DIPLOMAMUNKA

Major Sándor Roland

Debrecen

2011

Debreceni Egyetem

Informatika Kar

Algoritmusok implementálása FPGA-n

Témavezető:
Dr. Herendi Tamás
Egyetemi adjunktus

Készítette:
Major Sándor Roland
Programtervező informatikus

Debrecen
2011

Tartalomjegyzék

1. Bevezetés

A Field-programmable gate array (FPGA) chip-ek rendkívül sokoldalúan felhasználható eszközök. Rugalmasságuk a felépítésükből következik. A chip egyik szintjén különböző számításokhoz használható elemek találhatók, ezek típusai, száma és felépítése a chip-től függ. A legfőbb számításhoz használt elemek a look-up table-ök tömbje. Egy n bites bemenetű LUT 2^n bitet tárol, a kimenete ezen bitek közül az input alapján kiválasztott egy bit. A tárolt érték beállításával a LUT bármilyen legfeljebb n bit bemenetű Boole-függvényt meg tud valósítani. Szintén szokásos elemek a flip-flop-ok, blokk-memóriák, dedikált szorzók és DCM (digital clock manager) egységek. Ezen elemek között egy összeköttetéseket tartalmazó szint teremt kapcsolatot. A létrehozható kapcsolatok száma rendkívül nagy, ami magyarázza az eszköz rugalmasságát. A konfigurálás során a szükségleteknek megfelelő „célhardware" jön létre. Hatalmas előny azonban az újraprogramozhatóság, a létrehozott össszeköttetések nem „égnek bele" az FPGA-ba, a chip tetszőlegesen sokszor konfigurálható.

Az FPGA-k történelme az 1980-as években kezdődött, elődeinek tekinthetők a PLD-k (programmable logic device) és PROM-ok (programmable read-only memory). A CPLD (complex PLD) eszközök célja az FPGA-hoz hasonló, de a flexibilitásuk a felépítésükből adódóan alulmarad. Mára az FPGA-k számtalan területen kerülnek felhasználásra, mint a digitális jelfeldolgozás, kriptográfia, orvosi képalkotás, beszédfelismerés, célirányos (ASIC) alkalmazások fejlesztése és minden egyéb olyan terület, ahol különleges lehetőségei hasznosíthatók.

FPGA chip-ek segítségével mind kombinációs (vagyis memória nélküli, csak az inputtól függő függvényt megvalósító), mind szekvenciális (vagyis memóriával rendelkező, a megelőző inputoktól is függő függvényt implementáló) logika egyszerűen létrehozható. A kombinációs hálózatoknak megfelelő Boole-hálózatokról és a hozzájuk kapcsolódó bonyolultságelméleti fogalmakról a 2. fejezetben teszünk említést.

Az FPGA-k segítségével könnyen nagy fokú valódi párhuzamosság érhető el. A párhuzamos számítások bonyolultságelméleti vizsgálatáról szintén szót ejtünk. Röviden tárgyaljuk, hogy milyen elvárásaink vannak egy „jól" párhuzamosított algoritmussal szemben tekintettel a neki megfelelő szekvenciális algoritmusra. Definiáljuk az NC bonyolultsági osztályt, amely hasonló szerepet tölt be a párhuzamos számításokra nézve, mint a P osztály a szekvenciális számításoknál, vagyis a párhuzamos géppel hatékonyan megoldható problémák halmazának tekinthető.

A dolgozat több FPGA-kon készült implementációt is taglal. Ezek két különböző eszközre készültek, egy Spartan-3E XC3S250E FPGA-ra és a jóval nagyobb Virtex-5 XC5VLX110T FPGA-ra. A kisebb Spartan-3E kiválóan alkalmas az FPGA chip-ek által nyújtott lehetőségekkel való megismerkedésre és a konfigurálásukhoz szükséges alapvető ismeretek elsajátítására. Habár a technikai paraméterei elmaradnak a drágább, nagyobb teljesítményű eszközökétől, a Spartan-3E alkalmas a gyakorlatban is használható és igen költséghatékony alkalmazások kifejlesztésére. A Virtex-5 XC5VLX110T ezzel szemben egy jelentősen jobb paraméterekkel rendelkező és szélesebb körben felhasználható eszköz. Az általa nyújtott lehetőségek sokkal nagyobb és összetettebb alkalmazásokat is támogatnak.

Az XC3S250E FPGA-ra elkészült implementációk közül hármat mutatunk be. Mindhárom konfigurá-

ció szorzást hajt végre különböző FPGA-kon hatékonyan létrehozható algoritmusok felhasználásával. Az implementációk segítségével összehasonlíthatjuk az algoritmusok sebességét és erőforrásigényüket. A konfigurációk a Karatsuba-szorzás, a Partial Product és a Computed Partial Product eljárásokat valósítják meg.

Az XC5VLX110T FPGA-n egy rendkívül gyors mátrixszorzást végző implementációt mutatunk be. Ebből a munkából TDK dolgozat és publikáció is született. A modulok egy álvéletlen számsorozat generáló eljáráshoz kapcsolódnak, amelynek része nagyméretű mátrixok nagyon magas hatványra emelése. Kihasználva azt, hogy a mátrixok elemei két biten ábrázolhatók olyan modult írunk le amely egy órajel alatt képes vektorok belső szorzatának kiszámítására. Az így kapott modulokból hierarchikus módon egy mátrixszorzást végző modult szerveztünk össze. Bemutatjuk a kísérletet, amely az implementáció több fontos technikai paraméterét is meghatározta. Az adatok tárolása részben shift regiszterekből kialakított tárolókban, részben az eszközhöz csatlakoztatott 256MB-os DDR2 SODIMM memórián kerültek elhelyezésre. Szót ejtünk a memória hatékony felhasználásának gyakorlati kérdéseiről. Az adatok felhasználásához megadunk egy algoritmust, amely teljes egészében elvégezhető a számításokkal párhuzamosan, vagyis a számítás szempontjából soha nincs „holt idő". Az így kapott konfiguráció futási ideje megegyezik a számítás idejével, amely a mátrixok és a szorzómodul fix méretéből adódóan egy konstans. Felvázoljuk a munka továbbfejlesztésének tervezett vonalát, amely a mátrixok méretének növeléséből és a Strassen-algoritmus segítségével a futási idő további optimalizálásából áll.

Az implementációk elkészítése a Xilinx ISE Design Suite fejlesztői környezettel történt. Az XC3S250E FPGA-ra készült megvalósítások a 10.1 verzióban készültek, a XC5VLX110T FPGA-ra pedig a 12.2 verzióban történt a fejlesztés. Az ISE Design Suite a teljes fejlesztési folyamatot végigkíséri és több olyan szoftvert is tartalmaz amelyek a kifejezetten a folyamat egyes fázisait támogatják, mint a konfigurálást végző iMPACT vagy a PlanAhead, amellyel az implementáció chip-en való fizikai elhelyezkedését vizsgálhatjuk. A konfigurációk szimulálása a ModelSim szoftverrel történt.

Az itt ismertetett projektek Verilog harware definíciós nyelven íródtak. Az ISE a Verilog hardware definíciós nyelven írt .v kiterjesztésű fájlokból egy többlépcsős folyamaton keresztül elkészíti az FPGA-k konfigurálására használt .bit fájlt. A folyamat három fő lépcsőfoka a Szintézis, az Implementálás és a Generálás. A Szintézis során történik többek között a modulok optimalizálása, ami nagy méretű projektek esetén rendkívül időigényessé is válhat. Az Implementálás három fázisa során (Fordítás, Feltérképezés, Elhelyezés) kerül a Szintézis során megkapott alkotóelemek és összeköttetéseik hozzárendelése fizikai alkatrészekhez, vagyis a rendszer kijelöli a konfiguráció helyét az FGPA chip-en. Az így kapott adatokból a Generálás során (az előző két lépcsőfokhoz képest jelentősen gyorsabban) megkapjuk a .bit fájlt.

A dolgozathoz tartozó CD-mellékleten megtalálhatóak a tárgyalt implementációk ISE-projektjei.

2. Bonyolultságelméleti áttekintés

2.1. Hálózati bonyolultság

Ebben a fejezetben elsőként a Boole-hálózatokkal kapcsolatos néhány elméleti eredményt említünk meg, majd a párhuzamos számítások bonyolultságelméleti vizsgálatáról ejtünk szót.

A Boole-hálózatok az FPGA-kon létrehozható kombinációs logikához való kapcsolatuk miatt érdekesek számunkra. Lássunk néhány szükséges definíciót.

Egy n változós Boole-függvény alatt egy $f : \{igaz, hamis\}^n \to \{igaz, hamis\}$ alakú függvényt értünk. Az olyan logikai műveletek mint az ÉS, VAGY és NEM szintén tekinthetők Boole-függvényeknek. A Boole-formulák szintén értelmezhetők olyan Boole-függvényként, amely a benne található logikai változókból előállít egy új $igaz$ vagy $hamis$ értéket. Másfelől minden n változós Boole-függvény leírható legfeljebb n változós Boole-formulával. Ennek belátásához legyen F az $\{igaz, hamis\}^n$ azon vektorainak halmaza amelyre f igaz. Minden $t = (t_1, t_2, \ldots, t_n) \in F$ vektorra legyen D_t olyan n-tagú konjunkció amelynek i. tagja az i. változó, ha t_i igaz vagy az i. változó tagadása ha t_i hamis. Az f függvénynek megfelelő formula ekkor az összes ilyen D_t diszjunkciója. Ha az F halmaz üres, akkor a formula lehet egyszerűen $x_1 \wedge \neg x_1$.

A Boole-hálózatok a Boole-függvények igen takarékos reprezentációját tudják jelenteni a formulákhoz képest. Legyen a hálózat egy $C = (V, E)$ irányított gráf, ahol $V = \{1, 2, \ldots, n\}$ a csúcsok (másnéven kapuk) halmaza. A gráfban nem engedjük meg az irányított köröket, így feltehetjük, hogy minden (i, j) élre $i < j$ teljesül. Feltesszük továbbá, hogy a csúcsokba bevező élek száma (befoka) csak 0,1 vagy 2 lehet. Ez a feltevés attól függ, hogy milyen kapukat kívánunk használni a hálózatban. Azok a kapuk, amiket ebben a fejezetben használunk mind megfelelnek ennek a feltételnek. A mátrixszorzást megvalósító FPGA implementációval foglalkozó fejezetben fontos szerepet fognak olyan hálózatok játszani, amelyekben a kapuk befoka 6. Minden i kapuhoz hozzárendeljük az $s(i) \in \{igaz, hamis, ÉS, VAGY, NEM\} \cup \{x_1, x_2, \ldots, x_n\}$. A kapu befoka 0 ha a kapu típusa $igaz, hamis$ vagy valamelyik változó. Ezeket a kapukat, amelyekbe nem vezet más él, tekintjük a hálózat bemenetének. Ha a kapu típusa NEM, akkor a befoka 1, vagy ÉS vagy VAGY, akkor pedig 2. A hálózat kimenete az a legnagyobb sorszámú kapu amelynek nincs kimenő éle. A hálózatot úgy is értelmezhetjük, hogy egyszerre több függvényt számít ki, ekkor az összes olyan kapu a kimenet része amelynek nincs kimenő éle.

A kimenet meghatározása a kapuk $T(i)$ igazságértékének kiszámításával történik, az i kapuk növekvő sorrendben vételével:

- Ha $s(i) = igaz$, akkor $T(i) = igaz$, ha pedig $s(i) = hamis$, akkor $T(i) = hamis$.
- Ha $s(i) \in X$, vagyis a kapu egy változót jelöl, akkor $T(i) = T(s(i))$, vagyis a kapu értéke a változó értékével megegyező.
- Ha $s(i) = NEM$, akkor létezik pontosan egy olyan $j < i$ kapu, amelyre $(j, i) \in E$. $T(j)$ ekkor már definiált. Legyen $T(i) = \neg T(j)$.

- Ha $s(i) =$ÉS, akkor két i-be vezető él létezik, amelyeket jelöljünk (j, i) és (j', i)-vel. A j és j' kapuk értéke ekkorra már definiált. Legyen $T(i) = T(j) \wedge T(j')$.
- Ha $s(i) = VAGY$, akkor az előző esettel megegyezően járunk el, kivéve, hogy $T(i) = T(j) \vee T(j')$.

A teljes C hálózat $T(C)$ értéke alatt $T(n)$-t értjük, ahol n a kimeneti kapu.

Egy formulához könnyen tudunk induktív módon hálózatot definiálni, minden részformulához egy új kapu bevezetésével. Az így kapott hálózat mérete (vagyis a kapuk száma) megegyezik a formuláéval, vagyis a reprezentáció nem lett takarékosabb. A részformulákat azonban újrahasznosíthatjuk, ha a kapuk kifokát nem korlátozzuk. Ekkor a többször előforduló részformulákhoz nem szükséges minden alkalommal új kaput bevezetnünk. Ilyen hálózatokkal a Boole-függvények takarékosabban reprezentálhatók, mint a formulákkal. Minden hálózat kiszámít egy Boole-függényt. Azt mondjuk, hogy az n változós C hálózat kiszámítja a f Boole-függvényt, ha az összes lehetséges $t = (t_1, t_2, \ldots, t_n)$ vektorra $f(t) = T(C)$ ha a hálózat minden x_i változójára $T(x_i) = t_i$. Minden Boole-függvény kiszámítható hálózattal, hiszen a függvény leírható formulával. A hálózatok bonyolultságának fő kérdése az, hogy n-től függően mekkorának kell lennie a hálózatnak.

A hálózatok tekinthetjük úgy, hogy bizonyos $x = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ szavakat elfogad, másokat pedig elutasít. Ahhoz, hogy tetszőleges $\{0,1\}$ feletti nyelvet fel tudjunk ismerni, minden lehetséges bemenzhosszhoz alkotnunk kell egy hálózatot. Legyen a C hálózatcsalád $C = (C_0, C_1, \ldots)$ hálózatok sorozata, ahol C_n n változós. Az $L \subseteq \{0,1\}^*$ nyelv eldönthető polinomiális hálózatcsaláddal, ha létezik olyan p polinom amelyre C_n mérete legfeljebb $p(n)$ és egy x szó csak akkor eleme az L nyelvnek, ha $C_{|x|}$ igaz kimenetet ad az x bemenetre.

A változómentes hálózatok kiértékelése P-teljes probléma. Létezik visszavezetés, amely bármely $p(n)$ időben eldönthető L nyelvre és x bemenetre megad egy $O(p(|x|)^2)$ méretű hálózatot, amely az x-en történő számítási folyamatot kódolja. Ennek a hálózatnak a kiértékelése a definícióban megadott módon polinom időben elvégezhető.

Minden P-beli nyelv felismerhető polinomiális hálózatokkal, az állítás megfordítása viszont igen meglepő eredményt ad. Léteznek polinomiális hálózatokkal eldönthető nem rekurzív nyelvek, vagyis a számítási ereje meghaladja a Turing-gépét! Ez arra figyelmeztet minket, hogy a számítási modell nem reális, vagyis további módosításra van szükség. A modell hibájára fényt derít ha megmutatjuk hogyan oldható meg vele egy megoldhatatlan feladat.

Legyen L egy tetszőleges eldönthetetlen nyelv a $\{0,1\}$ ábécé felett, és legyen $U \subseteq \{1\}^*$ unáris nyelv a következő: $U = \{1^n : n\ bináris\ alakja\ L - ben\ van\}$. Világos, hogy U eldönthetetlen, hiszen L visszavezethető rá. A visszavezetés ugyan exponenciális idejű, de ez az eldönthetetlenségen nem változtat. U egy elég triviális hálózatcsaláddal felismerhető. Ha $1^n \in U$, akkor a C_n hálózat álljon $n-1$ ÉS kapuból, amelyek az input konjunkcióját számítják ki. Ez akkor és csak akkor lesz igaz, ha a bemenet 1^n. Ha $1^n \notin U$, akkor a hálózatban egyáltalán nem kellenek élek, elég a bemeneti kapukból és a *hamis* típusú kimeneti kapuból állnia. Ez minden bemenetet elutasít. Az így kapott hálózatcsalád nyilván pontosan U-t ismeri fel.

A hálózatcsalád implementálása során gyorsan rájönnénk a fenti leírás gyenge pontjára: a hálózatok megalkotása során implicit módon feltettük, hogy megoldottunk egy megoldhatatlan feladatot. A család elemeinek létrehozásához korlátlan számítási kapacitást feltételeztünk. Ezt kell tehát korlátoznunk ahhoz, hogy reális számítási modellt kapjunk.

Egy hálózatcsaládot uniformnak nevezünk, ha létezik olyan $log\ n$ tárat használó Turing-gép, amely az 1^n bemenetre a C_n hálózatot adja kimenetként. Ez tulajdonképpen azt jelenti, hogy a család minden tagja egy közös alapötlethez kapcsolódik, ugyanazt az algoritmust valósítják meg. Ez pontosan az a feltétel amire szükségünk van ahhoz, hogy a polinomiális számításokat azonosítani tudjuk a polinomiális hálózatokkal: egy L nyelv akkor és csak akkor dönthető el uniform polinomiális hálózatcsaláddal, ha $L \in P$. A P-beli nyelvek nyilván felismerhetők ilyen hálózatokkal, hiszen pont ilyenek segítségével látható be, hogy a változómentes hálózatok kiértékelése P-teljes. Másfelől tegyük fel, hogy L felismerhető uniform polinomiális hálózatcsaláddal. Ekkor egy Turing-gép az x bemenetről el tudja dönteni, hogy eleme-e L-nek úgy, hogy $log\ |x|$ tárral (ami legfeljebb polinomiális ideig futhat) előállítja $C_{|x|}$-et, majd polinom időben kiértékeli azt az x bemeneten.

A P osztályt gyakran azonosítják a hatékonyan megoldható problémák halmazával. Az irreális számítási erejű polinomiális hálózatok helyett az uniform polinomiális hálózatok fogalmának használatával egy sokkal hasznosabb modellhez jutottunk.

A Boole-hálózatokat nem csak a kombinációs logika implementálásához használjuk. A párhuzamos számítások vizsgálatánál is hasznunkra vannak. Vegyük észre, hogy a hálózatok értékének kiszámítása egyszerre több kapunál is folyhat, nem feltétlenül csak a definícióban szereplő szekvenciális módon. Az, hogy egy kapu igazságértékének meghatározása hanyadik lépésben történik valójában csak attól függ, hogy milyen hosszú úton érhető el a számára szükséges bemeneti kapukból. A hálózat számítási ideje ekkor a mélységével egyezik meg.

2.2. Párhuzamos számítások

Az FPGA-k nagyon nagy fokú párhuzamosságot tesznek lehetővé akár egyetlen chip-en is. Ezt kihasználva az FPGA-k rendkívül költséghatékonyak tudnak lenni. Egy olyan algoritmus, amely viszonylag kis mennyiségű adaton operál viszont nagy párhuzamosságot igényel, hatékonyabban és olcsóbban megvalósítható FPGA-n mint egy sok processzoros rendszeren. Vizsgáljuk meg a bonyolultságelmélet eszközeivel, hogy mikor tekintünk egy párhuzamos algoritmust hatékonynak.

A gyorsaság nyilvánvalóan fontos szempont a párhuzamos számítások terén. A célunk az, hogy az algoritmus drámaian gyorsabb legyen, mint a neki megfelelő szekvenciális algoritmus. Az implementálást szem előtt tartva arról sem feledkezhetünk meg, hogy nem használhatunk mértéktelenül sok processzort.

Példának nézzük meg a mátrixszorzás műveletét, amely nagyon jól párhuzamosítható. Ahogy azt a későbbi fejezetekben látni fogjuk, ez a Virtex-5 XC5VLX110T FPGA-ra készített implementáció hatékonyságához is hozzájárult.

Legyen adott két $n \times n$-es mátrix, ezeket jelölje A és B. Szeretnénk kiszámítani ezek $C = A \cdot B$

szorzatát, amelyhez a következő n^2 darab összeget kell kiszámítanunk:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj},\ i,j = 1,\ldots,n$$

Szekvenciálisan elvégezve ez egy $O(n^3)$ időbonyolultságú algoritmus. Az első észrevételünk lehet az, hogy az $A_{ik} \cdot B_{kj}$ szorzatok számítása egymástól független. Ezeket el tudjuk végezni párhuzamosan, egy lépésben, n^3 processzor felhasználásával. Nevezzük el a processzorokat: jelölje azt, amelyik a $A_{ik} \cdot B_{kj}$ szorzatot számítja (i,k,j). A szorzatok kiszámítása után az $(i,1,j)$ processzorok (összesen n^2) $n-1$ lépésben összeadják azt az n darab szorzatot, amelyek C_{ij}-het tartoznak. Összesen tehát a lépések száma n.

A bonyolultság máris n-re csökkent n^3-ről, de még ennél is jelentősen jobb eredményt tudunk elérni. A célunk itt az exponenciális felgyorsulás, vagyis a $log\ n$ (vagy legalább is $log^i\ n$) időbonyolultság.

Ezt megtehetjük úgy, hogy az összeadásokat egy bináris fa mentén hajtjuk végre. Az így szükséges párhuzamos lépések száma a bináris fa mélységével egyezik meg, ami $log\ n$. Az s. lépésben az $(i, 2^s\lfloor\frac{k}{2^s}\rfloor, j)$ processzor összeadja a saját tartalmát az $(i, 2^s\lfloor\frac{k}{2^s}\rfloor + 2^s, j)$ processzoréval, $s = 0,1,\ldots,\lceil log\ n\rceil - 1$. Végül itt is az $(i,1,j)$ processzorok tartalmazzák a C_{ij} értékeket. Az idő ebben az esetben $log\ n + 1$, a processzorok száma n^3.

Párhuzamos algoritmusok esetén a polilogaritmikus ($log^i\ n$) időbonyolultság elérése igen jelentős gyorsulást jelent. A bonyolultság ekkor exponenciálisan csökken a szekvenciális esethez képest, vagyis a különbség a polinomiális és exponenciális szekvenciális algoritmusok közötti különbséghez hasonlítható. A használt processzorok számának polinomiális volta szintén fontos kitétel, ugyanis az exponenciálisan sok processzor rendelkezésre állása nem valószerű.

A processzorok számának csökkentése nyilvánvalóan fontos szempont a párhuzamos algoritmusok tervezésekor. A mátrixszorzás példájánál maradva, mivel a szekvenciális algoritmus bonyolultsága $O(n^3)$, a párhuzamos algoritmus által végzett munkának szintén legalább ennyinek kell lennie. Munka alatt itt az egyes processzorok által végrehajtott lépések számának összegét értjük. Egy párhuzamos algoritmus által végzett munka nem lehet kevesebb, mint a legjobb szekvenciális algoritmus által végzett munka. Ennek az oka az, hogy minden párhuzamos algoritmus szimulálható olyan szekvenciálissal, amely ugyanannyi munkát végez. Megjegyezzük, hogy az itt használt mátrixszorzó algoritmus nem a legjobb (a későbbi fejezetekben szót ejtünk gyorsabb változatokról is). A példa kedvéért most tekinthetjük úgy, hogy az $O(n^3)$ bonyolultságú algoritmust kívánjuk párhuzamosítani.

Ha a párhuzamos idő $log\ n$ és legalább n^3 munkát végeznünk kell, akkor legalább $\frac{n^3}{log\ n}$ processzorra szükségünk van. Úgy kell lecsökkentenünk a processzorok számát n^3-ről $\frac{n^3}{log\ n}$-re, hogy a párhuzamos idő közben ne nőjön jelentősen.

Ezt megtehetjük úgy, hogy az n^3 szorzatot nem egyetlen párhuzamos lépésben végezzük el, hanem $log\ n$ „műszakban”, műszakonként $\lceil\frac{n^3}{log\ n}\rceil$ processzort használva. Ugyanezt tesszük az első $log\ log\ n$ párhuzamos összeadási lépésben is, amikor normálisan több, mint $\frac{n^3}{log\ n}$ processzort használnánk. A processzorok száma így a kívánt $\frac{n^3}{log\ n}$, a párhuzamos idő pedig nem több, mint $2log\ n$. Ez (egy kettes

tényezőtől eltekintve) mindkét szempontból optimális. Ezt a fajta „processzorműszakokkal" való optimalizálását a processzorszükségletnek és a párhuzamos időnek Brent-elvnek nevezzük.

A példa során mindvégig n függvényeként fejeztük ki a processzorok számát. Ez nyilvánvalóan nem életszerű, hiszen egy adott gépnek csak rögzített számú processzora van. Az FPGA-k flexibilitása itt is nekünk kedvez, hiszen a konfigurálás során annyi párhuzamos szálat definiálunk amennyire szükségünk van. Természetesen ebben az esetben is egy konkrét konfiguráció párhuzamos szálainak száma rögzített lesz, és minden problémamérethez új konfiguráció létrehozása nem lenne praktikus. A párhuzamos algoritmusunkat mindenképpen meg kell tudnunk valósítani a konkrét gép processzorainak számától függetlenül. Tegyük fel, hogy R processzorunk van, ahol R sokkal kisebb, mint $\frac{n^3}{log\ n}$. A leírt, kellően optimális algoritmust könnyen az elérhető hardware-hez tudjuk igazítani. Úgy kell szerveznünk, hogy a párhuzamos lépéseket az R darab processzor $\lceil \frac{n^3/log\ n}{R} \rceil$ műszakban végezze el. Az idő ekkor $\frac{2n^3}{R}$, ami R processzorral való párhuzamosítás esetén optimális. Ezzel egy újabb indítékot is láthatunk az algoritmus processzorigényének minimalizálására. Ha az algoritmus ezen igénye nagy, akkor az implementálás során a párhuzamos idő fog nagyon megnőni ahogy a fix processzorszámhoz igazodunk.

A párhuzamos számítások bonyolultságának két fontos mértéke a párhuzamos idő és a munka. Ezeket a fogalmakat összekapcsolhatjuk a Boole-hálózatokkal. Legyen C egy uniform hálózatcsalád, $f(n)$ és $g(n)$ pedig az egész számokat az egész számokba képező függvények. Ekkor C párhuzamos ideje legfeljebb $f(n)$, ha C_n mélysége minden n-re legfeljebb $f(n)$. A C által végzett munka pedig legfeljebb $g(n)$ ha C_n mérete minden n-re legfeljebb $g(n)$. A fogalmaknak ezt az azonosítását azért tehetjük meg, mert bár a Boole-hálózatok programozása ugyanolyan kényelmetlen, mint a Turing-gépeké, be tudjuk látni, hogy a hálózatok szoros kapcsolatban állnak a gyakorlatban használt, a valósághoz igen hűséges modellekkel. Ahogy a Turing-gépekről is be tudtuk látni, hogy könnyen megfeleltethetők a RAM-gépeknek, úgy a Boole-hálózatok és a PRAM (párhuzamos RAM) gépek között is egyszerűen megoldható az átjárás.

Legyen $PT/WK(f(n), g(n))$ az összes olyan $L \subseteq \{0,1\}^*$ nyelv osztálya, amely felismerhető $O(f(n))$ párhuzamos idejű és $O(g(n))$ munkát végző uniform hálózatcsaláddal. A szekvenciális számításokhoz kötődő osztályok általában csak az időbonyolultságra vagy csak a tárbonyolultságra tesznek feltételt. Párhuzamos számítások esetén egyszerre korlátozzuk a párhuzamos időt és a munkát is. Ahogy azt a példánál is láttuk, a két fogalom nem független egymástól, egy nagy munkaigényű algoritmus kevés processzoron történő implementálása nagy párhuzamos időhöz vezet. Gyakorlati szempontból is hasznos lehet ez a kettősség. Ahelyett, hogy csak egy dologra összpontosítanánk a többi figyelmen kívül hagyásával, egy gondosabban megválasztott, a valósághoz közelebb álló modellt használó osztályt tudunk definiálni.

A példa bemutatása során már tárgyaltuk, hogy milyen elvárásaink vannak egy párhuzamos algoritmussal szemben. Ezek formalizálásával adjuk meg a következő osztályt:

$$NC = PT/WK(log^k\ n, n^k)$$

Az NC osztály, amely a „Nick's Class" rövidítése Nick Pippenger után, pontosan a példa során megfogalmazott követelményeknek megfelelő problémákat tartalmazza: azon nyelvek osztálya, amelyek felismerhetők polilogaritmikus mélységű, polinomiális méretű uniform hálózatcsaládal. Ez egybeesik a

polilogaritmikus párhuzamos idejű, polinom sok processzort használó PRAM-programmmal eldönthető nyelvek osztályával.

Az NC osztály a P-hez hasonlítható abban az értelemben, hogy mindkettőt szokás a „hatékonyan megoldható" problémákkal azonosítani. A szekvenciális számításoknál a P ilyen jellegű kitüntetése széles körben elfogadott. Elméleti szempontból az osztály nagyon robosztus, sokféle átalakítással szemben ellenálló, gyakorlati szempontból pedig megtapasztalható, kézzelfogható különbség van a P-beli és a nem P-beli problémák között. A párhuzamos számítások esetén az NC osztály sikere nem ennyire osztatlan. Gyakorlati szemszögből a különbség a polilogaritmikus és az alacsonyabb kitevőjű polinomiális párhuzamos idejű algoritmusok között gyakran nincs olyan drámai, mint a polinomiális és az exponenciális szekvenciális algoritmusok között. Az NC definíciójának másik hátránya, hogy nyelvek osztálya, vagyis eldöntési problémák halmaza. Ahogy azt a példa is mutatta, a párhuzamos számításokat inkább szeretnénk valamilyen terjedelmesebb kimenet kiszámítására használni. Szekvenciális esetben a kétfajta probléma között egyszerű volt a megfeleltetés, párhuzamos esetben azonban ez nem ilyen egyértelmű. Ha csak nagy kimenettel rendelkező problémákkal kívánunk foglalkozni, akkor természetesen hasznos az NC osztályt inkább a polilogaritmikus párhuzams idejű, polinom sok munkával kiszámítható függvények osztályaként definiálnunk.

Bizonyos problémák esetén az NC részletesebb felbontása is érdekes lehet számunkra. Értelmezzük NC-t részosztályok uniójaként:

$$NC^j = PT/WK(log^j\ n, n^k)$$

NC ekkor NC^j összes nemnegatív j-re történő egyesítése. NC^j az NC osztály legfeljebb $O(log^j\ n)$ párhuzamos idejű része. Ebben a felbontásban a munkát nem korlátozzuk, leszámítva természetesen, hogy továbbra is polinomiálisnak kell lennie. Egy NC-probléma részletesebb vizsgálata során érdemes lehet megvizsgálni, hogy melyik NC^j részosztályba esik. Elsősorban az NC^0, NC^1 és NC^2 osztályokba esés az, amit szigorúbb értelemben „hatékony" párhuzamos számításnak tekintünk. NC^2-be esik például a gráfbeli elérhetőség problémája. Ismert az is, hogy $NC^1 \subseteq L \subseteq NL \subseteq NC^2$, ahol L és NL a determinisztikus és nemdeterminisztikus logaritmikus tárkorlátos Turing-géppel megoldható problémák osztályai.

Vegyük észre, hogy az NC^j osztályok egy lehetséges hierarchiát alkotnak. Szekvenciális esetben a P osztályon belül a $TIME(n^j)$ osztályok egy valódi, nem összeomló hierarchiát alkottak. Ez azt jelenti, hogy a hierarchia minden szintje szigorúan bővebb, mint az alatta álló szintek, nincs olyan kitevő ami felett a szintek már egybeesnek, másszóval a hierarchia összeomlik. Ekkor minden j-re létezik olyan P-beli nyelv, amely $O(n^j)$-nél kevesebb idő alatt nem ismerhető fel. A polinomiális hierarchiával ellentétben az NC-hierarchiáról nem tudjuk, hogy valódi-e, a sejtés bizonyítása nyitott kérdés.

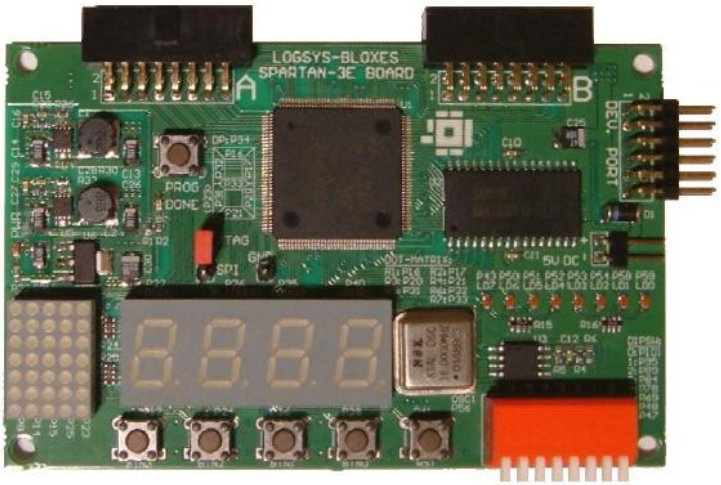
Szintén nyitott kérdés az NC osztály P-hez való viszonya. Mivel NC definíciójában kikötöttük a polinomiális korlátot a végzett munkára, ezért világos, hogy $NC \subseteq P$. A számunkra érdekes kérdés az $NC = P$ igazsága. Ez a párhuzamos számítások terén a P vs. NP kérdés megfelelője, amely a bonyolultságelmélet központi problémája és a jelenleg ismert egyik legfontosabb nyitott kérdés, a hét

Millenium Prize Problem egyike. Mindkét kérdésben arra vagyunk kíváncsiak, hogy egy olyan halmaz, amelyet elfogadunk kielégítően megoldhatónak (szekvenciális esetben a P, párhuzamos esetben az NC) megegyezik-e azzal a nagyobb halmazzal, amely a törekvéseink természetes korlátait jelenti (szekvenciális esetben az NP-vel, párhuzamos esetben a P-vel). Szintén a P vs. NP-hez hasonlóan a tapasztalat és az elért részeredmények inkább a nemleges válasz felé mutatnak. Az igenlő válasz azt jelentené, hogy minden polinom időben megoldható probléma hatékonyan párhuzamosítható, ami igen figyelemre méltó eredmény lenne. A gyakorlat nem azt mutatja, hogy ez a szerencsés helyzet állna fenn, néhány meglehetősen egyszerű P-beli probléma (például a folyamokhoz kapcsolódó maximális folyam megtalálásának problémája) NC-beli megoldására tett kísérlet sorozatos kudarcba fulladt. Indokoltnak tűnik feltételezni, hogy $NC \neq P$ és hogy léteznek „eredendően szekvenciális" problémák. Ennek bizonyítása nem ismert, de ha valóban léteznek, akkor ennek belátását a P-teljes problémáknál érdemes kezdeni. Tudjuk azt, hogy ha egy L nyelv visszavezethető $L' \in NC$-re logaritmikus tárkorlátos visszavezetéssel, akkor $L \in NC$. Ha egy P-teljes problémára belátnánk, hogy NC-beli, akkor ebből rögtön következne, hogy $P = NC$. Világos tehát, hogy a P-teljes problémáknak van a legnagyobb esélye arra, hogy eredendően szekvenciálisak legyenek. A P-teljes problémák hatékony párhuzamosítása tehát úgy tűnik nem megoldható, pedig több olyan probléma is szerepel közöttük, amelyek NC-beli párhuzamos megoldása igen hasznos lenne (pl. lineáris programozás, Horn-SAT probléma, LZW-tömörítés).

Ebben a fejezetben néhány olyan bonyolultságelméleti fogalommal ismerkedtünk meg, amelyek az FPGA-k konfigurálása során hasznunkra lehetnek. A Boole-hálózatok mind a kombinációs logikával, mind a párhuzamos számítások modelljének definiálásával kapcsolatban állnak. A mátrixszorzás példáján keresztül megmutattuk milyen feltételek teljesülése esetén lesz egy algoritmus igazán jól párhuzamosított. Ezen követelmények formalizálásával definiáltuk az NC osztályt, amely a párhuzamos számítások bonyolultságának fontos osztálya. Röviden taglaltuk NC felépítését és viszonyát a kiemelkedő fontosságú P osztállyal. A következő fejezet FPGA-n készült implementációk ismertetésével foglalkozik.

3. Spartan-3E implementációk

A következő fejezetek Spartan-3E XC3S250E és Virtex-5 XC5VLX110T FPGA-ra készült implementációkat ismertetnek. Ebben a fejezetben a Spartan-3E FPGA-ra készült konfigurációkat tárgyaljuk. Mindkét eszközt a Xilinx gyártja. A kisebb XC3S250E kiválóan alkalmas az FPGA-k nyújtotta alapvető lehetőségekkel való megismerkedésre. Az FPGA egy Logsys Spartan-3E kártyára van integrálva. Kis mérete miatt elsősorban olyan feladatok esetén igazán hatékony amikor viszonylag kis adatmennyiségen kell sok operációt elvégezni. Technikai paraméterei lehetővé teszik, hogy rendkívül költséghatékony megoldást tudjon nyújtani bizonyos számítási igényekre.



1. ábra. Logsys Spartan-3E kártya

A kártya jelen munka szempontjából érdekes paraméterei:

- A számítás legfőbb eszköze a négy bemenetű look-up table-ök (LUT-ok) tömbje. Ezek 16 bit tárolására képesek, amelyek közül a négy bites inputot címként értelmezve választanak ki egyet kimenetként. A LUT-ok kettessével vannak Slice-okba szervezve, a Slice-ok pedig négyessével alkotnak egy-egy CLB-t (Configurable Logic Block). Az FPGA-n összesen 612 CLB található, ami 2448 Slice-nak és 4896 LUT-nak felel meg. Minden LUT-hoz tartozik egy flip-flop is, amiből összesen szintén 4896 található az eszközön. A Slice-ok két csoportba oszthatók, SLICEM és SLICEL csoportokra. A SLICEM csoportba tartozó Slice-ok (összesen a LUT-ok fele) használható 16 bites memória vagy shift regiszterként is.
- Memóriák: 12 darab 18 kilobites blokk-RAM, egy 128 kilobyte-os SRAM.
- 12 darab 18×18 bites dedikált szorzó.
- 4 darab DCM (Digital Clock Manager) modul
- 16 MHz-es oszcillátor.

- Logsys fejlesztői kábel csatlakozó.
- Input perifériák: 5 darab nyomógomb, 8 darab DIP-kapcsoló
- Output perifériák: 8 darab LED, 4 digites hétszegmenses kijelző, 7×5-ös pontmátrix kijelző.

A kártya egy Logsys fejlesztői kábellel csatlakozik a számítógéphez. A kártya konfigurálása a Logsys Development Environment program használatával történik.

Három implementációt fogunk röviden bemutatni, amelyek az XC3S250E FPGA-ra készültek. Mindhárom olyan algoritmust valósít meg, amely FPGA-n nagyon hatékonyan hozható létre, és olyan ötleteket is felhasznál, amelyek később a részletesebben ismertetett XC5VLX110T implementációban is felhasználásra kerülnek. Mindhárom konfiguráció olyan számok szorzását valósítja meg, amelyek szorzása hosszuk miatt a dedikált szorzóegységekkel már nem hatékony.

Elsőként a Partial Product módszerrel és implementációjával ismerkedjünk meg. Az implementáció 64×64 bites számok szorzását végzi. Az eljárás egyfajta „osszd meg és uralkodj" elvet használ: a számokat mindig kétjegyű számokként értelmezi, amelyeket „számjegyenként" szoroz össze. Ezt a számjegyenkénti szorzást rekurzívan hajtja végre, vagyis a számjegyeket szintén kétjegyű számokként értelmezi, amelyek tovább bont egészen addig amíg ez már nem lehetséges. A szorzás során a következő összefüggést használja fel:

Legyen $x = x_1a + x_0$, $y = y_1a + y_0$ és $z = x \cdot y$, $a \in \mathbb{N}, x_1, x_0, y_1, y_0 \in \{0, 1, \ldots a-1\}$. Ekkor

$$z = (x_1y_1)a^2 + (x_1y_0)a + (x_0y_1)a + x_0y_0$$

Vagyis a szorzathoz kiszámításához négy szorzást kell elvégeznünk fele olyan hosszú számokon. Vegyük észre, hogy mivel 64 bites számokkal dolgozunk, ezért a értéke a bontás minden szintjén kettő hatványa lesz. Ezt kihasználhatjuk, hiszen binárisan ábrázolt számok esetén a kettő hatvánnyal való szorzás megfelel a bitek shiftelésének.

A számok további kettéosztása már nem éri meg amikor az eredményt egy lépésben el tudjuk végezni. Az XC3S250E FPGA 4-LUT-okat használ, vagyis bármilyen legfeljebb négy bit bemenetű logikai függvény elvégzésére felkonfigurálható. Vegyük észre, hogy két két bites szám összeszorzása megfeleltethető négy olyan logikai függvénynek, amelyek az összesen négy bit inputra mind kiadnak egy bitet a szorzatból. Formálisabban legyen $a = 2a_1 + a_0$, $b = 2b_1 + b_0$, $c = 8c_3 + 4c_2 + 2c_1 + c_0$, $c = a \cdot b$ és legyen $\sigma_3, \sigma_2, \sigma_1, \sigma_0$ négy olyan logikai függvény, amelyekre

$$\sigma_i(a, b) = c_i, \ i \in \{0, 1, 2, 3\}$$

teljesül. Ekkor a σ_i-k voltaképpen egy apró szorzótáblát hoznak létre. A függvények egy-egy LUT-tal megvalósíthatók, vagyis négy LUT használatával két két bites szám egy lépésben összeszorozható.

Ezen alapelemekből kiindulva modulok hierarchiáját hozzuk létre. Minden szinten az összeszorzott számok hossza a duplájára nő. Minden szinten egy modul négy darabot használ az egy szinttel alatta álló modulból. Egy modulnak a saját kimenete kiszámításához elég a felhasznált alacsonyabb szintű modulok kimeneteit megfelelően összekötnie, a szükséges összeadások és eltolások figyelembe vételével.

Az így kapott modul egy Boole-hálózatot alkot. A számítás ideje megegyezik a hálózat mélységével, amely az input méretében logaritmikus. A hálózat mérete pedig polinomiális, egészen pontosan négyzetes: mivel minden modul négyet használ az alatta álló szint moduljából, ezért az input méretének megduplázásával a hálózat mérete a négyszeresére nő. A négyzetes méret azért sem meglepő, mert a szorzás jól ismert négyzetes bonyolultságú algoritmusát párhuzamosítja. Az előző fejezet fogalmait tekintve láthatjuk, hogy a Partial Product eljárás NC-beli.

Következőnek nézzük meg a Computed Partial Product eljárást és implementációját. A megvalósítás itt is 64×64 bites számok szorzását végzi. Ez az algoritmus arra az észrevételre épül, hogy egy tetszőleges hosszú z számot megszorozni egy két bites számmal nagyon egyszerű, csupán négy lehetőség közül kell választanunk:

- A szorzó értéke nulla: ekkor a szorzat nulla.
- A szorzó értéke egy: ekkor a szorzat z.
- A szorzó értéke kettő: ekkor a szorzat $2z$, ami z egy bittel való shiftelése.
- A szorzó értéke három: ekkor a szorzat $3z = 2z + z$, vagyis az előző módon előállított érték és z összege.

A négy lehetőség előállítását könnyen összeszervezhetjük egyetlen modullá. A két bites szorzó használható egy multiplexer vezérlő jeleként. Ilyen modulokból 32 használatával össze is tudjuk szorozni a kívánt 64 bites számokat. A 32 modul a kimenetét egy lépésben kiszámítja, az eredmények összeadásával pedig meg is kapjuk a kívánt szorzatot. Természetesen itt is figyelnünk kell a részeredmények megfelelő eltolására. Az összeadások elvégezhetők egy bináris fába rendezve, úgy ahogy az előző fejezet mátrixszorzást végző példájában is tettük.

Az így kapott hálózat mélysége szintén logaritmikus a bemenet hosszában, a számítás sebessége a használt méret mellett a Partial Product eljárásával megegyező.

Az eljárás előnye a Partial Product-tal szemben a mérete. A használt LUT-ok száma jelentősen kevesebb, ugyanis a szorzás elvégzése ebben az algoritmusban nem a LUT-ok feladata. A Computed Partial Product egy helytakarékosabb megoldást jelent: a Partial Product méretének további növelését már akadályozná az FPGA-n található LUT-ok száma, a Computed Partial Product ezzel szemben probléma nélkül tovább növelhető lenne.

Az utolsó implementáció amit bemutatunk a Karatsuba-szorzás valósítja meg. Az előző két eljárás a szokásos „kézi” szorzás algoritmusát párhuzamosították. A Karatsuba-szorzás ezzel szemben magán a szorzás algoritmusán javít. Szekvenciális algoritmusként is az időbonyolultsága jobb, mint az eddig használt algoritmusé, $O(n^{log_2\ 3})$ az $O(n^2)$ helyett. A Partial Product eljáráshoz hasonlóan itt is az összeszorzandó számok kétjegyűként való értelmezésével kezdünk. Mint ahogy azt láttuk, a „számjegyekkel” négy szorzást kellett elvégeznünk. A Karatsuba-algoritmus bemutatja azt, hogy ez valójában megoldható három szorzásból, néhány plussz összeadásért „cserébe”.

Legyen most is $x = x_1 a + x_0$, $y = y_1 a + y_0$ és $z = x \cdot y$, $a \in \mathbb{N}, x_1, x_0, y_1, y_0 \in \{0, 1, \ldots a-1\}$. Ekkor

$$z_2 = x_1 y_1$$
$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$
$$z_0 = x_0 y_0$$
$$z = z_2 s^2 + z_1 a + z_0$$

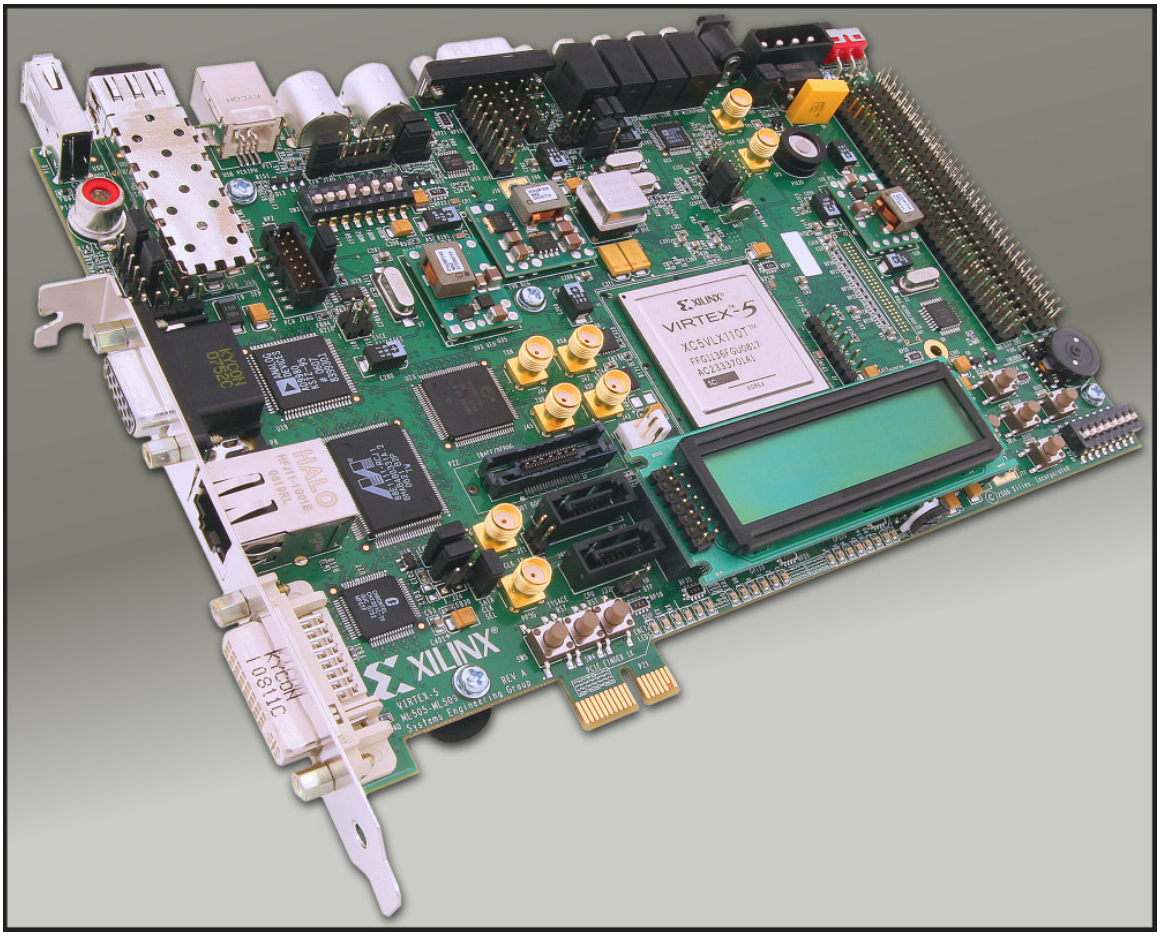
Az algoritmus párhuzamos megvalósítása a Partial Product eljárással analóg módon történik. Legalsó szinten, két bites számok szorzása esetén apró szorzótáblákat definiálunk. Ezután itt is modulok hierarchiáját definiáljuk, de ebben az esetben egy modul csak hármat használ az egy szinttel alatta álló modulokból. A Karatsuba-szorzás a Computed Partial Product-hoz hasonlóan kisebb méretű hálózatot eredményez, mint a Partial Product implementációjáé, de a számítás sebessége nem változik.

Nagyon hasonló ötlet alapján született a Strassen-algoritmus is, amely a mátrixszorzás műveletét gyorsítja a Karatsuba-szorzással analóg módon.

Az XC3S250E FPGA-ra készült implementációk ismertetésével példákat láthattunk az FPGA-k által nyújotta alapvető lehetőségekre. A következő, nagyobb lélegzetvételű fejezetek az XUPV505-LX110T platformon implementált kutatást ismertetik.

4. Virtex-5 implementációk

Ebben és a következő a fejezetben egy Virtex-5 XC5VLX110T FPGA-ra készült implementációt mutatunk be. A chip egy XUPV505-LX110T fejlesztői platformon található. Ez az eszköz paramétereit tekintve jelentősen meghaladja az XC3S250E nyújtotta lehetőségeket. Ennek megfelelően az itt leírt konfiguráció sokkal összetettebb az előző fejezetben leírtaknál. A kutatásból, amely az implementációhoz vezetett TDK-munka és publikáció egyaránt készült. A hardware azon elemei, melyek a konfiguárció számítási tevékenységében vesznek részt, a belőlük felépülő modulok ismertetése során kerülnek leírásra. Itt röviden ismertetjük azon elemeket és a felhasználásukkal készült modulokat amelyek a konfiguráció egyéb szükséges tevékenységeit (pl. kommunikáció) végzik.



2. ábra. XUPV505-LX110T fejlesztői platform

Használt hardware elemek:

- 148 darab 36 kilobites blokk-RAM.
- 64 darab 25 × 18 bites DSP48E dedikált szorzó.
- 6 darab CMT (Clock Management Tile). Mindegyikben két DCM (digital clock manager) és egy PLL (phase-locked loop) található.

- 100Mhz-es oszcillátor.
- RS-232 soros port csatlakozó.

Az eszköz az RS-232 soros porton keresztül kommunikál a külvilággal. A kommunikációt a *serialHandler* nevű projektben található modulok valósítják meg. A soros porton való kommunikáció formátuma:

- A használt átviteli sebességek 9600 baud (bit/s), 19200 baud, 38400 baud és 115200 baud.
- Az adatbitek száma 8.
- Nem használunk paritásbitet.
- Egy stop bitet használunk.
- Nem használunk Flow Control jeleket.

A *serialHandler* projekt négy modult tartalmaz. A *baudGenerator* és *oversampler* modulok a szükséges frekvenciák előállítását végzik, a *transmitter* és *receiver* modulok pedig a formátumnak megfelelő kommunikációról gondoskodnak.

A *baudGenerator* modul a fent leírt négy szabványos frekvencia egyikét állítja elő. A modul két bemenetet használ: egy 17 bites jelet, amelyen a négy frekvencia értékének egyikét várja (ha nem a négy érték közül valamelyiket kapja, akkor az alapértelmezett 9600Hz frekvenciát állítja elő) és egy 100MHz-es órajelet. Egyetlen kimenete az input által meghatározott frekvenciának megfelelő jel. A kívánt frekvenciát egyszerűen az inputként kapott 100MHz-es jel megfelelő osztásával éri el: ha a kívánt frekvencia f, akkor az outputon a 100Mhz-es órajel minden $\lceil \frac{100,000,000}{f} \rceil$-edik leütésekor jelenik meg 1-es bit.

A *baudGenerator* modult a *transmitter* modul használja, amely az FPGA-ból a külvilág felé irányuló jeleket továbbítja. A *transmitter* inputként fogadja a 100MHz-es órajelet és a 17 biten ábrázolt használt frekvenciát, amit átad a *baudGenerator* modulnak. Ezen kívül szintén inputként fogad egy nyolc bit széles adatjelet, ami a kiküldendő adatot jelenti, és egy start jelet. Egyetlen output-ja a fent leírt szabványnak megfelelő kimenő jelsorozat. A konfiguráció létrehozása során gondoskodnunk kell arról, hogy ez a jel a soros port kimenő vonalához legyen kötve. A kimenő jelen a modul egészen addig egyes biteket küld, amíg a start jel értéke igaz nem lesz, jelezve, hogy az adatjelen lévő nyolc bitet szeretnénk kiküldeni a külvilágnak. A start jel hatására a modul a *baudGenerator* által előállított frekvencia segítségével elkezdi továbbítani az adatokat a formátumnak megfelelően: minden nyolc bit adatot tíz bitként közvetít, az elején egy nulla, a végén egy egyes bittel közrefogva. A modul használata során figyelnünk kell arra, hogy ha a start jelet a beállított adatátviteli frekvenciánál nagyobb frekvenciával aktiváljuk, akkor a jeleket fogadó gépen hibás adatok fognak megérkezni.

Az *oversampler* modul működése a *baudGenerator* moduléval megegyező. Mind az bemenő jelei, mind a kimenő jele formailag megegyezik a *baudGenerator* neki megfelelő jeleivel. Az egyetlen különbség az, hogy a modul nem az inputként megkapott f frekvenciát állítja elő, hanem annak 16-szorosát. A kimenő jelet szintén a bejövő 100MHz-es órajel osztásával állítja elő, de most minden $\lceil \frac{100,000,000}{16f} \rceil$ leütésenként jelenik meg rajta 1-es bit.

Az adatátviteli frekvencia 16-szoros tovább osztását a külvilágból az FPGA irányába bejövő adatok fogadását végző *receiver* modul használja. Feladatából adódóan a *receiver* működése a *transmitter* inverze. Az *oversampler* modulnak átadott 100MHz-es órajel és 17 bites frekvencia érték mellett input vonala még a külvilág irányából érkező adatfolyam. A konfiguráció létrehozása során ügyelnünk kell arra, hogy ez a vonal a soros port bejövő jeléhez legyen csatlakoztatva. A modul két output jele a nyolc bites adat jel és egy „adat érkezett" jel. A 16-szoros frekvenciára azért van szükség, hogy a modul nagyobb stabilitással tudjon dolgozni. Mivel a soros porton keresztüli kommunikáció aszinkron, ezért hiába rögzített az átviteli sebesség, meg kell tudnunk határozni az adatok beérkezésének pontos kezdetét. A *receiver* modul 16-szoros sebességgel figyeli a csatornát, így nagyobb biztonsággal fel tudja ismerni ha a porton keresztül adatok érkeznek be. Amikor nincs adatforgalom a csatornán keresztül egyes bitek érkeznek. Az érkező adatok mindíg nullás bittel kezdődnek, a modul tehát arra figyel, hogy mikor lát a csatornán megfelelő hosszúságú nulla sorozatot. Ha ez a felismerés megtörtént, akkor a formátumnak megfelelően elkezdi összegyűjteni a beérkező biteket. Amikor nyolc adatbit megérkezett, akkor az adatokat kiteszi az output jelre és 1 értéket ad az „adat érkezett" jelnek.

A fejlesztés során az eszköz a soros portot egy PC-vel való kommunikálásra használta, amin a Tera Term Pro és Advanced Serial Port Terminal szoftverek kezelték az adatok fogadását és küldését.

A következő fejezet során ismertetett implementáció fő adattároló eszköze egy a fejlesztői platformhoz kapcsolt 256MB-os DDR2 SODIMM. A DDR2-es memória vezérlését megoldó modul leírása szintén a következő fejezetben található. A modulnak a helyes működéshez több különböző órajelre is szüksége van. Ezek előállítását végzi a $clockStuff$ projektben megtalálható modulok. A DDR2-vezérlő modulhoz szükséges órajelek:

- $clk0$: a felhasználói interfész jeleinek előállításához használt órajel. Ennek a jelnek a segítségével állítja elő a modul a fizikai buszokhoz kötődő jelek többségét, például az adatbuszon, a címbuszon és a parancsbuszon megjelenő jeleket. Szintén ezt az órajelet használják a write enable jelek, a read data valid jel, valamint az FPGA-n létrehozott felhasználói logika többsége, például az eszközzel való kommunikációra használt FIFO-k.

- $clk90$: A $clk0$ órajel 90 fokkal fázis-eltolt megfelelője. A memória írás során használatos, az írást végző fizikai réteg órajele. Szintén ez az órajel szükséges az adat-FIFO kezeléséhez a DDR2-es memória oldaláról.

- $clkdiv0$: A $clk0$ órajel kettővel elosztott változata. Ez az órajel felelős a memória inicializálásáért, valamint a fizikai rétegben a memória olvasás időzítésének megfelelő kalibrálásáért.

- $clk200$: kötelezően 200MHz-es órajel a Virtex-5 FPGA-k IDELAYCTRL blokkjaihoz.

A fejlesztői platformon található oszcillátor 100MHz-es órajelet szolgáltat az FPGA-nak. A következő fejezet implementációjának helyes működéséhez a DDR2 RAM-ot 200MHz-en kellett üzemeltetnünk. Ehhez az FPGA-n elő kell állítunk a megfelelő órajeleket, ahol $clk0$ és $clk200$ 200MHz-es, $clk90$ szintén 200MHz-es de 90 fokkal eltolt fázisban van, $clkdiv0$ pedig 100MHz-es.

A *clockStuff* modul ezen jelek előállítását végzi. A modul bemenetként fogadja az alapértelmezett 100MHz-es órajelet valamint egy reset jelet, kimenetként pedig szolgáltatja a négy előállítani kívánt órajelet és egy locked jelet, amely az órajelek helyességét jelzi. A modul az FPGA-n található Clock Management Tile elemeket használja arra, hogy elvégezze a szükséges órajelduplázást, fáziseltolást és kettéosztást. A memóriakezelő modul elvárja, hogy az órajelek a BUFG primitívet használva buffereltek legyenek, a *clockStuff* modul erről is gondoskodik. Két almodulból áll, a *clockDoubler* és a *phaseShifter* modulokból.

A *clockDoubler* modul végzi az órajel duplázását. A modul inputként fogadja a 100MHz-es órajelet és 200Mhz-es jelet állít elő belőle.

A *phaseShifter* modul neve ellenére nem csak a fáziseltolást végzi. Ez a modul inputként már a *clockDoubler* által létrehozott 200MHz-es órajelet fogadja, és előállítja belőle a *clk*0 és *clk*200 jeleket (ezekhez a jelekhez nem módosítja az inputként kapott jelet), a *clk*90 90 fokkal fáziseltolt, szintén 200Mhz-es jelét, valamint a *clkdiv*2 100MHz-es jelét (az input jel kettéosztásával). Az órajel ilyen módon történő megduplázására majd kettéosztására azért van szükség, mert a CMT teljesen „lefoglalja" az oszcillátor által szolgáltatott 100MHz-es jelet, vagyis miután felhasználtuk a *clockDoubler* modul bemeneteként máshol már nem használható.

A hardware és a működéshez szükséges segédmodulok megismerése után lássuk a dolgozat legösszetettebb felépítésű és legbonyolultabb munkáját végző implementációját.

5. Mátrixok moduláris hatványozása FPGA-n

Az FPGA-k (field-programmable gate array) több különleges opcióval is szolgálnak a számítások elvégzéséhez. Léteznek olyan algoritmusok, amelyek implementálása tradicionális architektúrákon nem praktikus, viszont az FPGA egyedi tulajdonságait kihasználva kényelmesen és költséghatékonyan megvalósíthatók. Az eszközön található look-up table-ök tömbje flexibilis megoldást nyújt az egyedi igényeknek megfelelő logika létrehozásához és természetes módon támogatja a nagyfokú párhuzamosságot.

A mátrixműveletek gyors elvégzése gyakorlati szempontból is nagy figyelmet érdemel. A mátrixokkal való számítások felgyorsítása számos alkalmazásban kap szerepet.

A mátrix algoritmusok implementációinak gyorsítása történhet „software" szempontból, magának az algoritmusnak a javításával, vagy „hardware" szempontból, gyorsabb vagy más felépítésű architektúra felhasználásával.

Elméleti jellegű javítása a mátrix algoritmusoknak például a Strassen algoritmus [1] és a Coppersmith-Winograd algoritmus [2]. A mátrix szorzás naiv algoritmusa egy jól ismert $O(n^3)$ bonyolultságú algoritmus. Strassen algoritmusa egy a Karatsuba-szorzáséhoz hasonló ötletet használ. $O(n^{lg\ 7})$ időbonyolultságot ér el a mátrixok részmátrixokra bontásával. Ezek összeszorzásának különleges elrendezésével eléri, hogy kevesebb szorzást kelljen végeznie, mint a klasszikus algoritmusnak. [3] a Cell Broadband Engine-en való implementálására irányuló kutatást ír le. Strassen algoritmusáról és a projekthez való kapcsolatáról a 7. alfejezetben szólunk. A Coppersmith-Winograd algoritmus kombinálja Strassen ötletét a Salem-Spencer tétellel, ezzel tovább javítva a bonyolultságon, $O(n^{2,376})$ eredményt érve el. [4] leírja és összehasonlítja ezen algoritmusok megvalósításainak teljesítményét.

Számos kutatás irányult különböző mátrix műveletek különböző architektúrákon való implementálására. [5] és [6] mátrix invertáló hardware konstrukcióról ír FPGA-n.

Jelen munka speciális mátrix szorzó eljárás struktúráját és paramétereinek megválasztását írja le. Hasonló jellegű munkákat taglal [7] és [8], melyek lebegőpontos számokkal végzett mátrix szorzást végző FPGA struktúrákat tárgyalnak. [9] digitális jelfeldolgozáshoz fejleszt alkalmazást FPGA-ra. [10] a mátrix szorzás FPGA-n való felgyorsítását írja le.

A munka témáját képező kutatás egy álvéletlen számsorozat generáló eljárás, amelynek része nagy mátrixok nagyon magas hatványra emelése. Ez több lehetőséget is nyújt az FPGA implementáció optimalizálására, amely egy rendkívül gyors konstrukciót eredményez.

Ehhez kapcsolódóan ismertetünk egy Virtex-5 XC5VLX110T FPGA-ra fejlesztett konfigurációt, amely két 896×896 méretű mátrixot szoroz össze. A szorzást mod 4 maradékosztály gyűrű felett végezzük el. Ezt szem előtt tartva kihasználjuk, hogy a hardware 6-LUT-okat tartalmaz és olyan modult írunk le, amely $\mathbb{Z}_4^{28}$-ből vett vektorok belső szorzatát egyetlen órajel alatt számítja ki 100MHz-es órajel frekvencia mellett. Ezen részmodulok segítségével létrehozunk egy összetett mátrix szorzó modult, amely d órajel alatt 100MHz-es sebesség mellett kiszámítja a $C = A \times B$ szorzatot, ahol $C \in \mathbb{Z}_4^{20\times 20}$, $A \in \mathbb{Z}_4^{20\times 28d}$ és $B \in \mathbb{Z}_4^{28d\times 20}$. Leírjuk a 28-as érték jelentőségét és kísérleti úton történő meghatározását. Ismertetjük $\mathbb{Z}_4^{896\times 896}$-ből vett mátrixok szorzásának módját a leírt modulok használatával. A bemutatott algoritmus

a tárolt adatok teljes kezelését megoldja a számításokkal párhuzamosan. A kapott konstrukció egy teljes szorzást 64800 órajel alatt végez el 100MHz-es órajel sebesség mellett.

Leírásra kerül még a projekt továbbfejlesztésének tervezett útja, amely a mátrixok méretének növelését és a Strassen-algoritmus segítségével a teljesítmény további optimalizálását foglalja magába.

5.1. Matematikai háttér

A munkát egyenletes eloszlású, nagy periódus hosszúságú véletlen számsorozatok előállítása inspirálta. Ehhez olyan rekurzív sorozatokat használunk amelynek alakja

$$u_n \equiv a_{d-1}u_{n-1} + a_{d-2}u_{n-2} + \ldots + a_0u_{n-d} \mod 2^s,\ a_i \in \{0,1,2,3\}, s \in \mathbb{Z}^+$$

Az itt használt elméleti háttér megtalálható [11]-ben.

Az $a_0, a_1, ..., a_{d-1}$ értékek úgy kerülnek kiválasztásra, hogy

$$x^d - a_{d-1}x^{d-1} - \ldots - a_0 \equiv (x-1)^2P(X) \mod 2$$

teljesüljön valamilyen $P(x)$ irreducibilis polinomra. $P(x)$-et praktikus úgy választani, hogy a rendje maximális legyen, ugyanis $P(x)$ rendje szoros kapcsolatban áll a sorozat periódushosszával. Az így kapott u_n sorozat eloszlása nem feltétlenül egyenletes. Megállapítható viszont, hogy a következő négy sorozat közül pontosan egyre ez teljesül:

$$\begin{aligned}
u_n^{(0)} &= a_{d-1}u_{n-1} + a_{d-2}u_{n-2} + \ldots + a_1u_{n-d+1} + a_0u_{n-d} \\
u_n^{(1)} &= a_{d-1}u_{n-1} + a_{d-2}u_{n-2} + \ldots + a_1u_{n-d+1} + (a_0+2)u_{n-d} \\
u_n^{(2)} &= a_{d-1}u_{n-1} + a_{d-2}u_{n-2} + \ldots + (a_1+2)u_{n-d+1} + a_0u_{n-d} \\
u_n^{(3)} &= a_{d-1}u_{n-1} + a_{d-2}u_{n-2} + \ldots + (a_1+2)u_{n-d+1} + (a_0+2)u_{n-d}
\end{aligned}$$

Az állítás bizonyítása megtalálható [12]-ben. Az egyenletes eloszlású sorozat megtalálása az eloszlás pontos ismerete miatt szükséges.

Legyen

$$M(u) = \begin{pmatrix}
0 & 1 & \ldots & 0 & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & 1 & 0 \\
0 & 0 & \ldots & 0 & 1 \\
a_0 & a_1 & \ldots & a_{d-2} & a_{d-1}
\end{pmatrix}$$

az u sorozat társmátrixa. A fenti négy sorozat közül az egyenletes eloszlású megtalálásához $M(u)^{2^{d-1}-2}$ kiszámítására van szükség. Ha $M(u)^{2^{d-1}-2}$ az egységmátrix, akkor u periódusa $2^{d-1}-2$, vagyis nem az a sorozat amit kerestünk.

Közönséges számítógépeken a mátrixhatványozás nagyon időigényes műveletté válhat a gyakorlati szempontból érdekes d értékekre. A fejlesztés célja a művelet jelentős felgyorsítása az FPGA által nyújtott különleges lehetőségeket kihasználva a klasszikus architektúrákon elérhető implementációkhoz képest.

5.2. Az implementációhoz használt hardware

A projekt egy Xilinx XUPV505-LX110T fejlesztői platformon került implementálásra. Az eszközzel való kommunikációra a platformon található különféle portok nyújtanak lehetőséget. A projekt az RS-232 soros portot használja a PC és az eszköz közötti kommunikációra. Egy nagysebességű PCI Express csatlakozó szintén rendelkezésre áll, ha az átküldeni kívánt adatmennyiség indokolja a használatát.

Az eszköz legjelentősebb eleme a Virtex-5 XC5VLX110T FPGA. Az FPGA számításokhoz használt fő alkotóeleme a hat bemenetű look-up table-ök, melyek 17280 db Slice-ba vannak rendezve, Slice-onként négy LUT-tal, azaz összesen 69120 LUT-tal. Egy hat bemenetű LUT 64 bit adat tárolására képes, a hat bites input pedig címként fogható fel, amely megmondja, hogy a tárolt bitek közül pontosan melyik legyen az output. A tárolt 64 bit beállításával a LUT bármilyen legfeljebb hat bit bemenetű logikai függvény előállítására felkonfigurálható. A LUT-ok először egy multiply-accumulate művelet végrehajtására tettük alkalmassá, majd később nagyobb és összetettebb modulokká szereltük össze. Az eszközön található LUT-ok negyede használható 32 bit mély shift regiszter-ként is, ezek segítségével kerültek implementálásra a tárolók, melyek a számítást végző modulokat közvetlenül táplálja adatokkal.

Az eszközhöz csatlakozik egy 256MB DDR2 SODIMM memória modul, amelyen azon adatok tárolhatók melyek az FPGA-n már nem férnek el.

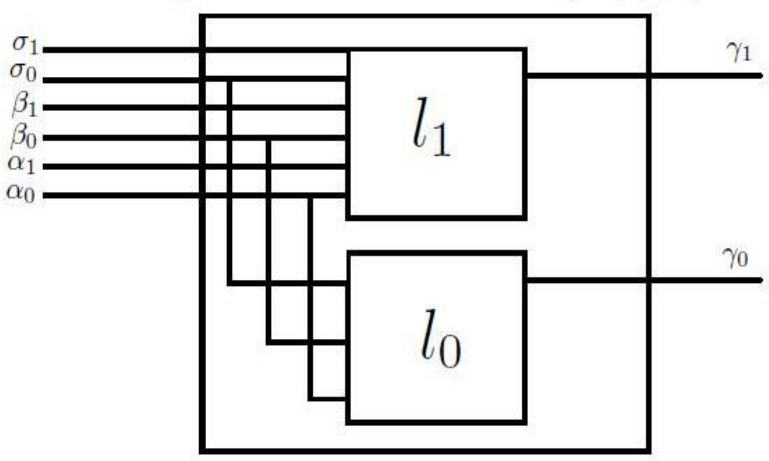
5.3. Az számításban résztvevő modulok struktúrája

Az alkalmazás alapvető alkotóelemei az $L(a, b, s) = c$-vel jelölt LUT-ok, ahol a, b, c és s kétjegyű bináris számok. Az L által végrehajtott művelet a multiply-accumulate:

$$c \equiv (a \cdot b) + s \mod 4$$

Legyen $a = \alpha_1\alpha_0$, $b = \beta_1\beta_0$, $s = \sigma_1\sigma_0$, $c = \gamma_1\gamma_0$ és $L = (l_1, l_0)$ ahol l_1 és l_0 a két LUT amely végrehajtja a műveletet. Mindkettő az output egy-egy bitjét állítja elő:

- $l_0(\alpha_0, \beta_0, \sigma_0) = \gamma_0$
- $l_1(a, b, s) = \gamma_1$

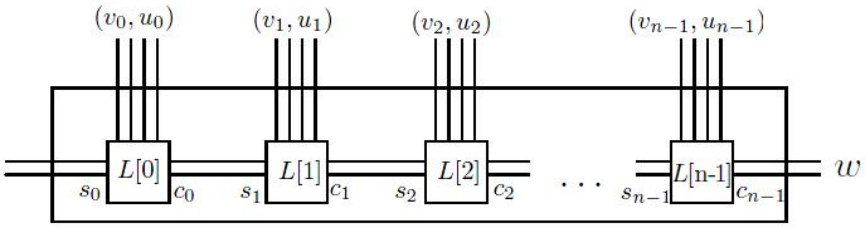


3. ábra. $L(a, b, s)$ felépítése

Az l_0 LUT-nak három bit inputra van szüksége a művelet végrehajtásához, l_1-nek viszont mind a hat bit inputra szüksége van.

Az l_0 és l_1 LUT-oknak a fent leírt művelet végrehajtásához beállított értéke a függelékben található meg.

Ezen alapelemek segítségével kiszámítható két vektor $v = (v_0, v_1, ..., v_{n-1})$ és $u = (u_0, u_1, ..., u_{n-1})$ belső szorzata. Definiáljuk az $m = (L[0], L[1], ..., L[n-1])$ modult n darab a fent leírtaknak megfelelő $L[i]$-vel jelölt alapelem sorbakapcsolásával. Ebben az m modulban az alkotóelemek kimenetét a sorban következő alkotóelem összeg-inputjaként használjuk, azaz $s_{i+1} = c_i$ minden $i = 0, 1, ..., n-2$-re, ahol s_i és c_i az $L[i]$ alkotóelem s inputja és c outputja.



4. ábra. $m(u, v)$ felépítése

Az m modul felfogható egy függvényként, amely az u, v n hosszúságú kétjegyű bináris számokból álló vektorpárra c_{n-1}-en kiadja a két vektor kétjegyű belső szorzatát, vagyis $m(u, v) = w$. Összesen az m-ben használt LUT-ok száma $2n$. Vegyük észre, hogy tetszőleges hosszúságú vektorok is használhatók a számításban ha összekötjük m kimenetét az $L[0]$ alkotóelem összeg-bemenetével ($c_{n-1} = s_0$), majd iterált módon n elemenként ráshifteljük a vektorokat a modul inputjára:

Function $iterated\_m(u, v)$

1. Legyen $\kappa = \lceil \frac{k}{n} \rceil$, $v', u' \in \mathbb{Z}_4^{\kappa \cdot n}$
2. for $i = 0$ to $\kappa \cdot n - 1$ begin

3. if $i < k$ $v'_i = v_i$ else $v'_i = 0$

4. if $i < k$ $u'_i = u_i$ else $u'_i = 0$

5. end for

6. Definiáljuk v_{temp}, u_{temp}, w, és legyen $w = 0$

7. for $i = 0$ to $\kappa - 1$ begin

8. $v_{temp} = (v'_{i \cdot n}, v'_{1+(i \cdot n)}, ..., v'_{n-1+(i \cdot n)})$

9. $u_{temp} = (u'_{i \cdot n}, u'_{1+(i \cdot n)}, ..., u'_{n-1+(i \cdot n)})$

10. $w = w + m(v_{temp}, u_{temp})$

11. end for

12. return w

end Function

Itt u' és v' u és v nullákkal való kiegészítése.

Látni fogjuk, hogy az n paraméter értéke kritikus volt a teljes projekt tulajdonságainak meghatározásában. A kísérlet, amellyel n értéke meghatározásra került a projekt felépítésének áttekintése után kerül bemutatásra.

A célunk egy olyan modul létrehozása, amely elvégzi két mátrix, $A, B \in \mathbb{Z}_4^{k \times k}$ szorzását, ahol $\mathbb{Z}_4$ a mod 4 maradékosztály gyűrű. A továbbiakban legyen $C \in \mathbb{Z}_4^{k \times k}$ az output mátrix, amire $C = A \times B$ teljesül. Továbbá legyen a_i az A mátrix $i.$ sora, b_j pedig a B mátrix $j.$ oszlopa.

Az m-mel jelölt szorzóegységek hierarchikus összeszervezésével bonyolultabb modulokat állítunk elő. Először tíz m modulból állítsuk elő szorzóegységek $R = (m_0, m_1, ..., m_9)$-vel jelölt sorát. Ezzel az output mátrix egy sorának tíz egymást követő elemét számíthatjuk ki:

$$R(a_i, b_j, b_{j+1}, ..., b_{j+9}) = (c_{i,j}, c_{i,j+1}, ..., c_{i,j+9})$$

Ahol $c_{i,j} = a_i \cdot b_j$. Az a_i input vektort R mind a tíz szorzóegysége közösen használja. Mint azt fent említettük, ezen vektorok hossza tetszőleges lehet, de az n-nél hosszabb vektorokat iteratív módon kell ráshiftelni R bemenetére.

Tíz R modul összeszervezésével megkapjuk az $M_{10\times10} = (R_0, R_1, ..., R_9)$ modult, amely C-nek egy 10×10-es részmátrixát számítja ki:

$M_{10\times10}(a_i, a_{i+1}, ..., a_{i+9}, b_j, b_{j+1}, ..., b_{j+9}) =$

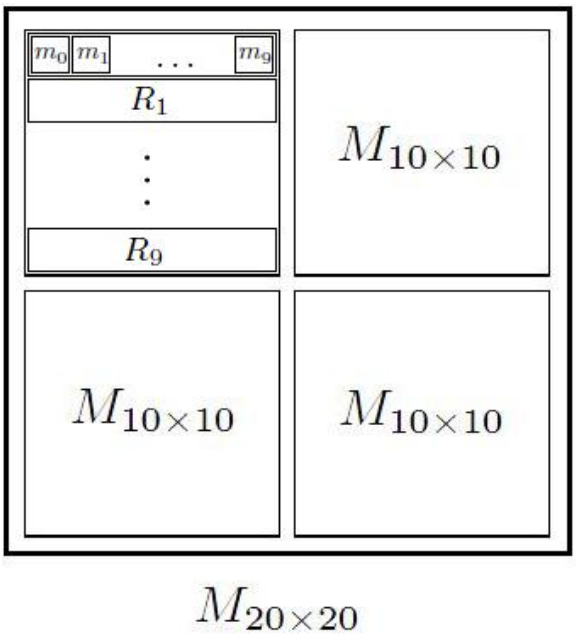
$$\begin{pmatrix} c_{i,j} & c_{i,j+1} & \cdots & c_{i,j+9} \\ c_{i+1,j} & c_{i+1,j+1} & \cdots & c_{i+1,j+9} \\ \vdots & \ddots & & \\ c_{i+9,j} & c_{i+9,j+1} & \cdots & c_{i+9,j+9} \end{pmatrix}$$

Végül négy ilyen modul összeszervezésével kapunk egy modult, amely C-nek egy 20×20-as részmátrixát számítja ki:

$M_{20\times 20}(a_i, a_{i+1}, ..., a_{i+19}, b_j, b_{j+1}, ..., b_{j+19}) =$

$$\begin{pmatrix} c_{i,j} & c_{i,j+1} & \cdots & c_{i,j+19} \\ c_{i+1,j} & c_{i+1,j+1} & \cdots & c_{i+1,j+19} \\ \vdots & \ddots & & \\ c_{i+19,j} & c_{i+19,j+1} & \cdots & c_{i+19,j+19} \end{pmatrix}$$

Az $M_{20\times 20}$ modul inputja húsz-húsz vektor az A és B mátrixokból. Hardware megszorítások miatt, egészen pontosan az eszközön található LUT-ok száma miatt a szorzóegységek ennél nagyobb hierarchiájának implementálása már nem praktikus. Az $M_{20\times 20}$ modul 400 m szorzóegységből áll. A 3. ábra mutatja az $M_{20\times 20}$ felépítésében használt modulok hierarchiáját.



5. ábra. $M_{20\times 20}$ felépítése

Az $M_{20\times 20}$ iteratív használatával tetszőleges méretű mátrixok összeszorozhatók, a C output mátrix 20×20-as részmátrixát előállítva minden iterációban. Miután inputként megadtunk húsz sort az A mátrixból és húsz oszlopot a B mátrixból és megkaptuk a kívánt outputot, egyszerűen megismételhetjük a folyamatot új inputokra, kicserélve az A vagy B mátrixból vett inputot húsz új sorra vagy oszlopra, amíg meg nem kapjuk a teljes C output mátrixot:

Function $large\_matrix\_mult(A, B)$

1. Legyen $\kappa = \lceil \frac{k}{20} \rceil$, $A', B', C' \in \mathbb{Z}_4^{\kappa\cdot n \times \kappa\cdot n}$
2. for $i = 0$ to $20\kappa - 1$ begin

3. for $j = 0$ to $20\kappa - 1$ begin

4. if $i < k$ and $j < k$ $a'_{ij} = a_{ij}$ else $a'_{ij} = 0$

5. if $i < k$ and $j < k$ $b'_{ij} = b_{ij}$ else $b'_{ij} = 0$

6. end for end for

7. for $i = 0$ to $\kappa - 1$ begin

8. for $j = 0$ to $\kappa - 1$ begin

9. $C'[^{i,j}_{i+19,j+19}] = M_{20\times 20}(a_i, a_{i+1}, ..., a_{i+19}, b_j, b_{j+1}, ..., b_{j+19})$

10. end for end for

11. return $C'[^{0,0}_{k-1,k-1}]$

end Function

Ahol

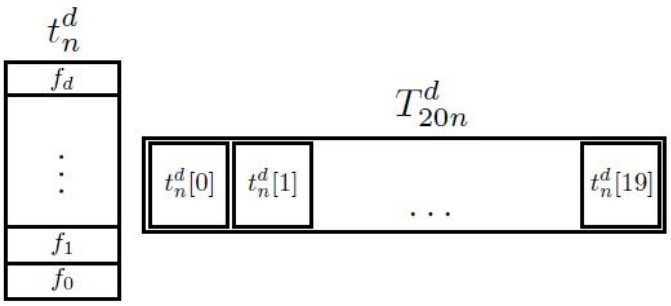
$$C'[^{i,j}_{k,l}] = \begin{pmatrix} c'_{i,j} & c'_{i,j+1} & \cdots & c'_{i,l} \\ c'_{i+1,j} & c'_{i+1,j+1} & \cdots & c'_{i+1,l} \\ \vdots & \ddots & & \\ c'_{k,j} & c'_{k,j+1} & \cdots & c'_{k,l} \end{pmatrix}$$

Vegyük észre, hogy a $large\_matrix\_mult(A, B)$ naív algoritmusban a fő ciklusokban (7-10. sorok) az A mátrixból beolvasott minden húsz sorhoz a teljes B mátrix beolvasásra kerül. A teljes eljárás során A pontosan egyszer lesz beolvasva, B pedig κ-szor. A 6. alfejezetben foglalkozunk azzal, hogy hogyan lehet ennél jobb eredményt elérni.

Mivel szinte minden a gyakorlatban érdekes esetben az $A, B \in \mathbb{Z}_4^{k\times k}$ mátrixok k mérete nagyobb lesz az n paraméternél, az ezen mátrixokból vett vektorokat iteratívan kell n elemenként ráshiftelnünk $M_{20\times 20}$ bemenetére. Emiatt az ezen mátrixokból vett vektorok tárolásának és felhasználásának hatékony módját nyújtja a shift regiszter-ekből kialakított tárolók használata.

Legyen t_n^d egy n széles és d mély shift regiszter, amely legfeljebb d darab n hosszú vektort tud tárolni, vagy ami ezzel ekvivalens, egyetlen legfeljebb nd hosszú vektort. A d értéket úgy válasszuk meg, hogy $nd \geq k$, hogy képes legyen tárolni egy teljes sort vagy oszlopot az A vagy B input mátrixokból. Legyen a t_n^d-ben tárolt vektor $f = (f_0, f_1, ...f_{d-1})$, ahol $f_i \in \mathbb{Z}_4^n,\ i = 0, 1, ..., d-1$. A t_n^d gyakorlatilag egy sor adatszerkezet, amely egy lépésben kiad egy n hosszú vektort és eltolja a teljes tartalmát n hellyel. Az i. aktiválásra az output f_i lesz, d aktiválás után a tároló üres.

Egy t_n^d tároló egy sort vagy oszlopot tárol az A vagy B mátrixokból, és húsz ilyen tároló párhuzamos összekapcsolásával (jelölje ezt $T_{20n}^d = (t_n^d[0], t_n^d[1], ..., t_n^d[19])$) egy olyan egységet kapunk, amely húsz sort vagy oszlopot képes tárolni, pontosan az $M_{20\times 20}$ által egy iterációban felhasznált adatmennyiséget. A T_{20n}^d tároló d-szeri aktiválása után az összes tárolt adat $M_{20\times 20}$ bemenetére került, n hosszú darabokra bontva aktiválásonként. Két T_{20n}^d tároló van összekötve $M_{20\times 20}$-al, egy az A mátrixból vett sorok számára, egy pedig a B mátrixból vett oszlopok számára.



6. ábra. T_{20n}^d felépítése

Az ebben az alfejezetben leírt modulok segítségével végre tudjuk hajtani a számítás egy iterációját. Miután feltöltöttünk egy T_{20n}^d tárolót húsz sorral az A mátrixból, egy másik T_{20n}^d tárolót pedig húsz oszloppal a B mátrixból, egyszerűen d aktiválás jelet küldünk a tárolóknak. Ez ráshifteli az adatokat $M_{20\times 20}$-ra, ami párhuzamosan az új adatok aktiválásonkénti fogadásával kiszámítja a 20×20-as szorzatmátrixot. Egy iterációban a lépések száma d.

5.4. A paraméterek kísérleti meghatározása

Most tekintsük az n paraméter (az m szorzóegységben összekapcsolt alkotóelemek száma) értékének meghatározását. Ez határozza meg, hogy egy lépésben milyen hosszú vektorokkal végezhetünk számítást, és ezáltal meghatározza az alkalmazás több más technikai paraméterét is. A cél az volt, hogy megtaláljuk azt a legnagyobb értéket, amely mellett a szorzó megbízhatóan kiszámítja a helyes belső szorzatot egyetlen órajel alatt. Ez az érték függ a használt hardware-től és az órajel frekvenciájától, ugyanis az eszközben a jelterjedés sebességével áll összefüggésben. A használt kártya esetén az órajel frekvenciája 100 MHz volt, ami a beépített órajelgenerátor által generált alapértelmezett frekvenciája.

A következő kísérlet szolgált n értékének meghatározására:

Legyen S egy m szorzó (az „Alany"), és legyen $E_0, E_1, ..., E_9$ még tíz m szorzó (a „Vizsgáztatók"). Képletesen fogalmazva, a Vizsgáztatók feladata, hogy ellenőrizzék az Alany válaszait olyan kérdésekre, amelyekre a válasz általuk már ismert. A „kérdések" itt a tesztadatok, olyan n hosszú u, v vektorpárok, amelyek kellően álvéletlen értékét a következő sorozat állítja elő:

$$D_i = D_{i-1} + D_{i-2} + 2D_{i-4} + D_{i-5}$$

Ahol $D_0 = D_1 = D_2 = D_3 = D_4 = 1$.

Formálisabban, legyen $p \in \mathbb{Z}_{10}$ egy számláló, amely sorra veszi fel a $0, 1..., 9$ értékeket, minden órajel során eggyel növelve az értékét, és a 9 után újra a 0 értéket felvéve. A kísérlet során minden órajel alatt a következők történnek p értékétől függően:

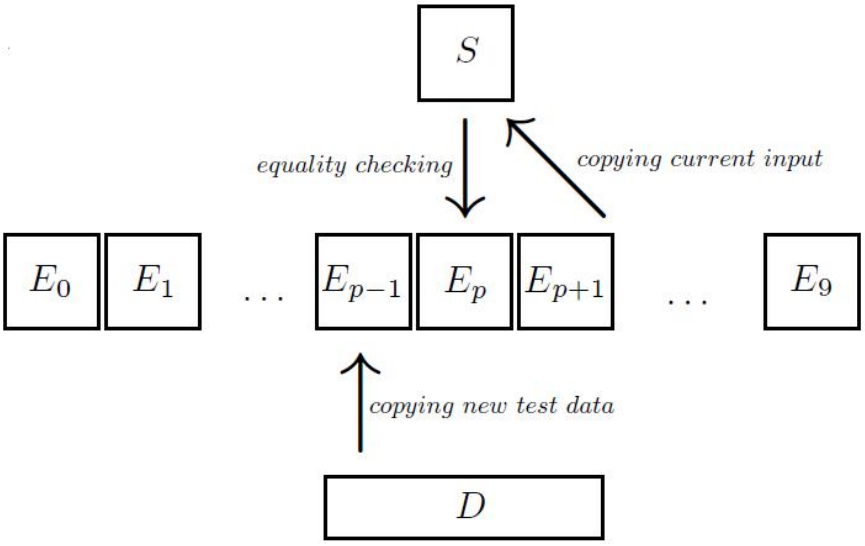
- S kimenete összehasonlításra kerül E_p kimenetével. Ha a két érték nem egyezik, hibát találtunk.
- S-nek átadjuk a tesztadatot amelyen E_{p+1} jelenleg dolgozik.

- E_{p-1} új tesztadatot kap amin dolgozhat.

Procedure testing

1. Legyenek $S, E_0, E_1, ..., E_9$ m szorzók
2. Legyen D a tesztadat generátor
3. Legyen $i \in \mathbb{N}, p \in \mathbb{Z}_{10}$
4. forever begin
   - | $i = i + 1$
   - | $p \equiv i \mod 10$
   - | if $S_{out} \neq E_{p\_out}$ then return ERROR
   - | $E_{p-1\_in} \leftarrow D(i)$
   - | $S_{in} \leftarrow E_{p+1\_in}$
5. end forever

end Procedure



7. ábra. A tesztelő modul tevékenysége p számláló érték mellett

Vegyük észre, hogy S kimenetét minden órajel alatt megvizsgáljuk, vagyis S-nek csak egy ütem áll a rendelkezésére, hogy kiszámítsa a választ a kérdésre, amit az előzőnél kapott. Egy adott Vizsgáztató viszont csak tíz órajelenként egyszer kap új tesztadatot amin dolgoznia kell, és a kimenetét csak kilenc órajellel később vizsgáljuk meg, pontosan azelőtt, hogy új tesztadatot kapna. Emiatt egy Vizsgáztatónak elég idő áll a rendelkezésére, hogy mire szükség lesz a kimenetére már biztosan helyes választ adjon.

Kezdeti értéknek $n = 16$-ot választottuk, ami kellően kicsi ahhoz, hogy feltételezhetően megfeleljen az n-re tett feltételeknek, de kellően nagy ahhoz, hogy gyakorlati szempontból érdekes legyen. Ha a kísérlet hiba nélkül zárult, vagyis az Alany megfelelően sokáig képes volt hibátlanul számítani a belső szorzatot,

akkor az n értékét megnöveltük és a kísérletet megismételtük. Miután az első hibák megjelentek, vagyis az Alany nem volt képes lépést tartani a számítással, n értéke a legnagyobb olyan érték lett amire nem voltak hibák.

A használt eszközön a legnagyobb ilyen érték 100 MHz-es órajel mellett a 28 volt, és $n = 28$-as érték mellett az m szorzók napokig hiba nélkül dolgoztak megszakítás nélkül.

5.5. Nagy mátrixok számítása

A 4. alfejezet adott egy naív algoritmust arra, hogy hogyan használhatók a leírt modulok nagy mátrixok összeszorzására. Az ily módon létrehozott implementáció csak a belső szorzatok számításában használja ki az FPGA által nyújtott párhuzamosítási lehetőségeket. A párhuzamosság további igénybe vételével a teljesítmény jelentősen növelhető. Ebben az alfejezetben leírjuk a projekt teljesítményének növelése céljából hozott implementációs döntéseket.

A legfontosabb szempont az adatok kezelése. Nagy mátrixokkal való számítás során a számítást végző modulok között mozgatandó adatok mennyisége könnyen meghaladhatja az FPGA-n hatékonyan tárolható mennyiséget. Ahogy azt korábban említettük, egy az eszközhöz csatlakoztatott 256MB DDR2 SODIMM a projekt fő adattároló eszköze. A memóriával való kommunikációhoz egy modult generáltunk a Xilinx Memory Interface Generator v3.5 intellectual property core segítségével. A modul struktúrája hierarchikus, célja a memória eszköz összekapcsolása egy felhasználói interfésszel. A RAM-mal való minden kommunikáció két FIFO soron keresztül történik, egy a parancs és cím adatok küldésére, egy pedig a memóriába írni kívánt adatok és a hozzá tartozó maszk (amikor létezik) küldésére.

A memória egy naiv felhasználása lenne minden iteráció előtt a szükséges adatok beolvasása, az iteráció után pedig az eredmény visszaírása. Ennek nem kívánt következménye lenne az, hogy a projekt sokkal több időt töltene a memória kezelésével mint a tényleges számítással. Az elvárásainknak az felelne meg a legjobban, ha a memória management (és minden egyéb segéd operáció) a számítással párhuzamosan történne. Vegyük észre, hogy mivel a mátrixok és a szorzómodul mérete is rögzített, ezért a szorzásra fordított idő mennyisége egy fix konstans, amin nem csökkenthetünk. Optimális esetben a számítás ideje a teljes futási idő felső korlátja is egyben. A nehézség ennek az optimumnak az elésében a szorzó modul és a memória modul közötti nagy sebesség eltérésben rejlik.

Egy módszer ennek elérésére az FPGA-n tárolt adatok mennyiségének növelése. Képletesen fogalmazva a fő ötlet az, hogy elegendő adatot kell készenlétben tartanunk ahhoz, hogy mire a szorzó modul minden számítással végez, addigra sikerült annyi új adatot betöltenünk, hogy tovább dolgozhasson. Formálisan, először definiáljuk a következő mennyiségeket:

- Legyen d a számítás egy iterációjához szükséges idő. Az előző alfejezetben leírtaknak megfelelően ez a T^d_{20n} tárolók mélységével egyenlő.
- Legyen $\kappa = \lceil \frac{k}{20} \rceil$, ahol k a mátrixok mérete. ($A, B \in \mathbb{Z}_4^{k \times k}$) Ezt a mennyiséget már használtuk a $large\_matrix\_mult$ algoritmusban. A fejezet további részében praktikus az A és B mátrixokat $\kappa \times \kappa$ méretűnek blokk-mátrixoknak tekinteni, ahol minden egyes elem egy 20×20-as mátrix.

- Legyen $f(A, B)$ egy algoritmus, amely mátrix szorzást végez A-n és B-n, beleértve a szükséges memóriakezelést is. Legyen $K(f)$ annak a száma, hogy az algoritmus során hányszor kell megtölteni egy T^d_{20n} tárolót, vagyis hogy hányszor kell húsz sort vagy oszlopot olvasni valamelyik mátrixból. Vegyük észre, hogy egy input mátrix egyszeri teljes beolvasásához κ alkalommal kell T^d_{20n} tárolókat feltöltenünk, hiszen egy T^d_{20n} egyszerre húsz sort vagy oszlopot tud tárolni. A $large\_matrix\_mult$ algoritmus fő ciklusa (7. sor) minden lépésben beolvassa a teljes B mátrixot és húsz sort olvas be az A mátrixból (megtöltve egy T^d_{20n} tárolót). Mivel a ciklus κ lépésből áll, ezért $K(large\_matrix\_mult) = \kappa^2 + \kappa$.

- Legyen δ egy T^d_{20n} tároló megtöltéséhez szükséges idő. Ez a mennyiség függ a tároló szélességétől és mélységétől is. A teljes idő, amit $f(A, B)$ a tárolók megtöltésével tölt $K(f)\delta$.

- Legyen $\Phi(f)$ a teljes idő, amit f a memória kezelésével tölt. Ez az összege annak az időnek, amit az A és B mátrixok memóriából való beolvasásával és amit a C szorzatmátrix memóriába írásával tölt. Az, hogy az A és B mátrixokat hányszor kell beolvasnunk f-től függ. Vegyük észre, hogy mivel a C mátrix mérete megegyezik A és B méretével, a memóriába írásának ideje egyenlő A vagy B egyszeri teljes beolvasásának idejével. Másszóval $\kappa\delta$ ideig tart. A teljes idő, amit f a memória kezelésre fordít $\Phi(f) = K(f)\delta + \kappa\delta$.

- Legyen $C(f)$ az az idő, amit $f(A, B)$ a számítással tölt. A d és κ mennyiségek definícióiból következik, hogy $C(large\_matrix\_mult) = d\kappa^2$.

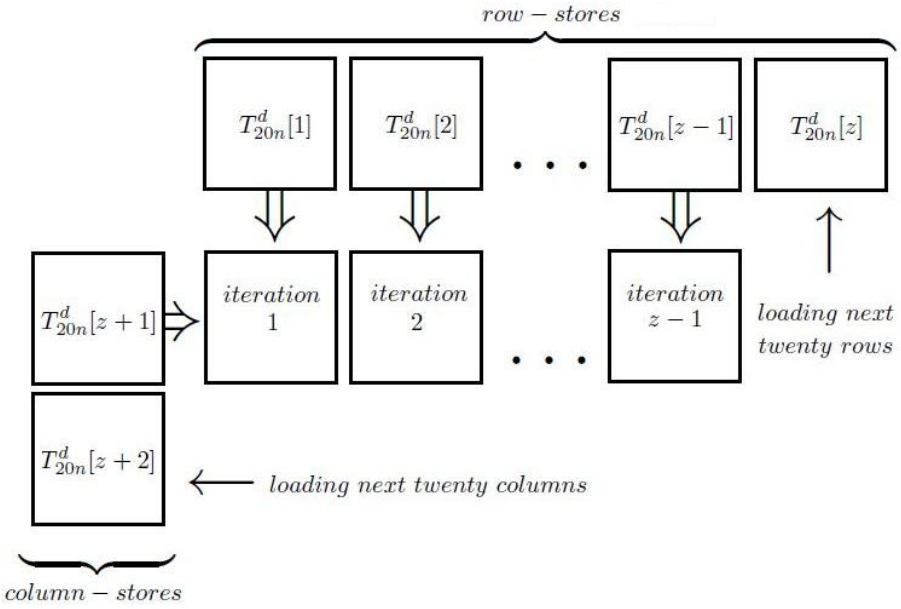
A cél $K(f)$ csökkentése úgy, hogy az egy adott iterációhoz szükséges adatok mindig készen álljanak az iteráció kezdete előtt. Ha ez elérhető, akkor $C(f)$ a projekt futási idejének felső korlátja is egyben.

Az FPGA-n való adattárolás növelése több T^d_{20n} tároló létrehozásával történhet. Egy iteráció során csak két ilyen tároló van közvetlen használatban. A többi használható a következő iterációkhoz szükséges adatok betöltéséhez.

Tegyük fel, hogy a konstrukció $z + 2$ darab T^d_{20n} tárolót tartalmaz. Ezek közül z darabot fogunk A sorainak tárolására használni (ezeket nevezzük „sor-táraknak”), kettőt pedig használjunk B oszlopainak tárolására (ezeket nevezzük „oszlop-táraknak”). Ezzel az elrendezéssel a számítás $z - 1$ iterációját tudjuk elvégezni, felhasználva $z - 1$ sor-tárban és egy oszlop-tárban tárolt adatokat. Így marad egy sor-tár és egy oszlop-tár a számítás ideje alatt új adatok betöltésére (6. ábra). A fenti definíciókat használva erre $(z - 1)d$ időnk van.

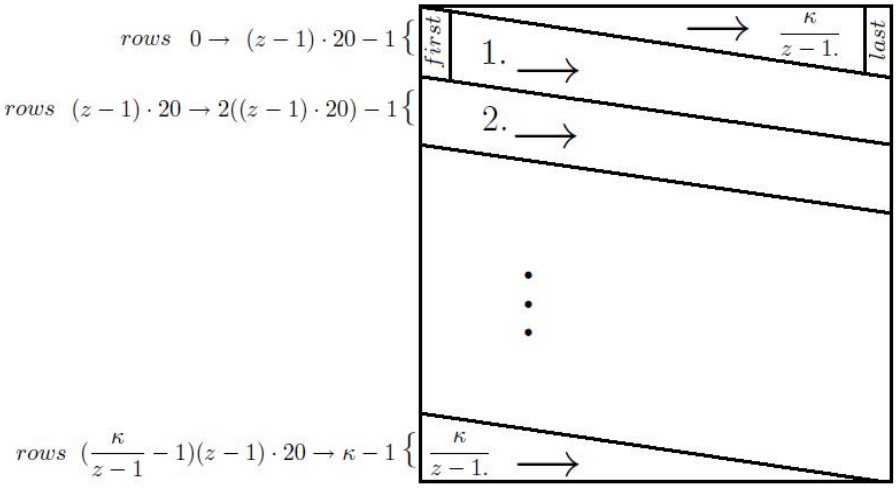
Ha mind a z sor-tárat és egy oszlop-tárat használjuk a számításban (a másik oszlop-tárba pedig az új oszlopokat töltenénk), akkor minden alkalommal amikor beolvastuk az összes oszlopot egyszerre kellene mind a z sor-tárat új sorokkal feltölteni mielőtt folytathatnánk a számítást. Ez $z\delta$ ideig tartana minden alkalommal amikor beolvastuk az összes oszlopot de még nem olvastuk be az összes sort, ami $\lfloor \frac{\kappa}{z} \rfloor$ alkalmat jelent. Ez összesen $\lfloor \frac{\kappa}{z} \rfloor z\delta$-val növelné a futási időt.

Ehelyett a kimeneti mátrix számítása enyhén átlósan halad. Lásd 7. ábra. A $z - 1$ sor-tároló, amit a számítás használ összesen $(z - 1) \cdot 20$ sort tárol. Az új sorok beolvasása lassabban történik, mint az új



8. ábra. Az FPGA-n tárolt adatok elrendezése

oszlopoké. Mire minden oszlopot beolvasunk egyszer a sor-tárak tartalma pontosan az adatok szükséges következő szegmensére (a következő $(z-1)\cdot 20$ sorra) shiftelődött. Emiatt célszerű z-t úgy választani, hogy $(z-1) \mid \kappa$ teljesüljön. Ilyen módon a teljes B mátrixot $\frac{\kappa}{z-1}$ alkalommal olvassuk be, a teljes A mátrixot pedig egyszer. Minden fent leírt $z-1$ iteráció alatt $\frac{(z-1)\cdot 20}{\kappa}$ új sort olvasunk be abba a sor-tárba, amelyet a számítás jelenleg nem használ. Amikor húsz sort beolvastunk ebbe a sor-tárba (vagyis a sor-tár megtelik), akkor aktívvá válik, vagyis felhasználásra kerül a következő iterációkban. A legkisebb indexű sorokat tartalmazó sor-tár ekkor inaktívvá válik a számításban és elkezdi befogadni az újonnan beolvasott sorokat.



9. ábra. A számítás útvonala a C szorzatmátrixban

Function $improved\_matrix\_mult(A, B)$

1. Legyen $z, \kappa = \lceil \frac{k}{20} \rceil$, $A', B', C' \in \mathbb{Z}_4^{\kappa \cdot n \times \kappa \cdot n}$
2. for $i = 0$ to $20\kappa - 1$ begin
3. for $j = 0$ to $20\kappa - 1$ begin
    4. if $i < k$ and $j < k$ $a'_{ij} = a_{ij}$ else $a'_{ij} = 0$
    5. if $i < k$ and $j < k$ $b'_{ij} = b_{ij}$ else $b'_{ij} = 0$
6. end for end for
7. A sor-tárak tartalma legyen a $0 - (z-1) \cdot 20$ sorok
8. Az oszlop-tárak tartalma legyen a $0 - 19$ oszlopok
9. For $i = 1$ to $\frac{\kappa^2}{z-1}$

Párhuzamosan: |végezzünk $z-1$ számítási iterációt
|olvassuk be a következő 20 oszlopot mod $\kappa \cdot 20$
|olvassuk be a következő $\frac{(z-1)\cdot 20}{\kappa}$ sort mod $\kappa \cdot 20$
|írjuk ki az előző $z-1$ iteráció eredményét

11. return $C'[^{0,0}_{k-1,k-1}]$

end Function

Az ebben az alfejezetben használt paraméterek lehetséges értékei függnek a használt hardware-től. A jelenlegi implementáció 896×896 méretű mátrixokkal dolgozik. Ez az érték az $n = 28$ paraméterből és a T^d_{20n} tárolókat alkotó shift regiszterekként konfigurált LUT-ok mélységéből (32 bit) származik. A $k = 896$ méretű mátrixok a legnagyobbak, amelyek tárolhatók egy LUT mélységű tárolókban. Ennél nagyobb méret választása esetén a T^d_{20n} tárolókhoz szükséges LUT-ok száma megduplázódna. Az ilyen módon felhasználható LUT-ok száma miatt $z = 10$ lett választva, vagyis összesen tizenkét T^d_{20n} tároló található a konstrukcióban. A $k = 896$-nál nagyobb mátrixok kezelése a projekt továbbfejlesztésének egyik célja, tervezett metódusait pedig a következő alfejezet taglalja.

Kényelmi szempontokból az idő mennyiségeket órajelekben mérjük, 100MHz-es órajel sebesség mellett. (Az $M_{20\times 20}$ által használ órajel sebesség)

A δ paraméter értéke a használt DDR2 RAM-tól függ. Az eszköz fizikai adatbusza 64 bit széles, a használt órajel sebesség 200MHz.

Ezen értékekből a következő paramétereket határozhatjuk meg:

- $\kappa = \lceil \frac{896}{20} \rceil = 45$
- $K(improved\_matrix\_mult) = \frac{\kappa}{z-1}\kappa + \kappa = \frac{45}{9} \cdot 45 + 45 = 270.$
- $\delta = 140$ órajel 100MHz-en.
- $\Phi(improved\_matrix\_mult) = K(improved\_matrix\_mult)\delta + \kappa\delta = 270 \cdot 140 + 45 \cdot 140 = 44100$ órajel 100MHz-en.

- $C(improved\_matrix\_mult) = d\kappa^2 = 32 \cdot 45^2 = 64800$ órajel 100MHz-en.

A $C(improved\_matrix\_mult) > \Phi(improved\_matrix\_mult)$ cél sikeres teljesítésével elértük, hogy a konstrukció futási ideje egyenlő a számításra fordított idővel.

5.6. Továbbfejlesztés

A projekt továbbfejlesztésének fő célja a mátrixok méretének növelése.

Az előző alfejezetben már említettük, hogy a T^d_{20n} tárolók d mélységének növelése nem praktikus. Az eszközön található LUT-ok csak 32 bit mély shift regiszterként használhatók, $d > 32$ paraméter mellett az egy T^d_{20n}-hez szükséges LUT-ok száma megduplázódna, és a konstrukció már most is az ilyen módon használható LUT-ok jóval több, mint felét használja (13440-et a 17280-ból). Ha ilyen módon akarjuk növelni a mátrixok méretét, ahhoz a szorzó és memória-kezelő modulok átalakítására is szükség lenne.

Ehelyett a jelenleg implementált mátrix méretet használhatjuk egységként a nagyobb mátrixokban. Ha ezeket a mátrixokat blokk-mátrixokként értelmezzük 896×896-os blokk mérettel, akkor elvégezhetjük a szorzásokat a modulok jelentősebb átalakítása nélkül.

Ily módon lehetőség nyílik a további optimalizálásra a Strassen-algoritmus segítségével. Tegyük fel, hogy megduplázzuk a mátrixok méretét és úgy értelmezzük őket, mint négy blokkból álló blokk-mátrixokat. A klasszikus algoritmus használatával a blokkokon nyolc szorzást kell elvégeznünk, hogy a két 1792×1792 méretű mátrixot összeszorozzuk. Egyfajta „osszd meg és uralkodj” stratégia felhasználásával kiválthatunk egy szorzást néhány plussz összeadásért cserébe.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} -D_2 + D_4 + D_5 + D_6 & D_1 + D_2 \\ D_3 + D_4 & D_1 - D_3 + D_5 - D_7 \end{bmatrix}$$

ahol

$$\begin{aligned} D_1 &= A_{11}(B_{12} - B_{22}) \\ D_2 &= (A_{11} + A_{12})B_{22} \\ D_3 &= (A_{21} + A_{22})B_{11} \\ D_4 &= A_{22}(B_{21} - B_{11}) \\ D_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ D_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ D_7 &= (A_{11} - A_{21})(B_{11} + B_{12}) \end{aligned}$$

Ez az algoritmus (melynek időbonyolultsága $O(n^{lg\ 7})$) felgyorsíthatja a konstrukciót nagy mátrixokon. Vegyük azonban észre, hogy a plussz összeadások sebességét részletesen érdemes megvizsgálnunk. Mivel a szorzás sebessége rendívül nagy, ezért az összeadás során is hasonló felgyorsításra lehet szükség, ha jelentős teljesítmény növekedést szeretnénk elérni.

5.7. Összefoglalás

A mátrixokkal végzett műveletek a tradícionális architektúrákon könnyen rendkívül időigényessé válhatnak. Az FPGA-k által nyújtott speciális lehetőségek, mint a konfigurálható look-up table-ök használata valamint a nagyfokú párhuzamosság támogatása alkalmassá teszik az eszközt a műveletek jelentős felgyorsítására.

A bemutatott implementáció egy álvéletlen számsorozatokat előállító eljáráshoz kapcsolódik. Az eljárás része $\mathbb{Z}_4$ felett értelmezett nagyméretű mátrixok nagyon magas hatványra emelése. A dolgozat egy Virtex-5 XC5VLX110T FPGA-n létrehozott rendkívül gyors mátrixszorzást végző implementációt mutat be. A matematikai háttér rövid ismertetése után az implementációhoz használt hardware fontosabb technikai paramétereiről esett szó.

Az implementációból elsőként egy olyan modult írtunk le, amely multiply-accumulate műveletre konfigurált LUT-ok sorba kapcsolásával vektorok belső szorzatát tudja kiszámítani. Kísérleti úton meghatározásra került, hogy legfeljebb hány LUT kapcsolható sorba úgy, hogy a helyes eredményt egy órajel alatt megkapjuk. Ez a szám a használt hardware-en 100MHz-es órajel mellett 28 volt. Az eszközön 400 ilyen modul került elhelyezésre, amelyek együttesen a szorzatmátrix 20×20-as részmátrixát számítják ki. Ezzel a modullal közvetlen kapcsolatban állnak a LUT-okból kialakított shift regiszterek, amelyek a szorzónak közvetlenül átadásra kerülő adatokat tárolják.

Bemutattuk az algoritmust, amely a leírt modulok segítségével 896×896 méretű mátrixokat szoroz össze. A legfontoabb kérdés ennél az algoritmusnál a memória kezelése. A mátrixokat az eszközhöz kapcsolt 256MB-os DDR2 SODIMM memórián tároltuk. A memóriakezelő és a szorzómodul közötti nagy sebességkülönbség miatt az adatmozgatás gondos megtervezése nélkül a memória management ideje jelentősen meghaladná a tényleges számítás idejét. A megoldást az FPGA-n tárolt adatok mennyiségének növelése és az adatok betöltésének speciális sorrendje jelentette. A számítás emiatt enyhén átlósan halad a szorzatmátrixban. A számítás ideje (összesen 64800 órajel 100MHz mellett) ily módon megegyezik a teljes futási idővel, ugyanis minden egyéb művelet vele párhuzamosan végezhetővé vált.

Szót ejtettünk még a továbbfejlesztés tervezett irányáról, amely a mátrixok méretének növelését és a Strassen-algoritmus felhasználásával a teljesítmény további optimalizálását jelentette.

A jelenlegi implementáció továbbfejlesztésének legnagyobb gyakorlati akadálya a konfiguráció Verilog kódjának fordítási ideje. A létrehozott modulok nagy mérete és összetettsége miatt az FPGA-t felkonfiguráló .bit fájl előállítása több órás folyamat. Ez az idő a projekt méretének növekedésével rohamosan nő, ugyanis a chip-en való fizikai elhelyezést optimalizáló folyamat egyre időigényesebbé válik ahogy a felhasznált erőforrások mennyisége közelít a maximumhoz. A két legnagyobb méretű modul (amelyek a szorzást és a memóriakezelést végzik) elhelyezése teszi ki a fordítási idő túlnyomó többségét. Az időigény ilyen mérvű növekedése jelentősen megnehezíti a feljesztési folyamatot.

Köszönetnyilvánítás

Köszönöm Herendi Tamás tanár úrnak, amiért mindenben támogatott és akinek segítsége és ösztönzése nélkül számtalan lehetőségtől elestem volna.

6. Összefoglalás

Az FPGA chip-ek nagyjából harminc éves múltjuk során az informatika számtalan területén sikeresen kerültek felhasználásra. Jelen diplomamunka legfőbb eredménye a mátrixszorzás műveletét rendkívüli módon felgyorsító implementáció, amely gyorsaságának növeléséhez kihasználja mind a használt adatok specialitását, mind az FPGA kínálta lehetőségeket a kombinációs logika megvalósítása és a nagyfokú párhuzamosság létrehozása terén.

A dolgozatban először olyan bonyolultságelméleti fogalmakat ismertünk meg, melyek kapcsolatban állnak az FPGA-kon létrehozható konfigurációkkal. A Boole-hálózatok a kombinációs logika megvalósításához és a párhuzamos számítások formalizálásához is szorosan kötődnek. A hálózatok fogalmának segítségével definiáltuk a megfelelően párhuzamos algoritmusok ismérveit. Bemutattuk az ezen ismérvek formalizálásával definiált NC osztályt, amely a párhuzamos számítások területén hasonlóan központi szerepet tölt be, mint a méltán nagyhírű P osztály a szekvenciális számítások esetén.

Az elméleti fogalmakat gyakorlati alkalmazások követték. Először a kisebb Spartan-3E XC3S250E FPGA rövid bemutatása és a chip-en létrehozott implementációk bemutatása következett. Mindhárom bemutatott algoritmus más-más irányból közelített a szorzás műveletének optimalizálásához. A Partial Product eljárás pontosan az előző fejezetben megismert fogalmaknak megfelelően párhuzamosítja a műveletet. A Computed Partial Product egy hasznos megfigyelés segítségével csökkenti a szükséges erőforrások mennyiségét. A Karatsuba-szorzás elméleti megközelítésből teszi ugyanezt, magán a szorzás algoritmusán javít.

Az ezt követő fejezet célja az XUPV505-LX110T fejlesztői platform számításhoz nem kapcsolódó hardware elemeinek bemutatása és a szükséges segédműveleteket (kommunikáció, órajelek előállítása) végző modulok ismertetése volt. A diplomamunka legnagyobb lélegzetvételű fejezete az eddigieknél jelentősen összetettebb mátrixszorzást gyorsító modult ismertette. A modulok továbbfejlesztése és a teljesítményük további optimalizálása jelenleg is aktív kutatás részét képezik.

Diplomamunkámban algoritmusok FPGA-n való megvalósításának különleges lehetőségeit tárgyaltam. A bemutatott több rövidebb és egy lényegesen komplexebb implementáció a gyakorlatban is hasznosítható, a tradicionális architektúrákon létrehozhatóktól jelentősen különböző módon közelítik a megoldani kívánt problémákat.

Az FPGA-k által megoldott problémák már ma is komoly méretű halmaza a jövőben csak tovább bővülhet. Az informatikát mindig is a rendkívül gyors fejlődés jellemezte. Rugalmasságának és sokoldalúságának köszönhetően a field-programmable gate array technológia tevékenyen képes hozzájárulni ehhez a fejlődéshez.

7. Irodalomjegyzék

Hivatkozások:

[1] Volker Strassen: Gaussian Elimination is not Optimal, Numer. Math. 13, p. 354-356, 1969

[2] Don Coppersmith, Samuel Winograd: Matrix multiplication via arithmetic progressions; J. Symbolic Computation, 9, pp. 251-280 (1990)

[3] Tyler J. Earnest: Strassen's Algorithm on the Cell Broadband Engine, http://mc2.umbc.edu/docs/earnest.pdf (2008)

[4] Boyko Kakaradov: Ultra-Fast Matrix Multiplication: An Empirical Analysis of Highly Optimized Vector Algorithms; Stanford Undergraduate Research Journal Volume 3: Spring 2004

[5]Ali Irturk, Shahnam Mirzaei, Ryan Kastner: An Efficient FPGA Implementation of Scalable Matrix Inversion Core using QR Decomposition; UCSD Technical Report, CS2009-0938. (2009)

[6] Marjan Karkooti, Joseph R. Cavallaro, Chris Dick: FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm; Asilomar Conference on Signals, Systems, and Computers, Nov. 2005.

[7] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, G. N. Gaydadjiev: 64-bit Floating-Point FPGA Matrix Multiplication; Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays (2005)

[8] Faycal Bensaali, Abbes Amira, Reza Sotudeh: Floating-Point Matrix Product on FPGA; aiccsa, pp.466-473, 2007 IEEE/ACS International Conference on Computer Systems and Applications (2007)

[9] Syed M. Qasim, Ahmed A. Telba and Abdulhameed Y. AlMazroo: FPGA Design and Implementation of Matrix Multiplier Architectures for Image and Signal Processing Applications; IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.2, February 2010

[10] Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, Muralidaran Vijayaraghavan: Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA; MEMOCODE '07 Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (2007)

[11] T. Herendi: Uniform distribution of linear recurrences modulo prime powers; J. Finite Fields And Applications 10 (2004), 1-23.

[12] T. Herendi: Construction of uniformly distributed linear recurring sequences modulo powers of 2, (megjelenés alatt)

Könyvek:

Christos Harilaos Papadimitriou: Számítási Bonyolultság; 1999 Győr Novadat

8. Függelék

(α_0, β_0) \ σ_0	0	1
(0,0)	0	1
(0,1)	0	1
(1,0)	0	1
(1,1)	1	0

1. táblázat. l_0 tartalma

(a, b) \ s	0	1	2	3
(0,0)	0	0	1	1
(0,1)	0	0	1	1
(0,2)	0	0	1	1
(0,3)	0	0	1	1
(1,0)	0	0	1	1
(1,1)	0	1	1	0
(1,2)	1	1	0	0
(1,3)	1	0	0	1
(2,0)	0	0	1	1
(2,1)	1	1	0	0
(2,2)	0	0	1	1
(2,3)	1	1	0	0
(3,0)	0	0	1	1
(3,1)	1	0	0	1
(3,2)	1	1	0	0
(3,3)	0	1	1	0

2. táblázat. l_1 tartalma