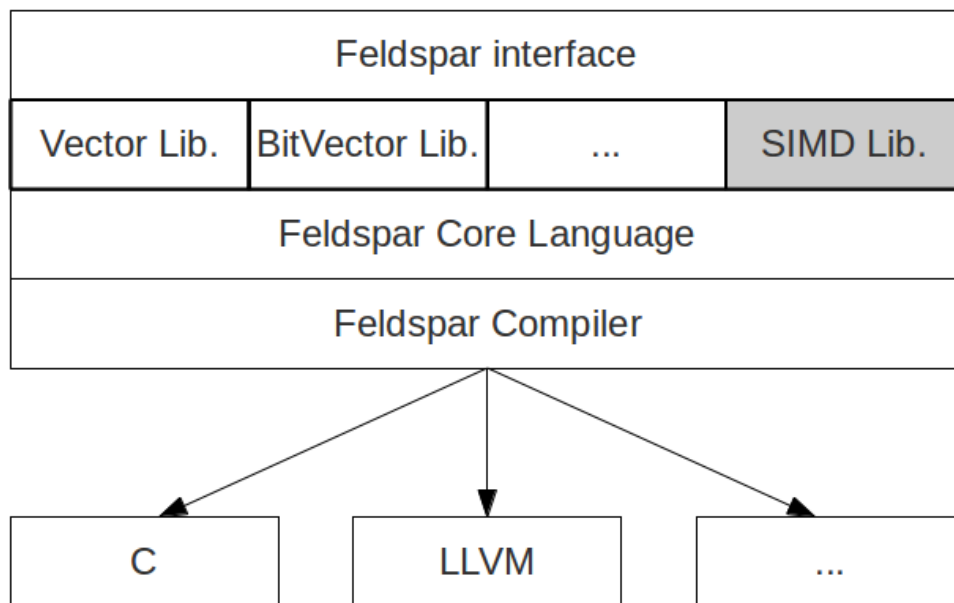
Ez persze önmagában nem lenne elég a korrekt működéshez, ezért a `SIMD` könyvtárban új típusokat is definiálunk (technikailag a magnyelvi megfelelőket csomagoljuk (wrap-eljük) be), majd ezekhez vezetünk be új műveleteket. Így a fordítás későbbi szakaszaiban, ahol szükség lenne az olyan információkra, mint hogy melyik adatot kell `SIMD`-ként kezelni, ott az adaton végzett művelet ezt elárulja.

A fenti példán az is jól látható, hogy nem feltétlen az adattípus szerkezeti hasonlósága alapján választunk magnyelvi megfelelőt, hanem méreti illeszkedés alapján.

2.3 Nyelvi rétegek

Feldspar interface-nek nevezzük azoknak a típusoknak, és függvényeknek a halmazát, melyeket a Feldspar magjának kívülről látható része, és az a fölött definiált könyvtárak (Vector, BitVector, Matrix stb...) tartalmaznak. Ezt az inerface-t egészítettük mi is ki, a SIMD könyvtárral.

Az eddigiek alapján, a következőképpen különíthetők el a nyelvi rétegek. A Legfelső szinten maga a Feldspar interface áll ami a magas szintű könyvtárakból, és magnyelv látható részéből építkezik. Ez alatt található a magnyelv, majd a fordító.



Ábra 2

2.4 SIMD adatok megkülönböztetése

Az egyszerű adatok és SIMD adatok közötti különbségek, a fordítás legnagyobb része alatt rejtve maradnak, és így a fordítási folyamatot semmilyen tekintetben nem kellett módosítanunk vagy kiegészítenünk. Ez pontosan annak köszönhető, hogy már létező magnyelvi típusokat használunk, új „típusú” adatok tárolására.

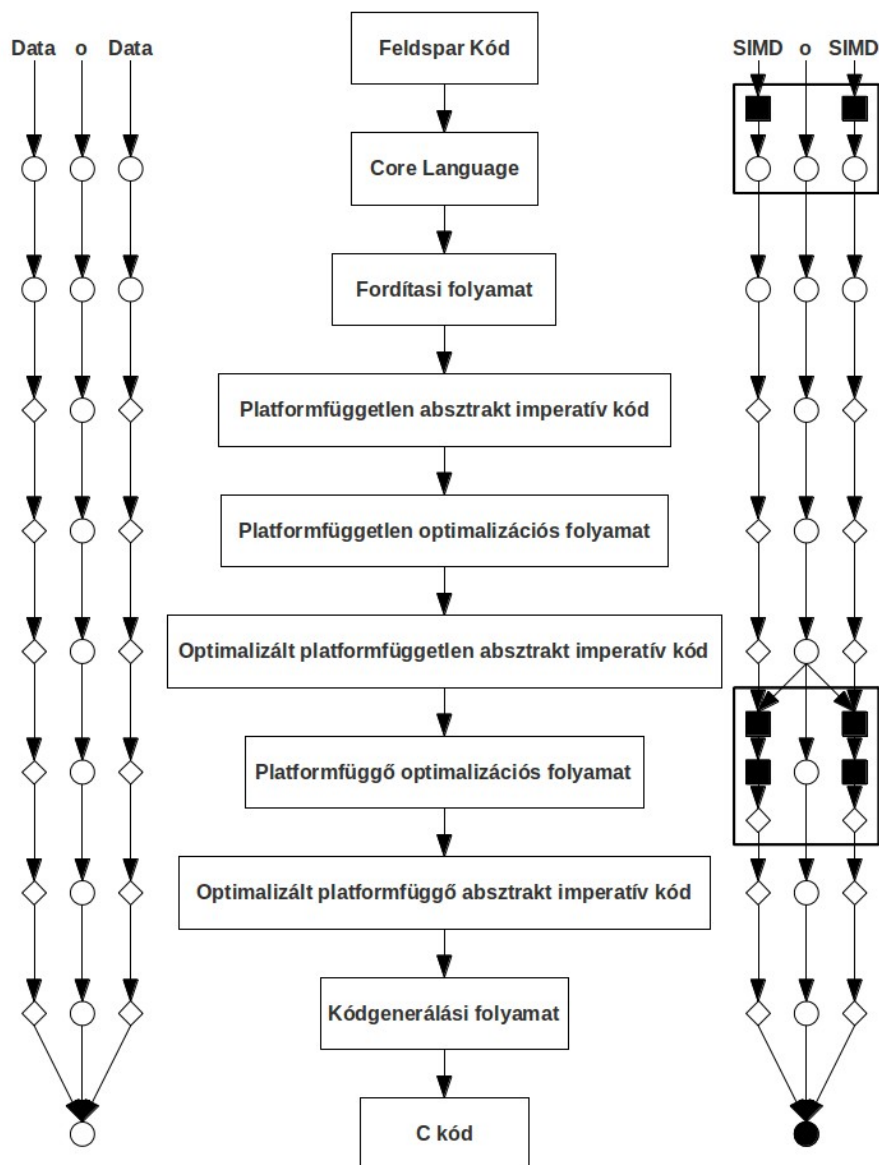
Összesen három olyan pont található, ahol különbséget teszünk egyszerű, és SIMD adat között. A legfelső réteg, azaz a Feldspar interface az első. Itt nyilvánvalóan szükség van erre, hogy a programozó kifejezhesse szándékát a SIMD optimalizációra.

Megkülönböztetjük még a SIMD adatokat, a platformfüggő optimalizációs eljárásban is. Ebben az esetben nem csupán optimalizációról van szó, hanem olyan szerkezeti átalakításokról is, amik elősegítik a platform megfelelő támogatását.

A célplatformon is szükség van megkülönböztetésre, bár ez a fázis inkább csak arra szolgál, hogy a platformtámogatás teljes legyen.

Feldspar Kód szintjén a különbözőség egyértelműen látszik, a maradék két fordítási fázisban, azonban mind két helyen, a rajtuk végzett műveletek segítenek.

Ábra 3 ábrán nyomon követhető, mely helyeken milyen módon különböztetjük meg a SIMD adatokat.



Ábra 3

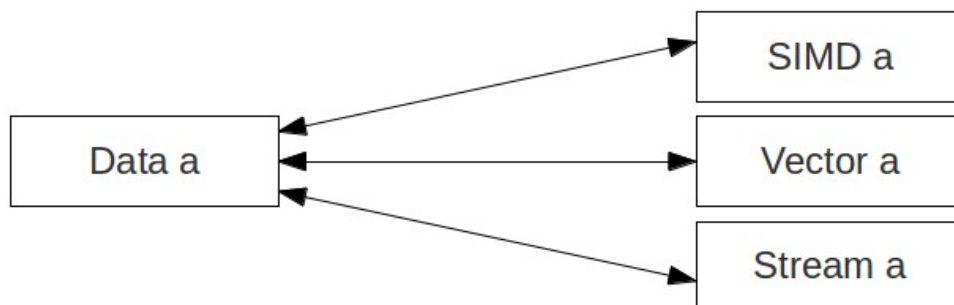
A képen a fordítási folyamat látható és az ahogyan a folyamat alatt a fordító hogyan kezeli az adatokat.

Egy példa látható, ahogyan két egyszerű Data a és két SIMD a között valamilyen operátorral műveletvégzés történik. Egyszerű

adatok közötti művelet fordításánál az látható, hogy a Feldspar interface-en a programozó leírja a műveletet, és ez annak megfelelő módon halad a fordítási folyamaton végig a kódgenerálási lépésig. A platformfüggetlen absztrakt imperatív kódban, már máshogy jelennek meg az adatok(ezeket jelöljük rombuszsal). SIMD adatok közötti művelet fordításánál a Feldspar interface-en SIMD adatokként tartja számon a programozó(SIMD adatokat jelöljük telt négyzettel), azonban ezek azonnal magnyelvi típusra alakulnak a fordítás kezdetén. A platformfüggő optimalizációs folyamatban a rajtuk elvégzett műveletből következtetve visszanyerik SIMD tulajdonságaikat, és akként is működnek, míg ez a folyamat véget nem ér, ahol ismét egyszerű típusokként vannak jelen az absztrakt imperatív kódban. Kódgenerálás után pedig platformfüggő módon kezelendők a SIMD sajátosságai.

2.4 Átjárás a nyelvi rétegek között

Általánosságban elmondható, hogy bármely magas szintű könyvtár típusai, leképezhetők magnyelvi típusra.



Ábra 4

Ezek olyan művelet párosok segítségével hajtódnak végre, melyeket konvencionálisan freeze-unfreeze műveleteknek nevezünk. Egy freeze művelet, egy magas szintű könyvtári elemet leképez egy magnyelvi típusra. Az unfreeze ennek ellenkezőjét csinálja.

2.5 Magnyelvi konstrukciók

Érdemes lehet megemlíteni, hogy a Feldspar magnyelve, nem csak egyszerű típusú adatok, hanem bonyolult számítások tárolására is képes.

Az alábbi konstrukciókból állítható össze egy magnyelvi elem:

```
Literal :: (...) => a -> Data a

Function :: (...) => String -> (a -> b) ->
               Data (a -> b)

Pair :: (...) => Data (a -> b -> (a,b))

Condition :: (...) => Data (Bool -> a -> a -> a)

Parallel :: (...) =>
               Data (Length -> (Index -> a) ->
               [a] -> [a])

Sequential :: (...) => Data (Length -> st ->
               (Index -> st -> (a,st)) ->
               (st -> [a]) -> [a])

ForLoop :: (...) =>
               Data (Length -> st ->
               (Index -> st -> st) -> st)

NoInline :: (...) => Data ((a -> b) -> (a -> b))

SetLength :: (...) => Data (Length -> [a] -> [a])

SetIx :: (...) => Data (Index -> a -> [a] -> [a])
```

3 SIMD könyvtár - Implementáció

Szükségünk volt olyan típusokra, melyekkel közel úgy dolgozhatunk, mint az általános Data típusokkal. Szükségünk volt olyan típusokra és általános típus hozzárendelési módszerre, mely elárulja, hogy a használt típusoknak, milyen mag nyelvi megfelelője van.

Definiálnunk kellett olyan műveleteket, melyeket általános Data típusú elemeken is végrehajthatóak. Ezeknek a műveleteknek a bonyolultsága azonban nem lehetett magas, hiszen a SIMD műveletek jellemzően egyszerűek.

Létrehoztunk még olyan kényelmi műveleteket is, melyekkel egyszerűen építhetőek, indexelhetőek, összegezhetőek SIMD vektorok.

3.1 Típusok

A Feldspart elsőként olyan típusosztályokkal egészítettük ki, melyek elárulják, mely típusokat lehet összecsomagolni egy vektorszerű adatszerkezetbe, és olyan jellemzőket rendeltünk ezen osztályok elemeihez, melyekből kiderül, hogy az összecsomagolt elemnek, milyen reprezentációja lesz a Feldspar magnyelvében.

Ilyen osztályok a Packable2 a, Packable4 a, Packable8 a, ahol a típusparaméter, a csomagolni kívánt típust jelöli, és ilyen jellemzők a Pack2 a, Pack4 a, Pack8 a. Ezek az osztályok azt is leírják, hogy mely típusok hányasával csomagolhatóak. Az alábbi kódrészletben látható példaképp a Packable4 osztály definíciója, majd a típus osztály egy példányosítása Int16 típusra.

```
class (...) => Packable4 a where  
  type Pack4 a
```

```
instance Packable4 Int16 where  
  type Pack4 Int16 = Int64
```

Négy darab Int16-ot tartalmazó SIMD vektor magnyelvi

reprezentációjaként, az Int64-t választottuk, hiszen ezek méreti megegyeznek, és minden platformon egy helyen tárolódnak a memóriában.

Ezt követően, új típusokat vezettünk be, melyek már magukat az összecsomagolt elemeket tárolják.

Példaképp a négy összecsomagolt elem definíciója:

```
data (Packable4 a) => SIMD4 a = SIMD4 (Data (Pack4 a))
```

SIMD4 a típusnak csak úgy értelmezhető, ha a típusparaméter négyesével csomagolható, így ezt megköveteljük.

Szükség van olyan függvényre, melyek SIMD típusú adatból, magnyelvi típusú adatot állít elő, és olyanra is, ami ezt visszafelé végzi el. Legyenek ezek a függvények a freezeSIMD4 és unfreezeSIMD4:

```
freezeSIMD4 :: (Packable4 a) =>
    SIMD4 a -> Data (Pack4 a)
freezeSIMD4 (SIMD4 a) = a
unfreezeSIMD4 :: (Packable4 a) =>
    Data (Pack4 a) -> SIMD4 a
unfreezeSIMD4 = SIMD4
```

Az ilyen típusú függvények biztosítják az átjárást, a SIMD könyvtár, és a magnyelv között.

Ugyanilyen freeze-unfreeze műveleteket létrehoztunk az összes SIMD típusra.

3.2 Új Feldspar Műveletek

Egy olyan magas szintű könyvtár esetében mint Pl. a Vector könyvtár, nincs szükség új Feldspar műveletek bevezetésére, hiszen csak a Feldspar tömbjeit kívánjuk olyan magas absztrakciós szintre emelni, mint amilyen például a Haskell lista típusa.

Azonban a SIMD könyvtárnál alapvető követelmény, hogy új Feldspar műveleteket definiáljunk, hiszen pontosan arra van szükség, hogy bizonyos műveleteket, más módon, más függvényhívásokkal vagy műveletekkel generáljunk. A SIMD

könyvtár tehát számos új Feldspar műveletet vezet be.

Új Feldspar műveletek bevezetésére több függvény is rendelkezésünkre állt. Ezek a függvények a Feldspar magnyelvi moduljában találhatóak.

Ezek közül a legegyszerűbb a `function1`:

```
function1 :: (Type a, Type b) => String ->
           (Size a -> Size b) ->
           (a -> b) -> (Data a -> Data b)
function1 fun sizeProp =
           function True fun (sizeProp . EdgeSize)
```

Paraméterül vár egy `String`et, ez lesz az új művelet neve, egy `(Size a -> Size b)` alakú függvényt, ezt egy méret kiköveztető algoritmus használja fel fordítás alatt, majd `(a -> b)` alakú függvényt, amit a kiértékelő rendszer használ. A függvény vissza ad egy `(Data a -> Data b)` alakú függvényt melyet aztán a programozó, új Feldspar műveletként használhat.

Egy egyszerű példa új műveletek definiálására:

```
id :: Data Int32 -> Data Int32
id a = (function1 "id" id id) a
```

`Int32` típusú adatok identitás függvénye. Itt a `function1` paramétereiként kapott `id` függvények, természetesen itt a Haskell alapsomagjának a `Prelude`-nak az `id` függvényei.

Minden új Feldspar művelet definiálásához azonban nem alkalmas a `function1` függvény, hiszen csak `(Data a -> Data b)` alakú (azaz csupán egy paraméteres) függvényeket képes definiálni. Rendelkezésünkre állt azonban több a `function1` függvényhez hasonló művelet definiáló eszköz is. Ezek a `function2`, `function3`, ..., `function8`. Ahogy a nevük is mutatja, rendre 2,3,...,8 paraméteres műveletek definiálására alkalmasak.

```

function1 :: ... => (Data a -> Data b)
function2 :: ... => (Data a -> Data b -> Data c)
function3 :: ... => (Data a -> Data b -> Data c
                    -> Data d)
...
function8 :: ... => (Data a -> Data b -> Data c
                    -> Data d -> Data e -> Data f
                    -> Data g -> Data h -> Data i)

```

Látható, hogy az összes ilyen művelet definiáló eszköz egy speciális tulajdonsága, hogy csakis `Data` típusú elemeket fogad paramétereiként. Ahhoz, hogy egy `SIMD` típusú elemet paraméterül lehessen adni egy ilyen új műveletnek, először magnyelvi reprezentációjára kell fordítanunk. Itt válik fontossá a magnyelvi átjárhatóság, mely a már említett `freeze-unfreeze` műveletek segítségével, rendkívül könnyedén és akadálymentesen végezhető.

Egyszerű példa, immáron `SIMD` típusokkal:

```

id :: SIMD Int32 -> SIMD Int32
id a = unfreezeSIMD4 $
      (function1 "id" id id) (freezeSIMD4 a)

```

3.3 Elemenkénti alpműveletek

Ezekhez az új típusokhoz, értelmeznünk kellett a szokásos műveleteket. A legáltalánosabb műveleteket (mint például két csomag összeadása – ami elemenkénti összeadást jelent), a Haskell `Num` osztályában találjuk, így ennek példányosítását elvégeztük.

Haskell `Num` osztály definíciója:

```

class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a

```

```
fromInteger :: Integer -> a
```

Ennek példa példányosítása SIMD4-re:

```
instance (Packable4 a, Type (Pack4 a),
         Numeric (Pack4 a))
  => Num (SIMD4 a) where
  (+) = fSIMD4_Op "SIMD4_Add" (+)
  (*) = fSIMD4_Op "SIMD4_Mul" (*)
  (-) = fSIMD4_Op "SIMD4_Sub" (-)
  abs = fSIMD4_Un "SIMD4_Abs" abs
  signum = fSIMD4_Un "SIMD4_Sgn" signum
  negate = fSIMD4_Un "SIMD4_Neg" negate
  fromInteger a = unfreezeSIMD4 $
    (FB.shiftLU (FB.shiftLU (FB.shiftLU
      (cast a) s + (cast a)) s +
      (cast a)) s + (cast a))
```

Ahol `fSIMD4_Op` függvény, új két paraméteres Feldspar műveleteket (pontosabban operátorokat) definiál egy `String`, és egy operátor segítségével. A `String` az új Feldspar művelet neve, míg az operátor egy kiértékelő rendszernek nyújt segítséget. Az `fSIMD4_Un` függvény pedig egy egyparaméteres Feldspar műveleteket definiál. Ezek a műveletek egy csomagon belül, minden elemre egyenként végrehajtandó utasításokat jelentenek.

`fSIMD4_Op` és `fSIMD4_Un` függvények definíciói:

```
fSIMD4_Op :: (Packable4 a, Type (Pack4 a)) =>
  String -> (a -> a -> a) -> SIMD4 a ->
  SIMD4 a -> SIMD4 a
fSIMD4_Op op f (SIMD4 a) (SIMD4 b) = SIMD4 $
  (function2 op sizePropagation
    evalFunction) a b

fSIMD4_Un :: (Packable4 a, Type (Pack4 a)) =>
  String -> (a -> a) -> SIMD4 a -> SIMD4 a
fSIMD4_Un un f (SIMD4 a) = SIMD4 $
  (function1 un sizePropagation evalFunction) a
```

Ahogy azt már szóba került, `function1` és `function2` művelet definiáló eszköz egyik tulajdonsága, hogy paramétereiben, és eredményében is, magnyelvi típusokat vár, illetve add vissza. Hogy egy csomagot paraméterül tudjunk adni, előbb magnyelvi reprezentációra kell alakítanunk. Ezt végzi a `freezeSIMD4` művelet. Ennek párja, az `unfreezeSIMD4`, melyet most nem használtunk, hiszen a feladatnak megfelelően most `Data`, azaz mag nyelvi típust adunk vissza.

3.4 Feltöltő, konstrukciós műveletek

3.4.1 Számliterálok túlterhelése

A `fromInteger` művelet, külön tárgyalást igényel. A `fromInteger` szemantikája szerint egy egyparaméteres függvény, mely egy csomag minden elemének egy Haskell Integer értéket ad értékül. Ily módon, ez egy feltöltő művelet.

A `fromInteger` más szempontból is speciális művelet. Ez egy olyan függvény, ami Haskell szabvány szerint olyan esetekben, ahol a fordító egy számliterállal találkozik és a típuskikövetkeztetés szerint olyan típusú elemnek kell a számliterál helyén állnia melyre példányosítva van a `Num` osztály, a számliterált a fordító `Integer` típusú értéként kezeli és kicseréli egy olyan `fromInteger` függvényhívásra, melynek argumentuma a számliterál. Például a következő esetben látható ennek haszna:

```
f :: SIMD4 Int32
f = 10
```

Ebben az esetben a fordító a 10 literált kicseréli erre a kifejezésre:

```
fromInteger 10
```

Ahol a `fromInteger` a `SIMD4 Int32` típushoz tartozó metódusa.

Ezzel egyfajta túlterhelést biztosít a számliterálokra.

A Haskell `Integer` típusa, tetszőleges méretű egész számot képes tárolni. Azonban ez megadott méretű csomagoknál érthető módon, nem lehetséges. Ennek kezelése a programozó feladata. Az ok amiért ezt mégis célszerű használni az imént említett számliterálok kényelmes használati lehetősége. Ezzel könnyítheti a programozó munkáját, és olvashatóbbá teheti a program kódját.

3.4.2 Közösítéses feltöltő műveletek

Más feltöltő műveleteket is definiáltunk. Esetenként igény támadhat, olyan csomagokat létrehozni, melynek minden elemét egyenként be lehet állítani. Ilyen műveletek a `simd2` `simd4` `simd8`.

```
simd2 :: (Packable2 a) => Data a -> Data a -> SIMD2 a
simd4 :: (Packable4 a) => Data a -> Data a -> Data a ->
    Data a -> SIMD4 a
simd8 :: (Packable8 a) => Data a -> Data a -> Data a ->
    Data a -> Data a -> Data a -> Data a ->
    Data a -> SIMD8 a
```

Ezek is, új Feldspar műveleteket jelölnek.

3.4.3 Feltöltő operátorok

Még definiáltunk néhány kényelmi operátort, SIMD2 adatok egyszerű, és intuitív manipulálására.

`simd2` művelet operátorral megvalósítva:

```
(-|-) :: (Packable2 a) => Data a -> Data a -> SIMD2 a
```

Valamely elem cseréje:

```
(-||) :: (Packable2 a) => Data a -> SIMD2 a -> SIMD2 a
(||-) :: (Packable2 a) => SIMD2 a -> Data a -> SIMD2 a
```

Valamint SIMD4 adatok építésére:

```
(-\) :: (Packable4 a) => Data a -> Data a ->
      SIMD4Pack1 a
(-\\) :: (Packable4 a) => Data a -> SIMD4Pack1 a ->
      SIMD4Pack2 a
(-\\\) :: (Packable4 a) => Data a -> SIMD4Pack2 a ->
      SIMD4 a
```

Ahol `SIMD4Pack1` és `SIMD4Pack2` olyan típusok, melyekkel csak az itt látható műveletek operálnak. Ezek használatára példa:

```
f :: (Packable4) => Data a -> Data a ->
      Data a -> Data a -> SIMD4 a

f a b c d = a -\ b -\\ c -\\\) d
```

3.5 Indexelő műveletek

Szükség lehet a csomagokban tárolt elemek lekérdezésére is. Ezt teszi lehetővé a `SimdIndex` típusosztály, melynek definíciója:

```
class SimdIndex simdType a where
  (|!) :: simdType a -> Data DefaultInt -> Data a
```

Ahol `simdType` típusparaméter a csomag típusa, a pedig csomagolt típus. Egy példányosítás SIMD4 csomagok esetén:

```
instance (Packable4 a) => SimdIndex SIMD4 a where
  (|!) a i = (function2 "simd4_index"
                  sizepropagation evalfunction)
              (freezeSIMD4 a) i
```

Ahol `a` polimorf típus, így minden négyesével csomagolható típusra vonatkozik a példányosítás. A `sizepropagation` és

`evalfunction` függvények a fordító, és kiértékelő rendszernek nyújtanak segítséget, esetünkben nem fontosak.

Ezek az indexelő műveletek, semmilyen garanciát nem adnak arra vonatkozólag, hogy használatukkor az index értéke, nem esik kívül az indextartományon. Ha ilyen hibakezelésre szükség van, azt megfelelő platform támogatással lehet biztosítani. Ennek lehetőségével a platformtámogatás fejezetben részletesebben is foglalkozom.

3.6 Összegző műveletek

Még néhány kényelmi függvényt definiáltunk. Ilyen, az egy csomagban található elemek összegzése. Erre is típusosztályt hoztunk létre.

```
class Summarize simdType a where
  ssum :: simdType a -> Data a
```

Ahol *simdType* típusparaméter a csomag típusa, a pedig csomagolt típus. Példa példányosítás SIMD4 esetén:

```
instance (Packable4 a) => Summarize SIMD4 a where
  ssum = (function1 "SIMD4_sum" sizepropagation
    evalfunction) a
```

Az összeadás mintájára létrehoztunk egy másik osztályt, és a hozzá tartozó műveletet, a szorzás esetén támogatására:

```
class SIMDProduct simdType a where
  sprod :: simdType a -> Data a
```

Ahol *simdType* típusparaméter a csomag típusa, a pedig csomagolt típus. Példa példányosítás SIMD4 esetén:

```
instance (Packable4 a) => Summarize SIMD4 a where
  sprod = (function1 "SIMD4_sprod" sizepropagation
    evalfunction) a
```

4 Platform támogatás

A különféle platformok egészen eltérő módon is támogathatják a SIMD műveleteket. Természetesen olyan eset is előfordulhat, amikor egy platform egyáltalán nem rendelkezik SIMD műveletekkel, vagy amikor az általunk használt eszközkészletnek, csak egy részét támogatja. Ahol minden általunk használt eszköz támogatott, ott sem feltétlen működik a fordított kód speciális platform támogatás nélkül. Az is előfordulhat, hogy több platformon a SIMD támogatás eltérő logika szerint működik. Például C nyelven (Intel x86 architektúrán) nagyszámú intrinsic áll a programozó rendelkezésére, de ezek nagy része is ugyanazt a néhány műveletet hivatott biztosítani. LLVM platformon, az olyan SIMD műveletek, mint vektorok páronkénti összeadása vagy szorzása, a nyelv beépített elemeként, implicit használhatók. Ezeket a különbségeket a platformtámogatás elrejtheti oly módon, hogy a programozónak ne kelljen foglalkoznia az alacsony szintű platformspecifikus kóddal.

Beágyazott nyelvként, a Feldspar (pontosabban a Haskell) típusrendszere sok mindenre garanciát adhat, számtalan hibalehetőséget kiszűrhet, azonban vannak esetek, amikor ez sem adhat garanciát, vagy a lehetőségek teljes kihasználása, kevésbé lehet kényelmes a programozó számára.

A SIMD könyvtár néhány esetben, megengedőbb eszközöket ad a programozó kezébe. Ilyen például tetszőleges SIMD adatcsomag indexelése. Nincs garancia arra, hogy egy index, nem esik kívül az indextartományon. Például egy SIMD4 vektort nincs értelme hattal indexelni. Mivel azonban ez futási időben keletkező érték is lehet, a fordítónak nincs is lehetősége vizsgálatra vagy hibakezelésre.

Ilyen esetekben is célszerű lehet a platform támogatásra hagyatkozni, hiszen ebben a lépésben könnyedén megadható a platformnak leginkább megfelelő hibakezelési támogatás, amennyiben erre szükség van.

4.1 Átírási szabályok

A támogatást, a platform sajátosságait is figyelembe véve két helyen is meg lehet adni attól függően, hogy ez hol kényelmesebb. A további pontokban, ezt a két helyet részletesen be fogom mutatni.

Alapvetően olyan átírási szabályokkal kell dolgozni, amik kifejezésről kifejezésre, vagy programrészletről programrészletre transzformálják a kódot.

Ilyen átírási szabály lehet, ha például egy SIMD összeszorzását implicit támogató platformra (tehát a platformon használhatjuk a „*” operátort SIMD vektorok szorzására) akarunk lefordítani két SIMD vektor összeszorzását. A 3.3 pontban látható, hogy ez egy speciális Feldspar művelet, aminek két paramétere van.

Például:

```
f :: SIMD4 Int16 -> SIMD4 Int16 -> SIMD4 Int16
f v1 v2 = v1 * v2
```

Magnyelvi, és imperatív reprezentációbeli megfelelője is egy kétparaméteres függvény: `SIMD4_Mul`.

Melynek egy hívása pszeudokódban:

```
SIMD4_Mul(v1,v2)
```

Ezt kell átranzformálni erre a pszeudokódra:

```
v1 * v2
```

Ez az egyszerű átírási szabály ebben a környezetben természetesen szemantikusan hibás, hiszen `v1`, és `v2` elemek, „valódi” típusát, a rájuk alkalmazott művelet határozza meg oly szempontból, hogy jelenlegi szerepük szerint, csak adat tárolására használt helyek. Ezt a problémát, és megoldását, az 5.2, 5.3 fejezetben részletezem.

Átírási szabályokat meg lehet adni a Feldspar transzformációs keretrendszeréhez tartozó Rule Plugin segítségével, illetve a célplatformon is többféleképpen, például

függvénykönyvtárak hozzáadásával.

4.2 Rule plugin

A Rule Plugin egy általánosabb eszköze a szabályok leírásának. Lényegében ez is beágyazott nyelv, amely a Feldspar, absztrakt imperatív reprezentációját manipulálja.

A szabályok leírása egyszerű függvényekkel zajlik. Ezeket a függvényeket meg kell adni a fordítást konfiguráló függvénynek.

A szabályok alkalmazása a következőképpen zajlik:

A Rule Plugin bejárja a teljes absztrakt imperatív reprezentációt, és minden lépésben alkalmazza a szabályt alkotó függvényt. A függvények úgy épülnek fel, hogy mintát illesztenek a bemeneti paraméterre, és ha ez sikeres volt, akkor utasítások sorozatát rendelik hozzá. Amennyiben a mintaillesztés sikertelen, egy üres listát adnak vissza.

Utasításból kétfélét különböztetünk meg. A `replaceWith` egy új kifejezést vagy programrészletet illeszt, a szabály illeszkedő mintájának helyére. A `propagate` egy új szabályt futtat le az absztrakt imperatív reprezentáción.

A fordítás folyamata alatt a SIMD könyvtár után ez a következő pont, ahol megkülönböztetünk egyszerű adatot, és SIMD típusú adatot.

Ezt a megkülönböztetést az adatokon végzett műveletek szerint tesszük meg. Ha egy függvény, vagy operátor SIMD specifikus adatot vár bemeneti paraméterként vagy ad vissza tudjuk, hogy ezeket az adatokat így kell kezelni.

A korábbi szorzást, átalakító szabályt így írhatjuk le:

```
SIMD4_Mul_to_op (Fun t "SIMD2_Mul" [a1,a2]) =  
  [ replaceWith $ Binop t "*" [a1,a2] ]  
SIMD4_Mul_to_op _ = []
```

Mintát illesztünk egy műveletre, az imperatív program egy függvény hívására, melynek visszatérési értékének típusa egy generikus `t` típus, neve „SIMD2_Mul” és két bemeneti paramétere van, ezeket elnevezzük `a1` és `a2`-nek. Amennyiben létezik ilyen, azt lecseréljük egy olyan bináris operátorra, melynek visszatérési

típusa `t`, neve „`*`” és két bemeneti paramétere van: `a1` és `a2`.

Ahol a minta nem illeszkedik, a végrehajtandó parancsok listája üres.

A Haskell nyelv, és ezáltal a Feldspar is képes operátor túlterhelésre, ezáltal képes többféle típushoz ugyan azt a műveletet használni. Azonban nem mindent platform képes erre.

Ugyan az a Feldspar művelet hajtódik végre két SIMD4 Int8-ak összeadásakor, és két SIMD4 Int16-ok összeadásakor. Ennek legfőbb oka, az áttekinthetőség megőrzése volt. Például Intel (x86) architektúrán `ansi C` nyelven, 4.5.2-s verziószámú `gcc` fordító programmal fordítva, nem származik probléma ebből, hiszen minden SIMD összeadásra használható a szokásos `+` operátor. Ennél kisebb verziószámú `gcc` esetén azonban a platform speciális `intrinsic`-eit kell használnia a programozónak. Ebben az esetben a Rule Plugin-t felhasználva kell ezeket a platformspecifikus elemeket a kódba helyezni.

Az alábbi szabály segítségével lehet feloldani a fenti Int8 - Int16 problémát:

```
SIMD4_Add_Int16_Int32_Intrinsics (Fun I32 "SIMD4_Add"
                                   [a1,a2]) = [ replaceWith $
                                   Fun I64 "SIMD4_add_Int8" [a1,a2] ]
SIMD4_Add_Int16_Int32_Intrinsics (Fun I64 "SIMD4_Add"
                                   [a1,a2]) = [ replaceWith $
                                   Fun I32 "SIMD4_add_Int16" [a1,a2] ]
SIMD4_Add_Int16_Int32_Intrinsics _ = []
```

A szabálynak kétféle eredménye lehet. Vagy egy 32 bites integert visszaadó `SIMD4_Sum` függvényre próbál illeszkedni, vagy egy 64 bites integert visszaadóra. Ha 32 bitesre illeszkedik, akkor tudjuk, hogy itt Int8 típusú adatok összegéről van szó, hisz az tisztázott, hogy SIMD4 típusú műveletet végzünk, és pontosan négy darab Int8 típusú adat fér el 32 biten. Ilyenkor ezt a műveletet ki kell cserélni `SIMD4_sum_Int8` műveletre, ami már egy operátor túlterhelést nem támogató platform számára is megfelelő. Ugyanilyen logika mentén kikövetkeztethető, hogy 64 bites művelet esetén `SIMD4_sum_Int16`-ra van szükség.

4.3 Célplatform

Célplatformon többnyire új függvénykönyvtár hozzáadásával lehet legkönnyebben platformtámogatást adni. Nézzük például a 4.2 szorzás példáját C nyelven (Intel x86 architektúrán).

Az új könyvtárban el kell helyeznünk egy olyan függvényt, vagy makrót, ami a támogatni kívánt függvényhívásokat helyettesítik vagy ami ennél pontosabb, definiálják azokat. Használjunk makrót, hiszen ez mindig inline helyettesítődik. Például a 4.1 pontban szereplő szorzás definíciója így nézhet ki:

```
#define SIMD4_Mul(arg1, arg2)
    ({ v4hi temp = ((* (v4hi*) &arg1) +
    (*(v4hi*) &arg1)); *((int64_t*) & temp); })
```

v4hi platformspecifikus típus, melynek definíciója:

```
typedef int16_t v4hi __attribute__((vector_size (8)));
```

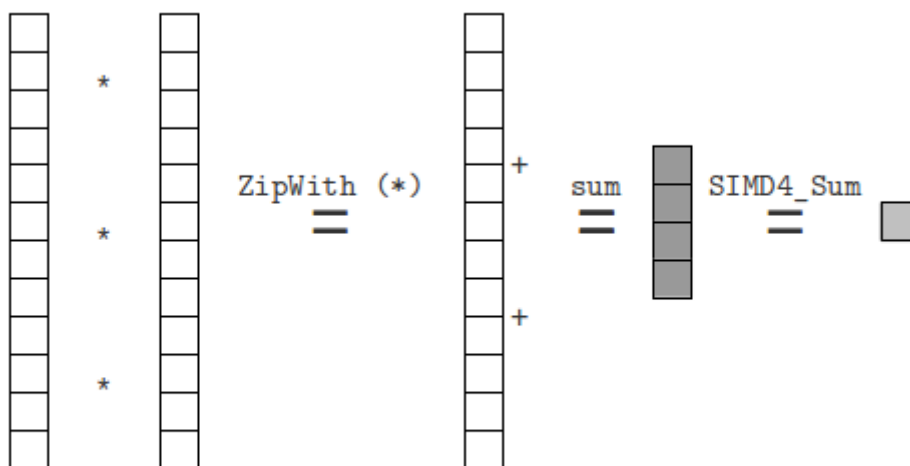
Ez a típusdefiníció a következőt jelenti: Egy int16_t típusú elemekből felépített nyolc bájt hosszú vektor, melynek neve v4hi. Természetesen ezt a típusdefiníciót is hozzá kell adnunk a platformtámogatáshoz.

Ilyen típusdefiníciót a Rule Plugin nem támogat, így ez egy olyan platformspecifikus elem, melyet csak a célplatformon lehet megadni.

5 Példa: Skalárszorzás

A skalárszorzás egyszerű példáján keresztül mutatjuk be a fordítási folyamat SIMD optimalizáció specifikus részleteit.

Intel (x86) architektúrán lehetőség van négy integert SIMD vektorként kezelni, így a példában is SIMD4 típussal optimalizáltunk.



Ábra 5

Mint az már említésre került, összesen három helyen teszünk különbséget SIMD típusok adatok, és egyszerű Data típusú adatok között. A legelőször a programozó szintjén vagy a Feldspar interface szintjén beszélhetünk különbségekről. Ez az a szint, ahol a programozó kifejezheti szándékát, a SIMD optimalizációra.

A Rule Pluginban, és Célplatformban olyan esetekben találkozunk különbségekkel, amikor az adatfolyam egy részére, a programozó SIMD optimalizációt kívánt használni.

5.1 Feldspar interface

A SIMD könyvtár tervezésénél egyik fő szempont volt, hogy a SIMD optimalizálás ne változtasson a kelleténél többet egy algoritmus kódján.

Az eredeti Feldsparos skalárszorzás kódja így nézett ki:

```
scalarProd :: (...) =>
    Vector (Data a) -> Vector (Data a) -> Data a
scalarProd a b = sum (zipWith (*) a b)
```

Az *Ábra 5* ábrán is látható, hogy a típusok átírása mellett, itt egy plusz műveletre is szükség van, hiszen a `sum (zipWith (*) a b)` eredményének is `SIMD4 a` lesz a típusa. Erre létezik egy kényelmi függvény, a `Summarize` osztály `ssum` függvénye, ami egy tetszőleges SIMD vektort összegez.

Ezzel kiegészítve:

```
scalarProd :: (Packable4 a, ...) => Vector (SIMD4 a) ->
    Vector (SIMD4 a) -> Data a
scalarProd a b = ssum $ sum (zipWith (*) a b)
```

Tehát a SIMD optimalizáció során, csupán néhány apróságot kellett módosítani az algoritmus kódján: az algoritmust leíró függvény szignatúráját, illetve be kellett szűrni egy plusz műveletet.

5.2 Technikai apróságek

Érdemes áttekinteni azokat a műveleteket melyeket támogatnunk kell a skalárszorzás példájához.

Támogatnunk kell olyan műveletet, mely összead két `SIMD4` vektort, összeszoroz két `SIMD4` vektort, illetve összegez egy `SIMD4` vektort. Ezeknek Feldspar primitív megfelelőik rendre: `SIMD4_Add`, `SIMD4_Mul`, `SIMD4_Sum`. Ezen műveletek mindegyike

esetünkben 64 bites integereket vár paraméterül és azokkal is tér vissza. C99 nyelvre és X86 architektúrára akarunk támogatást nyújtani, tehát ennek az architektúrának a speciális intrinsic-eit és típusait kell használnunk C99 nyelven. 4.5.2 verziószámú gcc-t használunk így a speciális intrinsic-ek hívásával nem kell foglalkoznunk, elég a C nyelv egyszerű operátorait használnunk. Az egyetlen amire külön figyelmet kell fordítanunk, a típusok kezelése. A SIMD4 Int16 típus X86 architektúra szerinti megfelelője a `v4hi` (vector 4 half integer) típus.

5.3 Rule Plugin

Az 5.1 fejezet alapján a Rule Plugin szintjén történő támogatáshoz, egy hasonló szabályra lenne szükség például a szorzás műveletéhez:

```
mul (Fun I64 "SIMD4_Add" [a1,a2]) =  
    [ replaceWith $  
      BinOp I64 "+" [a1,a2] ]
```

Ezzel azonban a következő a probléma: Egy ilyen összeadás generálásából nem derülne ki, hogy SIMD összeadást akarunk használni, hiszen `a1` és `a2` adatok a generált kód minden pontján 64 bites integereként jelennek meg. Tehát ezek használata előtt `a1` és `a2` kifejezéseket konvertálnunk kellene a megfelelő `v4hi` típusra. Sajnos ez viszont nem lehetséges, mert `v4hi` és a 64 bites integer C99 nyelvi megfelelője az `int64_t` típus merőben eltérnek egymástól, így egyszerű típuskonverzió nem lehetséges közöttük. A megoldást az jelentené, ha a szóban forgó adat címét lekérdeznénk, ezt a mutatót konvertálnánk más típusú mutatóra, majd ezt a memória területet kérdeznénk le. Ez azonban nem írható le a Rule Plugin szintjén. Ennek oka, hogy a Rule Pluggin által használt reprezentáció nem tartalmaz pointereket.

5.4 Célplatform

Már szó volt azokról a műveletekről, melyeket használunk a skalárszorítás példájához:

```
SIMD4_Add  
SIMD4_Mul  
SIMD4_Sum
```

Ezekből, Intel (x86) architektúrán az első kettőt lehet intrinsice fordítani. Mi ezeket is egyszerű operátoroknak fordítottuk, mert 4.5.2 verziószámú gcc-t használtunk. A harmadik műveletnek, azaz SIMD4_Sum-nak az általunk használt platformon nincs intrinsic megfelelője, szerencsére ez most nem igazán lényeges teljesítmény szempontjából, hiszen ez az algoritmus alatt mindössze egyszer hívódik meg, az első kettővel szemben, melyek annyiszor, ahány elem van SIMD4 elemeket tartalmazó tömbben.

A támogatás a skalárszorítás példájánál annyiból állt, hogy definiáltuk a szükséges SIMD típusokat, és megadtuk a három művelet implementációját.

Szükséges egy olyan típus, ami a leírja, hogy miként kell tekinteni egy memóriában tárolódó adatcsomagra. Ez a típus a *v4hi*.

Definíciója:

```
typedef int16_t v4hi __attribute__((vector_size (8)));
```

Nyolc byte-ból álló adatterületet osszon fel *int16_t* típusú adatokra, és ezt az egészet hívja *v4hi*-nek. (vector 4 half integer)

Két *v4hi* típusú adatot lehet összeadni az egyszerű + operátorral. Olyan implementációt kellett tehát adnunk, mely elvégzi ezeket a típus konverziókat, végrehajtja az utasítást, majd visszafelé is elvégzi a típus konverziót az eredményen. Mivel ezek eléggé fontos műveletek teljesítmény szempontjából, így ezekre makrókat definiáltunk.

```
#define SIMD4_Mul(arg1, arg2) ({ v4hi temp =
    ((* (v4hi*) &arg1) * (* (v4hi*) &arg1));
    *((int64_t*) & temp); })
#define SIMD4_Add(arg1, arg2) ({ v4hi temp =
    ((* (v4hi*) &arg1) + (* (v4hi*) &arg1));
    *((int64_t*) & temp); })
```

SIMD4_Sum műveletre egy egyszerű függvényt definiáltunk:

```
int64_t feldspar_4x16_sum (int64_t arg) {
    return ((*((int16_t*)&arg) + 0) +
        *((int16_t*)&arg) + 1)) +
        ((*((int16_t*)&arg) + 2) +
        *((int16_t*)&arg) + 3));
}
```

5.5 Generált kódok

5.5.1 C platform tömb típusa

A Feldspar fordító C platformra, egy speciális tömb típust használ. A generált kódok megértéséhez, szükség lehet ennek megértésére is.

Az új töm típus létrehozásának sok oka van, de a legfőbb, hogy olyan magas szintű szolgáltatásokat (függvényeket) kellett a tömbökhöz adnunk, melyek egy sajátos adatszerkezet definiálását is igényelték.

Az új tömb típus definíciója:

```
struct array
{
    void* buffer;
    unsigned int length;
    int elemSize;
};
```

`void*` `buffer` tárolja annak a C memória területnek a címét, melyen az adatok helyet foglalnak.

`unsigned int` `length` tárolja a tömb hosszát.

`int` `elemSize` tárolja, egy tömbbeli elem méretét, byte-okban.

`void*` típusból látható, hogy ez egy típustalan reprezentáció, tehát bármilyen típusú adatot képes tárolni. Olyan helyeken, ahol szükség van az elemek típusára, például egy művelet elvégzésénél, a Feldspar fordító ezt képes megadni. Mindössze egyetlen ilyen művelet létezik, ez pedig a tömb indexelése. Ezt egy makróval végzi, melynek definíciója:

```
#define at(type,arr,idx) (((type*)((arr).buffer))[idx])
```

Egy indexelés:

```
at(int16_t,tomb,index)
```

`length` adattagra nyilvánvaló okokból szükség van.

`elemSize` adattagra azért van szükség, mert az olyan műveletek, mint a másolás, ezen adattag segítségével határozzák meg, hogy pontosan mekkora adatterület másolására van szükség.

Minden generált C függvény bemeneti, és kimeneti tömbje is mutatókon keresztül érhető el.

5.5.2 Generált kódok

Az eredeti Vektor könyvtár skalárszorzás függvényének generált kódja:

```
// scalarProd :: Vector (Data Int16) ->
//              Vector (Data Int16) -> Data Int16
/*
 * Memory information
 *
 * Local: none
```

```

* Input: signed 16-bit integer array, signed 16-bit
*                                     integer array
* Output: signed 16-bit integer
*
*/
void scalarProd(struct array mem, struct array in0,
struct array in1, int16_t * out2)
{
    uint32_t v3;
    uint32_t v4;
    int16_t temp6;
    uint32_t w5;

    v3 = length(in1);
    v4 = length(in0);
    if((v3 < v4))
    {
        w5 = v3;
    }
    else
    {
        w5 = v4;
    }
    (* out2) = 0;
    {
        uint32_t i7;
        for(i7 = 0; i7 < w5; i7 += 1)
        {
            temp6 = ((* out2) + (at(int16_t,in0,i7) *
                                at(int16_t,in1,i7)));
            (* out2) = temp6;
        }
    }
}

```

SIMD könyvtár változatának generált kódja:

```

// scalarProd :: Vector (SIMD4 Int16) ->
//               Vector (SIMD4 Int16) -> Data Int16

```

```

/*
 * Memory information
 *
 * Local: none
 * Input: signed 64-bit integer array, signed 64-bit
 *                                     integer array
 * Output: signed 16-bit integer
 *
 */
void scalarProd(struct array mem, struct array in0,
struct array in1, int16_t * out2)
{
    uint32_t v3;
    uint32_t v4;
    int64_t w5;
    int64_t temp7;
    uint32_t w6;

    v3 = length(in1);
    v4 = length(in0);
    if((v3 < v4))
    {
        w6 = v3;
    }
    else
    {
        w6 = v4;
    }
    w5 = 0;
    {
        uint32_t i8;
        for(i8 = 0; i8 < w6; i8 += 1)
        {
            temp7 = SIMD4_Add(w5,
                               SIMD4_Mul(at(int64_t,in0,i8),
                                           at(int64_t,in1,i8)));
            w5 = temp7;
        }
    }
    (* out2) = SIMD4_sum(w5);
}

```

```
}
```

A generált kódok között, meglehetősen kevés eltérés tapasztalható.

Az első különbség, a `Memory Information` comment blokkban található. Az eredeti kód bemeneti paraméterei 16 bites integerek tömbje, a SIMD optimalizált kód bemeneti paraméterei pedig 64 bites integerek tömbje. Ebből látszik, hogy SIMD optimalizált kódban, nagyobb adatterületekkel dolgozunk. Ennek következménye, hogy egy ciklus iteráción belül is nagyobb adatterületet kezelünk egyszerre, így nyilvánvaló módon kevesebb ciklus iterációra van szükség, ugyanakkor bemeneti tömbök esetén.

A második különbség, nyilvánvaló módon a skalárszorzás ciklus magjában található műveletek. Az eredetiben szokásos C operátorok, az SIMD optimalizáltban az ezeknek megfelelő `Feldspar` SIMD műveletek.

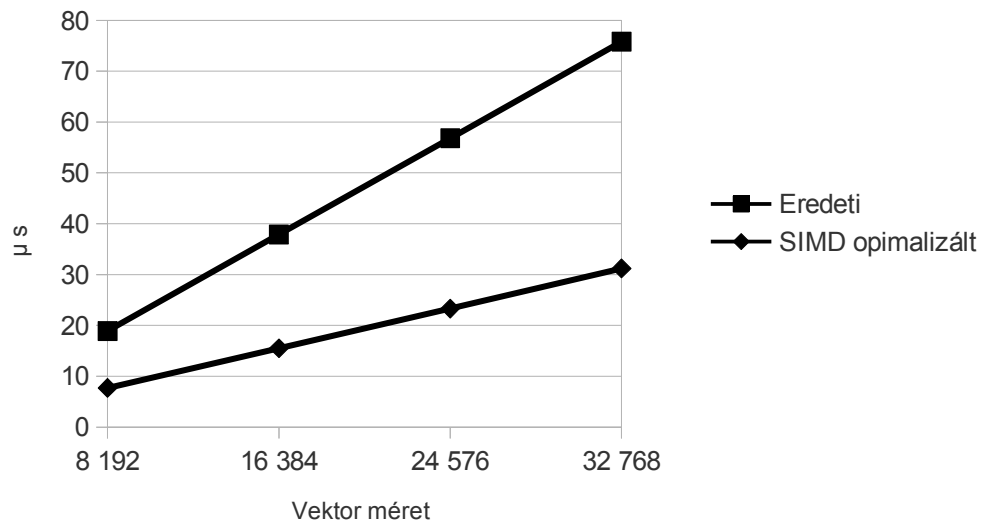
5.6 Eredmények

A `Feldspar` eredeti Vektor könyvtárában található skalárszorzás függvényét hasonlítottuk össze a SIMD könyvtár eszközeivel implementált skalárszorzással.

A mérést egy *Intel Core 2 Duo CPU*-t (melynek órajel frekvenciája 2.10 Ghz) tartalmazó számítógépen végeztük, melyen Linux alapú operációs rendszer (kernel: 2.6.38-13) található, és a generált C kód fordításához `gcc`-t (4.5.2) használunk.

Mind a két kódot négy különböző bemeneti vektormérettel teszteltük. Ezek a vektor méretek: 8 192, 16 384, 24 576, 32 768. Milyen tesztesetet 100 000-szer futtattunk és az eredményeket átlagoltuk, hogy minél valósabb arányt kapjunk eredményül.

Eredeti, és SIMD optimalizált skalárszorzás mérési eredményei



Az alábbi diagramon jól láthatóak az eredmények, a SIMD optimalizált skalárszorzás egyenletesen, minden vektor méret esetén, 60% körüli gyorsulást ért el.

6 Összefoglalás

A digitális jelfeldolgozás területén az egyik leghatásosabb és leggyakrabban használt optimalizációs eljárás, hogy az adatfolyam néhány egymás után következő, azonos típusú elemeit, úgynevezett SIMD (Single Instruction Multiple Data) típusú műveletekkel, „párhuzamosan” dolgozzuk fel. Ez annyit jelent, hogy bizonyos processzorok, képesek egyszerű műveleteket (például összeadás), egyszerre elvégezni több elempáron (például négy darab 16 bites integeren), közel azonos idő alatt, mintha egyen végezné.

Ezért kiegészítettem a Feldspar nyelvet, egy olyan magas szintű könyvtárral, mellyel a programozó képes az ilyen speciális SIMD utasításokat használni úgy, hogy az eredeti kódon csupán minimális változtatásokra van szükség. Az ebből generált platform függő kódok már tartalmazzák a speciális műveleteket.

Hatékonyágelemzéshez a skalárszorzás műveletet választottam. Ezt implementáltam az új, SIMD optimalizációt támogató könyvtár segítségével. Azt tapasztaltam, hogy forráskódja az eredeti, vektor könyvtárban megtalálható implementációtól csak minimálisan tér el, azaz sikerült megőrizni a Feldspar magas absztrakciós szintjét az új könyvtárban is. A két implementációt x86 architektúrára fordítottam le. Méréseim szerint a SIMD könyvtár segítségével implementált változat több mint kétszer (kb. 60%-kal) gyorsabb az eredeti implementációnál.

7 Hivatkozások

- [1] Feldspar: Functional Embedded Language for Digital Signal Processing and Parallelism
<http://feldspar.inf.elte.hu/>
- [2] D. A. Patterson, and J. L. Hennessy, Computer Organization and Design, Revised Fourth Edition: The Hardware/Software Interface, Elsevier, Inc., 225 Wyman Street, Waltham, MA 02451, USA, 2011, 4th edn., ISBN 978-0-12-374750-1.
- [3] H. Thielemann, CoRR abs/1004.4796 (2010).
- [4] MOUSE: A Shortcut From Matlab Source to SIMD DSP Assembly Code
Gordon Cichon and Gerhard Fettweis
Vodafone Chair Mobile Communications Systems
Technische Universität, D-01069 Dresden
- [5] A Domain-Specific Language for the Generation of Optimized SIMD-Parallel Assembly Code
Software Quality Research Laboratory
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1
- [6] Automatikus generált transzformációs keretrendszer beágyazott programozási nyelvhez
Erdélyi Viktor Tamás
ELTE Informatikai Kar