دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

پایان‌نامه کارشناسی

گرایش معماری سیستم‌های کامپیوتری

پیاده‌سازی لایه‌ی کانولوشن و کاهش بعد شبکه‌ی عصبی روی FPGA

نگارش

سینا مهدی‌پور سراوانی

استاد راهنما

دکتر رضا صفا بخش

تیر ۱۳۹۸

۳۱۵۹۱۷۳

بسمه تعالی

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

فرم تعریف پروژه
فارغ التحصیلی دوره کارشناسی

دانشکده مهندسی کامپیوتر
و فناوری اطلاعات

تاریخ: .....
شماره: .....

عنوان پروژه: پیاده‌سازی لایه‌ی کانولوشن و کاهش بعد شبکه‌ی عصبی روی FPGA

استاد راهنمای پروژه: دکتر رضا صفا بخش — امضاء: ۹۷/۱۲/۱۴

مشخصات دانشجو:

نام و نام خانوادگی: سینا مهدی‌پور سراوانی — **گرایش:** معماری سیستم‌های کامپیوتری

شماره دانشجوئی: ۹۳۳۱۰۴۹ — **ترم ثبت نام پروژه:** ۹

داوران پروژه:

۱- دکتر صفا بخش — امضاء داور: ۹۷/۱۲/۱۵ مورد تایید است

۲- — امضاء داور:

شرح پروژه (در صورت مشترک بودن بخشی از کار که بعهده دانشجو می باشد مشخص شود):

شتاب‌دهنده‌های استنتاج شبکه‌های عصبی بر پایه‌ی FPGA اخیراً به شکلی فزاینده مورد توجه قرار گرفته‌اند. هرچند که PFGA-ها سرعت و منابع کمتری نسبت به یک واحد پردازش گرافیکی مدرن دارند، اما یک پیاده‌سازی بهینه و هوشمندانه بر روی آن‌ها می‌تواند همان گذردهی یک واحد پردازش گرافیکی را با مصرف انرژی بسیار کمتری به ارمغان آورد. به کمک FPGA-ها، می‌توان معماری شتاب‌دهنده و عرض مسیر داده را دقیقا متناسب با شبکه‌ی هدف طراحی کرد که این مزیتی نسبت به استفاده از واحدهای پردازش گرافیکی یا طراحی مدارهای ASIC است. از طرفی یکی از مهم‌ترین و پرکاربردترین شبکه‌های عصبی موجود شبکه‌ی عصبی کانولوشنی است. هم‌چنین علاقه و نیاز به اعمال استنتاج CNN در محیط‌های محاسبات نهفته (مثل دستگاه‌های قابل حمل و ماشین‌های خودراننده) در حال افزایش است که در آن‌ها مصرف پایین توان و تاخیر کم از فاکتورهای مهم هستند. از این رو در این پروژه می‌خواهیم به شتاب‌دهی یک لایه‌ی کانولوشن و یک لایه‌ی کاهش بعد به کمک FPGA برای یک شبکه‌ی عصبی کانولوشنی بپردازیم. در این پروژه، هدف پیاده‌سازی بهینه‌ی یک لایه‌ی کانولوشنی و یک لایه‌ی کاهش بعد (اتحاد) از یک شبکه‌ی عصبی کانولوشنی بر روی یک تراشه‌ی FPGA Zynq است. برای این پیاده‌سازی می‌خواهیم از ابزار سنتز سطح بالای Vivado استفاده کنیم. البته نکته‌ی قابل توجه این است که هدف تنها به پیاده‌سازی این دو لایه‌ی شبکه‌ی عصبی خلاصه نمی‌شود؛ بلکه می‌خواهیم این پیاده‌سازی با توجه به نکات و راهکارهای بهینه‌سازی (بخش ۲ – ۱ پیوست) بهینه و هوشمندانه باشد. معماری هدف به این شکل است که k فیلتر m*m به ماتریس n*n تصویر اعمال شده، کانولوشن محاسبه شده، کاهش بعد انجام شده و نتایج در k ماتریس ذخیره می‌شود.

وسائل مورد نیاز:

برد FPGA سری Zynq، نرم‌افزار Vivado High Level Synthesis

محل انجام پروژه: دانشکده مهندسی کامپیوتر و فناوری اطلاعات — **تاریخ شروع:**

این قسمت توسط دانشکده تکمیل میگردد:

تاریخ تصویب در گروه: ۹۷/۱۲/۱۵ — **اسم و امضاء:**

تاریخ تصویب در دانشکده: — **اسم و امضاء:**

اصلاحات لازم در تعریف پروژه:

توجه: پروژه حداکثر یکماه و نیم پس از شروع ترمی که در آن در درس پروژه ثبت نام بعمل آمده است باید به تصویب برسد.

نسخه۱- دانشکده	نسخه ۲- استاد راهنما	نسخه ۳- دانشجو

تقدیر و تشکر

با سپاس فراوان از راهنمایی‌ها و زحمات استاد ارجمندم جناب آقای دکتر صفابخش کـه از ابتـدای راه و در طی انجام این تحقیق، با رهنمودهایشان مرا در نگارش این اثر یاری نمودند.

به نام خدا

تعهدنامه اصالت اثر

تاریخ: ۱۳۹۸/۴/۱

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

اینجانب سینا مهدی‌پور سراوانی متعهد می‌شوم که مطالب مندرج در این پایان نامه حاصل کار پژوهشی اینجانب تحت نظارت و راهنمایی اساتید دانشگاه صنعتی امیرکبیر بوده و به دستاوردهای دیگران که در این پژوهش از آنها استفاده شده است مطابق مقررات و روال متعارف ارجاع و در فهرست منابع و مآخذ ذکر گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم‌سطح یا بالاتر ارائه نگردیده است.

در صورت اثبات تخلف در هر زمان، مدرک تحصیلی صادر شده توسط دانشگاه از درجه اعتبار ساقط بوده و دانشگاه حق پیگیری قانونی خواهد داشت.

کلیه نتایج و حقوق حاصل از این پایان نامه متعلق به دانشگاه صنعتی امیرکبیر می‌باشد. هرگونه استفاده از نتایج علمی و عملی، واگذاری اطلاعات به دیگران یا چاپ و تکثیر، نسخه‌برداری، ترجمه و اقتباس از این پایان نامه بدون موافقت کتبی دانشگاه صنعتی امیرکبیر ممنوع است. نقل مطالب با ذکر مآخذ بلامانع است.

سینا مهدی‌پور سراوانی

امضا

چکیده

شتاب‌دهنده‌های شبکه‌های عصبی بر پایه‌ی FPGA اخیراً به شدت مورد توجه قرار گرفته‌اند. هرچند که FPGA-ها سرعت و منابع کمتری نسبت به یک واحد پردازش گرافیکی مدرن دارند، اما یک پیاده‌سازی بهینه و هوشمندانه بر روی آن‌ها می‌تواند همان گذردهی یک واحد پردازش گرافیکی را با مصرف انرژی بسیار کمتری به ارمغان آورد. به کمک FPGA-ها، می‌توان معماری شتاب‌دهنده و عرض مسیر داده را دقیقاً متناسب با شبکه‌ی هدف طراحی کرد که این مزیتی نسبت به استفاده از واحدهای پردازش مرکزی یا گرافیکی است. از طرفی یکی از مهم‌ترین و پرکاربردترین شبکه‌های عصبی موجود شبکه‌ی عصبی کانولوشنی است. همچنین علاقه و نیاز به اعمال استنتاجِ CNN (جلورانی یک ورودی در شبکه و تولید خروجی) در محیط‌های محاسبات نهفته مثل دستگاه‌های قابل حمل و ماشین‌های خودراننده در حال افزایش است که در آن‌ها مصرف پایین توان و تاخیر کم از فاکتورهای مهم هستند. از این رو، در این پروژه می‌خواهیم به شتاب‌دهی توابع کانولوشن و کاهش بعد به کمک FPGA برای یک شبکه‌ی عصبی کانولوشنی بپردازیم. در این پروژه، هدف پیاده‌سازی بهینه‌ی یک لایه‌ی کانولوشنی و یک لایه‌ی کاهش بعد (ادغام) از یک شبکه‌ی عصبی کانولوشنی برای یک تراشه‌ی FPGA ZYNQ است. برای این پیاده-سازی می‌خواهیم از ابزار سنتز سطح بالای Vivado استفاده کنیم. معماری هدف به این شکل است که ۳ فیلتر ۳×۳ به ماتریس تصویر ورودی اعمال، کانولوشن محاسبه، کاهش بعد انجام و نتایج در ۳ درگاه خروجی تحویل داده می‌شوند. این کار از طریق پیاده‌سازی یک هسته‌ی مالکیت معنوی (IP Core) به کمک ابزار سنتز سطح بالای ویوادو انجام شده است. این هسته سپس به روش‌های شبیه‌سازی و همینطور اجرا بر روی برد مورد آزمایش قرار گرفت و بررسی نتایج، صحت عملکرد آن را نشان داد. علاوه بر این، با استفاده از این شتاب‌دهی سخت‌افزاری توانستیم دو عمل کانولوشن و کاهش بعد را در زمان کمتری نسبت به نرم‌افزار اجرایی روی پردازنده‌ی مرکزی انجام دهیم.

واژه‌های کلیدی:

FPGA، شبکه‌ی عصبی کانولوشنی، هسته‌ی مالکیت معنوی (IP Core)، کانولوشن، سنتز سطح بالا (HLS)، کاهش بعد، برد زیبو (ZYBO)

فهرست مطالب

صفحه

فصل اول

مقدمه

مقدمه

تحقیقات اخیر در زمینه‌ی شبکه‌های عصبی، مزیت‌های قابل توجه آن‌ها را نسبت به روش‌های سنتی که وابسته به استخراج دستی ویژگی‌ها و مدل‌ها هستند، نشان می‌دهد. امروزه شبکه‌های عصبی به طور گسترده در تشخیص تصویر، صدا و فیلم به کار گرفته شده‌اند. پیش از اینکه بتوان به طور مستقل از آن‌ها برای طبقه‌بندی الگوهای جدید استفاده کرد، لازم است که حجم زیادی داده برای آموزش آن‌ها مورد استفاده قرار گیرد. اما پیچیدگی زیاد محاسباتی و حافظه‌ای در فاز استنتاج شبکه‌های عصبی مشکلات بزرگی را در کاربرد این ابزار به وجود می‌آورد. شبکه‌های عصبی عمیق را می‌توان همچون لایه-هایی از ضرب ماتریسی در نظر گرفت. شبکه‌ها معمولا تعداد زیادی لایه و وزن (تا چند صد هزار) دارند و در نتیجه استنتاج در این سیستم‌ها به حجم زیادی از محاسبات نیاز دارد. پلتفورم‌های مبتنی بر واحد پردازش مرکزی ظرفیت محاسباتی لازم را برای برخی کاربردها ندارند. خوشبختانه مقدار زیادی از این محاسبات را می‌توان به صورت موازی انجام داد. در نتیجه، پلتفورم‌های مبتنی بر واحد پردازش گرافیکی به علت ظرفیت محاسباتی بالا و وجود فریم‌ورک‌های توسعه‌ی خوب و ساده انتخاب اول برای انجام محاسبات شبکه‌های عصبی هستند [۱ و ۲ و ۳].

از طرفی شتاب‌دهنده‌های استنتاج شبکه‌های عصبی بر پایه‌ی FPGA اخیراً مورد توجه قرار گرفته‌اند. متاسفانه مصرف انرژی یک واحد پردازش گرافیکی به تنهایی ۲۰۰ تا ۳۰۰ وات است و وزن تقریبی آن ۱ کیلوگرم است. این ویژگی‌ها چندان مناسب دستگاه‌های نهفته و قابل حمل نیستند. با توجه به سخت افزار طراحی شده به صورت هدف محور، FPGA-ها راه حل ممکن برای پیشی گرفتن از واحدهای پردازش گرافیکی از نظر سرعت و بهره‌وری انرژی هستند. هرچند که FPGA-ها سرعت و منابع کمتری نسبت به یک واحد پردازش گرافیکی مدرن دارند، اما یک پیاده‌سازی بهینه و هوشمندانه بر روی آن‌ها می‌تواند همان گذردهی یک واحد پردازش گرافیکی را با مصرف انرژی بسیار کمتری به ارمغان آورد. تا کنون طراحی‌های گوناگونی از شتاب‌دهنده‌های مبتنی بر FPGA با شیوه‌های بهینه-سازی نرم‌افزاری و سخت‌افزاری به منظور دست‌یابی به سرعت بالا و کاهش مصرف انرژی ارائه شده‌اند که در مرور کارهای گذشته به آن‌ها می‌پردازیم [۱ و ۲ و ۳].

نتایج و دقت‌های لبه‌ی علم در تشخیص تصویر، تولید عنوان و توصیف تصاویر و بسیاری از کاربردهای دیگر با استفاده از شبکه‌های عمیق کانولوشنی به دست آمده‌اند. این شبکه‌ها نیز از نیازهای

محاسباتی سنگین اشاره شده در هر دو فاز آموزش و استنتاج مستثنی نیستند. شتاب‌دهی سخت‌افزاری به طور خاص‌تری برای فاز استنتاج مورد توجه است، زیرا معمولا آموزش تنها یک بار و به صورت برون-خطی انجام می‌شود اما استنتاج بارها و بارها تکرار می‌شود. همچنین تاکید و علاقه‌ی در حال افزایشی برای اعمال استنتاجِ CNN در محیط‌های محاسبات نهفته (مثل دستگاه‌های قابل حمل و ماشین‌های خودراننده) به وجود آمده است که در آن‌ها مصرف پایین توان و تاخیر کم از فاکتورهای مهم هستند [۴].

طبق تحقیقات گذشته، اصلی‌ترین بخش یک شبکه‌ی کانولوشنی، محاسبه‌ی کانولوشن است. در نتیجه مهم‌ترین گام در تلاش برای بهینه‌سازی این شبکه‌ها به کمک سخت‌افزار، شتاب‌دهی تابع کانولوشن در آن‌هاست. دیگر تابع مهمی که در CNN-ها معمولا پس از هر لایه‌ی کانولوشن قرار می-گیرد، تابع کاهش بعد یا ادغام است. از این رو در این پروژه به شتاب‌دهی این دو تابع کانولوشن و کاهش بعد با طراحی یک IP Core با استفاده از ابزار سنتز سطح بالای ویوادو[۱] پرداخته شد.

این پروژه استفاده از FPGA را برای پیاده‌سازی تابع کانولوشن و کاهش بعد که هسته‌ی اصلی در شبکه‌های عصبی کانولوشنی هستند، برای اعمال سه فیلتر تشخیص لبه[۲]، منبت‌کاری[۳] و شفافیت (تیز کردن[۴]) به ماتریس تصویر، هدف قرار داده است. قابلیت موازی‌سازی ارثی در FPGA آن را به ابزاری بسیار مناسب برای این منظور تبدیل می‌کند. برای پیاده‌سازی، ابزار سنتز سطح بالای ویوادو با توجه به منابع موجود گزینه‌ی مناسبی تشخیص داده شد.

در ادامه‌ی این گزارش ابتدا در فصل دوم به تعریف و تشریح کلی مفاهیم و معرفی ابزارهای مورد استفاده می‌پردازیم. در فصل سوم به بررسی کارهای مرتبط و روش‌ها و ترفندهای شتاب‌دهی توابع به کمک FPGA می‌پردازیم و همچنین روش‌های طراحی و ضوابط آن‌ها را تشریح می‌کنیم. در فصل چهارم شیوه‌ی پیاده‌سازی سامانه به کمک تعاریف و

---

۱ Vivado High-Level Synthesis

۲ Edge Detection Filter

۳ Emboss Filter

۴ Sharpening Filter

ابزارهـا و راهکارهـای معرفـی شـده در فصـل‌هـای دو و سـه را بیـان مـی‌کنـیم. در پایـان در فصـل پـنجم نتـایج حاصـل از پیـاده‌سـازی را بررسـی کـرده و بـه کارهـای آینـده مـی‌پـردازیم.

فصل دوم

معرفی مبانی

معرفی مبانی

در این فصل به تعریف مفاهیم و معرفی ابزارهای مورد استفاده در این گزارش می‌پردازیم.

۲-۱- یادگیری ماشین

علم یادگیری ماشین از علوم کامپیوتر سرچشمه می‌گیرد و علمی است که بر توانایی کامپیوترها برای یادگیری، بدون اینکه به طور واضح برنامه‌نویسی شده باشند، تمرکز دارد [۵]. تعریف رسمی‌تر و جدیدتر زیر توسط تام م. میشل[۱] در [۶] ارائه شده است:

«گفته می‌شود که یک برنامه‌ی کامپیوتری از تجربه‌ی E نسبت به یک طبقه‌ی T از کارها و معیار عملکرد P، یاد می‌گیرد، اگر عملکرد آن در این طبقه‌ی کارها با معیار سنجش P با تجربه‌ی E بهبود یابد.»

برای شرح موضوع از مثال برنامه‌ای که می‌خواهد دست‌نوشته‌ها را تشخیص دهد استفاده می‌کنیم. کار T تشخیص و طبقه‌بندی کلمات دست‌نویس در تصویر، معیار عملکرد P درصد کلماتی که درست طبقه‌بندی شده‌اند و تجربه‌ی E پایگاه داده‌ای از کلمات است که طبقات آن‌ها مشخص است.

یادگیری ماشین کاربردهای بسیاری در طبقه‌بندی تصاویر، تشخیص شئ، تشخیص گفتار، یادگیری بازی و غیره دارد.

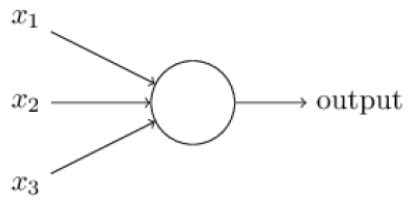
شبکه‌های عصبی زیرمجموعه‌ای از یادگیری ماشین هستند که در ادامه به توضیح آن‌ها پرداخته شده است.

۲-۱-۱- شبکه‌های عصبی

شبکه‌های عصبی نوعی از مدل‌های یادگیری ماشین هستند که ارزش کاربردی بسیار بالایی در زمینه‌ی تشخیص الگو دارند. عبارت شبکه‌ی عصبی ریشه در تلاش برای پیدا کردن بازنمایی اطلاعات در

[۱] Tom M. Mitchell

پردازش‌های موجود در سامانه‌های زیستی دارد که یکی از خروجی‌های ارزشمند و تاثیرگذار این تلاش‌ها پرسپترون[۱] بود [۷]. پرسپترون یک نورون مصنوعی است که توسط فرانک روزنبلت[۲] در دهه‌های ۱۹۵۰ و ۱۹۶۰ میلادی توسعه پیدا کرد. پرسپترون چندین ورودی دودویی را می‌گیرد و یک خروجی دودویی تولید می‌کند [۲].



$$output = \begin{cases} 0 \ if \ \sum_j w_j x_j \leq threshold \\ 1 \ if \ \sum_j w_j x_j > threshold \end{cases} \quad (۱)$$

شکل ۱- مدل و معادله‌ی پرسپترون [۲]

روزنبلت وزن‌ها (w_i) و مقدار آستانه[۳] را که مقادیر حقیقی و پارامترهای پرسپترون هستند، معرفی کرد. بر اساس این پارامترها، پرسپترون یکی از مقادیر یک یا صفر را با توجه به ورودی، همان طور که در شکل ۱ نشان داده شده است، خروجی می‌دهد.

مدل‌های حال حاضر نورون‌ها به اشکال مختلف به پرسپترون روزنبلت شبیه هستند. به جای مقدار آستانه، مقدار تمایل[۴] یا b معرفی شده است که قرینه‌ی مقدار آستانه است. به علاوه، تابع فعال-سازی معرفی شده است که هدف آن این است که اجازه دهد تا تغییرات کوچک در وزن‌ها یا مقدار تمایل تنها موجب تغییری کوچک در خروجی شوند؛ این ویژگی به آموزش شبکه بسیار کمک می‌کند. در نورون پرسپترون چنین تغییرات کوچکی می‌توانستند باعث شوند که خروجی برعکس شود. مدل و تعریف جدید نورون در معادله‌ی زیر نشان داده شده است:

$$y = f(\mathbf{w} \cdot \mathbf{x} + b) = f(\sum_j w_j x_j + b) \quad (۲)$$

---

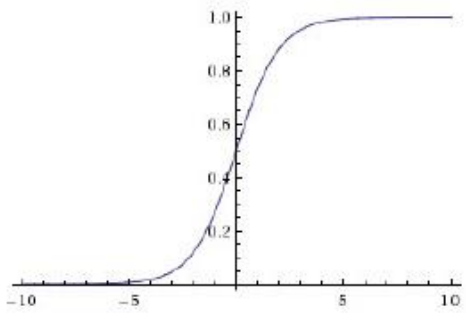
۱ Perceptron

۲ Frank Rosenblatt

۳ Threshold

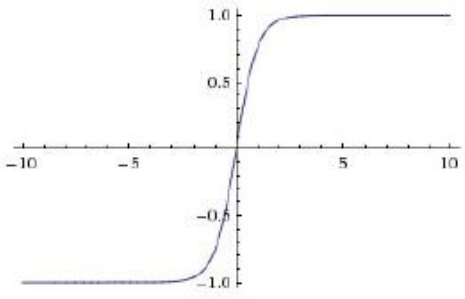
۴ Bias

در این معادله خروجی y با جمع مقدار تمایل با ضرب نقطه‌ای بردار w که شامل وزن‌های نورون است با بردار ورودی x حاصل می‌شود. تابع f تابع غیر خطی فعال‌ساز است. در شکل ۲ سه مورد از رایج-ترین توابع فعال‌سازی همراه معادلات آن‌ها نمایش داده شده است.



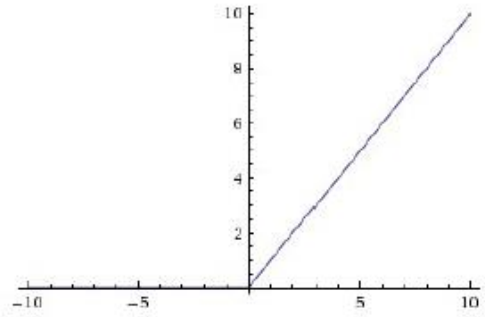
$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad (2.3)$$

(a) Sigmoid



$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \qquad (2.4)$$

(b) tanh



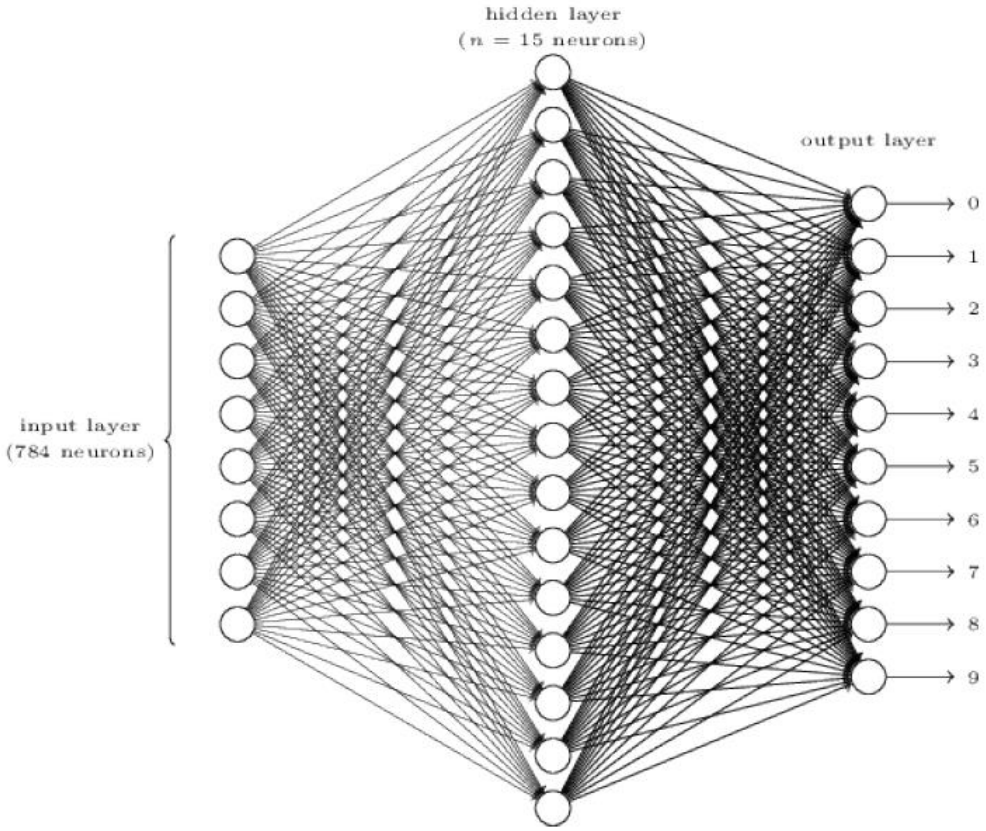
$$RelU(z) = max(0, z) \qquad (2.5)$$

شکل ۲- توابع فعال‌سازی [۲]

یک شبکه‌ی عصبی از تعداد زیادی نورون که در چندین لایه قرار گرفته‌اند تشکیل می‌شود. اولین لایه‌ی یک شبکه‌ی عصبی لایه‌ی ورودی است که پس از آن یک یا چند لایه‌ی پنهان قرار می-گیرد. این لایه‌ها به این علت پنهان نامیده می‌شوند که نورون‌های این لایه‌ها نه ورودی هستند و نه خروجی. پس از آخرین لایه‌ی پنهان، لایه‌ی خروجی قرار می‌گیرد.

تعداد نورون‌ها در لایه‌ی خروجی به کار مورد نظر بستگی دارد. برای مثال برای طبقه‌بندی ارقام دست‌نویس طبیعی است که ده نورون خروجی استفاده شود که هر یک نمایان‌گر یک رقم باشند. یک

نمونه شبکه‌ی عصبی با یک لایه‌ی پنهان در شکل ۳ نشان داده شده است. شبکه‌ای که در آن همه‌ی ورودی‌های یک لایه از لایه‌ی پیشین می‌آیند، یک شبکه‌ی عصبی پیشخور نامیده می‌شود. اگر اتصالات بین نورون‌ها بتوانند یک دور جهت‌دار در شبکه به وجود آورند، شبکه بازگشتی نامیده می‌شود [۲ و ۸].



شکل ۳- نمونه‌ای از یک شبکه‌ی عصبی [۸]

۲-۱-۲- آموزش شبکه

روند آموزش یا یادگیری یک شبکه در واقع راهی برای بهینه‌سازی وزن‌ها و مقادیر تمایل شبکه است. از آنجا که آموزش یک شبکه در این پروژه مورد بحث نیست، تنها مقدمه‌ای از آن بیان می‌شود.

برای آموزش یک شبکه داشتن مجموعه‌ای از بردارهای ورودی $\{X_n\}$ و مجموعه‌ای از بردارهای هدف $\{t_n\}$ که n در آن‌ها از ۱ تا N است، الزامی است. همچنین تابعی به اسم تابع هزینه C(w, b) مورد نیاز است که هدف کمینه‌سازی این تابع برای رسیدن به یک نتیجه‌ی بهتر در طبقه‌بندی است:

$$C(w,b) = \frac{1}{2N}\sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n) - \mathbf{t}_n||^2 \qquad (۳)$$

در اینجا w و b نمایان‌گر کل مجموعه‌ی وزن‌ها و مقادیر تمایل هستند. برای کمینه کردن تابع هزینه، از الگوریتم کاهش گرادیان[۱] استفاده می‌شود. ایده‌ی این الگوریتم این است که مقادیر وزن‌ها و تمایل‌ها را با تغییر آن‌ها با گام‌های کوچک در جهت عکس گرادیان به‌روزرسانی می‌کند. این به‌روزرسانی برای w_k و b_l به صورت زیر است [۲]:

$$w_k \to w'_k = w_k - \eta\frac{\partial C(w,b)}{\partial w_k} \qquad (۴)$$

$$b_l \to b'_l = b_l - \eta\frac{\partial C(w,b)}{\partial b_l} \qquad (۵)$$

این به‌روزرسانی‌ها بارها و بارها تکرار می‌شوند تا تابع هزینه به سمت یک کمینه‌ی محلی یا سراسری همگرا شود. پارامتر η نرخ آموزش است و تعیین می‌کند که تابع هزینه با چه سرعتی همگرا شود. انتخاب نرخ آموزش بزرگ باعث می‌شود که مقدار تابع هزینه زیاد شود و در نتیجه همگرا نشود.

[۱] Gradient Descent

پارامترهایی مثل η هایپرپارامتر نامیده می‌شوند و مثل وزن‌ها یا مقادیر تمایل یاد گرفته نمی‌شوند؛ اما با این حال باید به طور مناسب انتخاب شوند [۹].

بسته به تعداد ورودی‌های آموزشی، زمان آموزش می‌تواند بسیار طولانی باشد. یک راه حل برای افزایش سرعت آموزش استفاده از کاهش گرادیان تصادفی[۱] است که در آن به جای استفاده از کل داده-های آموزشی، یک زیرمجموعه تصادفی از آن‌ها انتخاب می‌شوند.

امروزه از مهم‌ترین روش‌های آموزش شبکه‌های عصبی استفاده از الگوریتم پس‌انتشار[۲] است که در محاسبه‌ی گرادیان تابع هزینه سریع عمل می‌کند. ایده‌ی این الگوریتم به این صورت است که ابتدا یک بردار ورودی در شبکه جلو رانده می‌شود تا خروجی توابع فعال‌سازی همه‌ی نورون‌ها محاسبه شود، سپس مقدار خطا با توجه بردار هدف مربوط به این ورودی در جهت عقب‌گرد در کل شبکه منتشر می-شود[۷ و ۸].

۲-۱-۳- شبکه‌ی عصبی کانولوشنی

شبکه‌ی عصبی کانولوشنی (CNN) یک شبکه‌ی عصبی پیشخور است که حلقه در آن وجود ندارد. این شبکه‌ها بسیار مشابه شبکه‌های عصبی معمولی هستند و از نورون‌ها با وزن‌ها و مقادیر تمایل قابل یادگیری تشکیل شده‌اند. تفاوت اصلی این است که در کنار لایه‌های کاملا متصل، دو نوع لایه‌ی اصلی دیگر نیز در معماری شبکه‌های عصبی کانولوشنی مورد استفاده قرار می‌گیرند: لایه‌ی کانولوشن و لایه‌ی کاهش بعد. این لایه‌های چندین بار پشت هم قرار می‌گیرند تا یک CNN را بسازند [۹].

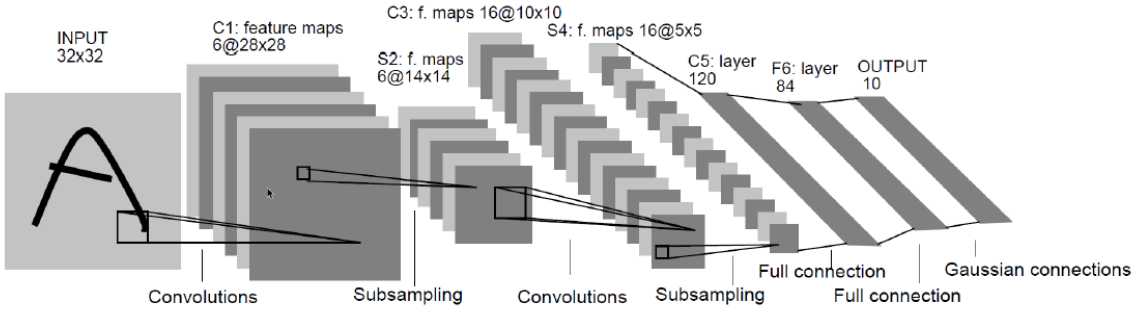
شکل ۴ نمونه‌ای از یک CNN را نشان می‌دهد که از ۷ لایه به جز ورودی تشکیل شده است. چهار لایه‌ی اول جفت‌های از لایه‌ی کانولوشن و لایه‌ی کاهش بعد هستند. لایه‌ی پنجم هم یک لایه‌ی کانولوشنی است اما از آنجا که خروجی لایه‌ی قبل (بردار ویژگی‌های S4) هم‌اندازه‌ی فیلترهای این لایه

---

۱ Stochastic Gradient Descent

۲ Backpropagation

است، این لایه معادل یک لایه‌ی کاملاً متصل است. دو لایه‌ی آخر لایه‌های کاملاً متصل و خروجی هستند.



شکل ۴- نمونه‌ای از یک شبکه‌ی عصبی کانولوشنی. هر صفحه یک نقشه‌ی ویژگی است [۱۰].

در قسمت بعدی لایه‌های اصلی شبکه‌ی عصبی کانولوشنی توضیح داده می‌شوند و نشان داده می‌شود که CNN-ها از سه ایده‌ی پایه‌ای استفاده می‌کنند: زمینه‌های پذیرش محلی[۱]، وزن‌ها و مقادیر تمایل مشترک و کاهش بعد.

۲-۱-۴- لایه‌های اصلی شبکه‌ی عصبی کانولوشنی

۲-۱-۴-۱- لایه‌ی کانولوشن

بلوک اصلی سازنده‌ی شبکه‌ی عصبی کانولوشنی، لایه‌ی کانولوشن است. در شبکه‌های عصبی معمولی که پیش‌تر درباره‌ی آن‌ها صحبت کردیم، که همان لایه‌های کاملا متصل هستند، ورودی یک بردار است و این بدان معنی است که تصاویر دو بعدی در یک بردار جای داده می‌شوند. به این شکل، یک ویژگی اساسی تصاویر که همان هم‌بستگی بیشتر پیکسل‌های نزدیک به هم نسبت به پیکسل‌های دور از هم است، نادیده گرفته می‌شود. در یک لایه‌ی کانولوشنی، ورودی شکل اصلی خود را حفظ می‌کند تا از این هم‌بستگی بین پیکسل‌ها بهره ببرد. همچنین با استفاده از زمینه‌های پذیرش محلی یا زیرناحیه‌های

[۱] Local Receptive Fields

تصویر ورودی، لایه‌ی کانولوشنی قادر است که ویژگی‌های محلی استخراج کند. این استخراج با اعمال یک عمل کانولوشن بر روی تصویر ورودی با استفاده از یک هسته[۱] که مانند فیلتر عمل می‌کند، انجام می‌شود. می‌توان این عمل را اینگونه توصیف کرد که یک فیلتر روی کل تصویر لغزانده می‌شود و هر بار برای هر زیرناحیه‌ای که زیر فیلتر قرار می‌گیرد، یک عمل ضرب نقطه‌ای بین مقادیر وزن‌های فیلتر و ناحیه انجام می‌شود و نتیجه در خانه‌ی متناظر در نقشه‌ی ویژگی[۲] قرار می‌گیرد. ایده‌ی اشتراک وزن‌ها به این معنی است که همه‌ی واحدهای یک نقشه‌ی ویژگی محدود هستند که مقادیر وزن یکسانی را به اشتراک بگذارند. استفاده از چند فیلتر در لایه‌ی کانولوشن به حصول چند نقشه‌ی ویژگی در خروجی می‌انجامد که باعث می‌شود تا ویژگی‌های بیشتری استخراج شوند. پس از اعمال ضرب نقطه‌ای، یک مقدار تمایل به هر عنصر نقشه‌ی ویژگی خروجی اضافه می‌شود. این مقدار تمایل نیز در کل نقشه‌ی ویژگی یکسان است. در آموزش شبکه وزن‌های فیلتر می‌توانند طوری تنظیم شوند تا ویژگی‌هایی را استخراج کنند که برای تشخیص طبقه‌های هدف مورد نیاز هستند.

به عنوان مثال لایه‌ی C3 در شبکه‌ی شکل ۴ را در نظر بگیرید؛ لایه‌ی C3 دومین لایه‌ی کانولوشنی است و ۶ نقشه‌ی ویژگی ۱۴×۱۴ از لایه‌ی پیشین می‌گیرد، سپس عمل کانولوشن با ۱۶ فیلتر بر روی این نقشه‌ها اعمال می‌شود که حاصل آن ۱۶ نقشه‌ی ویژگی در خروجی است. عموماً عمق فیلترها برابر تعداد نقشه‌های ویژگی ورودی است و طول و عرض معمول آن‌ها ۳×۳ و ۵×۵ است.

محدوده‌ی فضایی[۳] یا همان طول و عرض فیلترها، هایپرپارامتر[۴] هستند و در کل ۴ هایپرپارامتر در یک لایه‌ی کانولوشنی مورد نیاز است [۹]:

- تعداد فیلترها، K.
- محدوده‌ی فضایی فیلترها، F، طول و عرض فیلترها که می‌توانند متفاوت باشند.
- گام، S، تعداد پیکسل‌هایی که فیلتر لغزان حرکت می‌کند.

---

۱ Kernel

۲ Feature map

۳ Spatial Extent

۴ Hyperparameter

- مقدار افزونه‌ی صفر[۱]، P، که برای کنترل اندازه‌ی خروجی استفاده می‌شود.

مقدار این هایپرپارامترها با توجه به خاصیت‌های مورد نیاز لایه‌ی کانولوشنی انتخاب می‌شوند. معادله‌ی ۶ برای محاسبه‌ی اندازه‌ی نقشه‌ی ویژگی خروجی مورد استفاده قرار می‌گیرد. W عرض است و طول نیز به همین شکل محاسبه می‌شود [۲].

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1 \qquad (۶)$$

۲-۱-۴-۲- لایه‌ی کاهش بعد (ادغام)[۲]

لایه‌های کانولوشن معمولا با لایه‌های کاهش بعد جفت می‌شوند. این لایه‌ها نقشه‌ی ویژگی خروجی لایه‌های کانولوشنی را دریافت و اندازه و ابعاد آن‌ها را به منظور کاهش پارامترها در شبکه کوچک می‌کنند. این کار باعث می‌شود که حجم محاسبات مورد نیاز کم شود و بیش‌برازش[۳] کنترل شود. بیش‌برازش مشکلی است که هنگامی رخ می‌دهد که شبکه به جای خود داده برای اختلالات[۴] آن مناسب می‌شود. عملیات کاهش بعد به صورت جداگانه روی هر نقشه‌ی ویژگی اعمال می‌شود و در نتیجه تعداد نقشه‌ها در دو طرف این لایه یکسان است. لایه‌ی ادغام نیز مانند لایه‌ی کانولوشن روی زیرنواحی نقشه‌ی ویژگی یک فیلتر اعمال می‌کند. کار این فیلتر معمولا یک عمل محاسبه‌ی بیشینه است که بر روی یک ناحیه‌ی ۲×۲ با گام ۲ اعمال می‌شود. این کار باعث نصف شدن طول و عرض خروجی می‌شود.

لایه‌ی کاهش بعد به دو هایپرپارامتر نیاز دارد:

- محدوده‌ی فضایی، F.

---

۱ Zero Padding

۲ Pooling Layer

۳ Overfitting

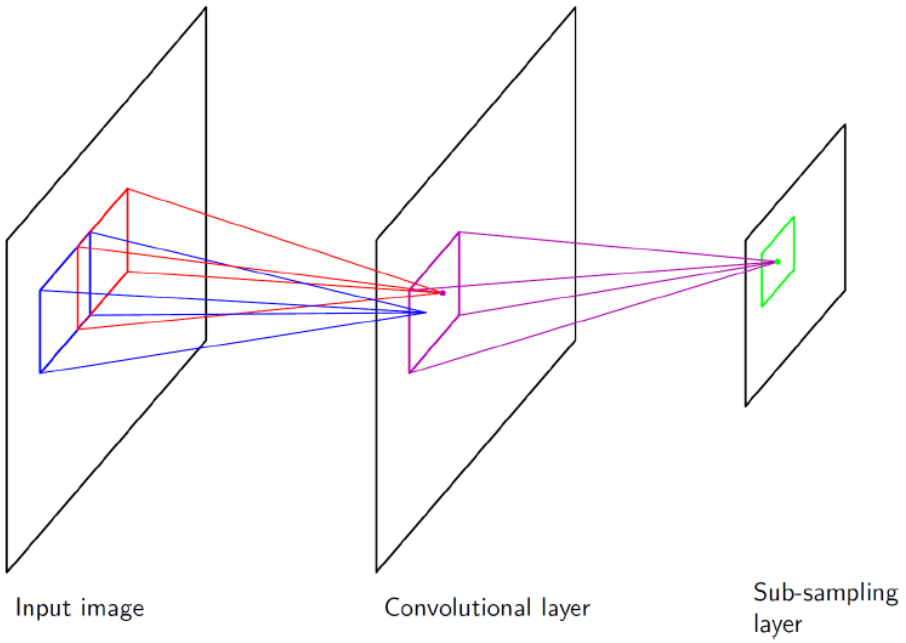
۴ Noise

- گام، S.

مشابه لایه‌ی کانولوشن، اندازه‌ی نقشه‌ی ویژگی خروجی با معادله‌ی محاسبه می‌شود:

$$W_{out} = \frac{W_{in} - F}{S} + 1 \quad (۷)$$

دو نسخه‌ی معمول برای تنظیم هایپرپارامترها در این لایه وجود دارد: رایج‌ترین نسخه این است که گام و طول و عرض همگی برابر ۲ باشند؛ اما در نسخه‌ی دیگر که ادغام هم‌پوشان نام دارد طول و عرض برابر ۳ و گام برابر ۲ است. همچنین، با اینکه رایج‌ترین نوع عملیات ادغام، محاسبه‌ی بیشینه است اما می‌تواند محاسبه‌ی میانگین یا نرم L2 نیز باشد. ادغام میانگینی در تاریخ محبوب‌تر بوده است اما اثبات شده است که ادغام بیشینه‌ای در عمل بهتر کار می‌کند [۹].



شکل ۵- نمایش لایه‌ی کانولوشن و کاهش بعد [۷]

شکل ۵ نشان می‌دهد که چگونه زمینه‌های پذیرش در تصویر ورودی به یک نـورون در خروجـی لایـه‌ی کانولوشن مربوط می‌شوند. همچنین در این عکس نشان داده شده است که لایه‌ی ادغام اندازه‌ی خروجی لایه‌ی کانولوشن را کاهش می‌دهد.

۲-۲- ابزار سنتز سطح بالای ویوادو[۱]

کامپایلر ویوادو HLS یک محیط برنامه‌نویسی مشابه نرم‌افزارهای توسعه‌ی برنامه‌های کاربردی برای پردازنده‌های استاندارد و اختصاصی فراهم می‌کند و در تفسیر، تحلیل و بهینه‌سازی برنامه‌های C/C++ با تکنولوژی‌های اصلی کامپایلرهای پردازنده آشناست.

با هدف قرار دادن FPGA به عنوان سخت‌افزار محاسباتی، ویوادو HLS به مهندس نرم‌افزار اجازه می‌دهد تا گذردهی، تاخیر و توان مصرفی کد را بدون نیاز به حل مشکلات تنگنا بودن فضای تک-حافظه‌ای و محدودیت منابع محاسباتی، بهینه کند. این خاصیت اجازه می‌دهد تا الگوریتم‌های نرم‌افزاری با حجم محاسبات بالا به محصولات واقعی تبدیل شوند و فقط نمایشگر کارکرد نباشند [۱۱].

از تفاوت‌های قابل توجه بین ویوادو HLS و سایر کامپایلرها محدودیت‌هایی‌ست که بر طراح اعمال می‌کنند. درمحیط پردازنده‌ای با کامپایلر پردازنده، معماری پردازشی ثابت است و کاربر تنها با کاهش وابستگی‌های عملیاتی و دست‌کاری طرح حافظه برای رسیدن به بیشینه‌ی کارایی حافظه پنهان می‌تواند در کارایی سامانه تاثیرگذار باشد. در ویوادو HLS اما، محدودیت پلتفورم پردازش ثابت وجود ندارد و بر اساس ورودی‌های کاربر، یک پلتفورم الگوریتم‌محور تولید می‌شود. به این شکل طراح می‌تواند در کارایی برنامه‌ی کاربردی مؤثر باشد [۱۱].

---

۱ Vivado High-Level Synthesis

۲-۲-۱- طراحی FPGA بر پایه‌ی C

ابزار سنتز سطح بالای ویوادو یک توصیف C را به یک پیاده‌سازی سطح انتقال ثبات[۱] که قابل سنتز بر روی آرایه‌ی دروازه‌ی قابل برنامه‌ریزی میدان (FPGA)[۲] است تبدیل می‌کند. توصیف C را می‌توان با C، C++، SystemC یا با رابط برنامه‌نویسی کاربردی (API)[۳] زبان محاسبات باز[۴] نوشت. FPGA یک معماری به شدت موازی، بهبود عملکرد و کاهش هزینه و توان را به ارمغان می‌آورد [۱۲].

۲-۲-۱-۱- منافع سنتز سطح بالا

سنتز سطح بالا با برقراری اتصال بین دامنه‌ی نرم‌افزار و سخت‌افزار منافع اصلی زیر را طبق [۱۲] با خود همراه دارد:

- افزایش بهره‌وری طراحان سخت‌افزار:

طراحان سخت‌افزار برای طراحی سخت‌افزار با کارایی بالا می‌توانند در سطح بالاتری از تجرد[۵] عمل کنند.

- افزایش کارایی سامانه برای طراحان نرم‌افزار:

توسعه‌دهندگان نرم‌افزار می‌توانند قسمت‌های پیچیده‌ی محاسباتی الگوریتم‌های خود را برای سخت‌افزار هدف جدید، که همان FPGA است، کامپایل کنند.

همچنین استفاده از روش طراحی با سنتز سطح بالا اجازه می‌دهد تا:

- الگوریتم‌ها در سطح C توسعه پیدا کنند:

---

۱ Register Transfer Level

۲ Field Programmable Gate Array

۳ Application Programming Interface

۴ Open Computing Language (OpenCL)

۵ Abstraction

این ویژگی باعث می‌شود تا طراح در سطحی از تجرد فعالیت کند که به دانش کمتری از جزییات پیاده‌سازی بر روی سخت‌افزار نیاز باشد.

- عملکرد طراحی در سطح C سنجیده شود:

باعث می‌شود تا اعتبارسنجی صحت عملکرد طراحی بسیار سریع‌تر از راه‌های سنتی در زبان‌های توصیف سخت‌افزار انجام شود.

- پردازش سنتز C از طریق رهنمودهای[۱] بهینه‌سازی کنترل شود:

باعث می‌شود تا بتوان یک پیاده‌سازی سخت‌افزاری اختصاصی با کارایی بالا داشت.

- با استفاده از رهنمودهای بهینه‌سازی بتوان چندین پیاده‌سازی سخت‌افزاری از یک کد C داشت:

با بررسی حالت‌های مختلف فضای طراحی، احتمال یافتن بهینه‌ترین پیاده‌سازی افزایش می‌یابد.

- بتوان کد C قابل خواندن و قابل انتقال ساخت:

به این شکل می‌توان از کد C منبع برای دستگاه‌های متفاوت پیاده‌سازی‌های سخت‌افزاری متفاوتی ساخت یا کد را در پروژه‌های دیگر جا داد.

البته باید توجه داشت که هر کد زبان C قابل سنتز با این ابزار نیست. برای تشخیص این موضوع باید به مستندات شرکت تولید کننده‌ی ابزار سنتز سطح بالا (در اینجا شرکت Xilinx) مراجعه کرد. HLS در مجموع هنوز نوپا است و به نحوه‌ی کدنویسی و جزییات پیچیده‌ی طراحی وابسته است.

۲-۲-۱-۲- مفاهیم پایه‌ی سنتز سطح بالا

سنتز سطح بالا با توجه به [۱۲] شامل فازهای زیر است:

---

[۱] Directives

- زمان‌بندی

بر اساس موارد زیر مشخص می‌کند که در هر چرخه‌ی ساعت چه عملیاتی انجام شود:

- طول چرخه‌ی ساعت و فرکانس ساعت
- زمانی که بر اساس دستگاه هدف طول می‌کشد تا عملیات انجام شود
- رهنمودهای بهینه‌سازی مختص کاربر

اگر دوره‌ی ساعت طولانی‌تر باشد یا FPGA سریع‌تری مورد هدف باشد، عملیات بیشتری در یک چرخه‌ی ساعت انجام می‌شوند، یا حتی ممکن است همه‌ی عملیات در یک چرخه‌ی ساعت پایان پذیرند. متقابلاً اگر دوره‌ی ساعت کوتاه‌تر باشد یا FPGA کندتری مورد هدف باشد، سنتز سطح بالا به صورت خودکار عملیات را در تعداد چرخه‌های ساعت بیشتری زمان‌بندی می‌کند و برخی عملیات ممکن است به عنوان منابع چندچرخه‌ای پیاده‌سازی شوند.

- انطباق[۱]

مشخص می‌کند که کدام منبع سخت‌افزاری هر عملیات زمان‌بندی شده را پیاده‌سازی می‌کند. برای پیاده‌سازی راه‌حل بهینه، سنتز سطح بالا از اطلاعات دستگاه هدف استفاده می‌کند.

- استخراج کنترل منطق

کنترل منطق را استخراج می‌کند تا یک ماشین حالت متناهی بسازد که ترتیب عملیات در طراحی زبان انتقال ثبات را مشخص می‌کند.

سنتز سطح بالا کد منبع C را طبق [۱۲] به صورت زیر سنتز می‌کند:

- آرگومان‌های تابع بالا[۱] به درگاه‌های ورودی/خروجی زبان انتقال ثبات سنتز می‌شوند.

---

[۱] Binding

- توابع C به بلوک‌های سخت‌افزاری در سلسله مراتب زبان انتقال ثبات سنتز می‌شوند.

اگر کد C شامل سلسله مراتبی از زیر توابع باشد، طراحی نهایی زبان انتقال ثبات شامل سلسله مراتبی از ماژول‌ها[۲] یا موجودیت‌هایی[۳] که یک تناظر یک به یک با سلسله مراتب تابع C اصلی دارند. همه‌ی نمونه‌های یک تابع از یک بلوک پیاده‌شده‌ی زبان انتقال ثبات استفاده می‌کنند.

- حلقه‌های توابع C به صورت پیش‌فرض باز نمی‌شوند.

وقتی که حلقه‌ها باز نباشند، سنتز یک منطق برای یک دور اجرای حلقه ایجاد می‌کند و طراحی زبان انتقال ثبات این منطق را برای هر تکرار حلقه به ترتیب اجرا می‌کند. با استفاده از رهنمودهای بهینه‌سازی طراح می‌تواند حلقه‌ها را باز کند که این کار اجازه می‌دهد تا همه‌ی تکرارهای حلقه به صورت موازی اجرا شوند.

- آرایه‌ها در کد منبع C به بلوک‌های حافظه در طراحی نهایی FPGA سنتز می‌شوند.

اگر آرایه در رابط تابع بالا باشد، سنتز سطح بالا این آرایه را به عنوان درگاهی برای دست-رسی به یک بلوک‌حافظه خارج طرح سنتز می‌کند.

سنتز سطح بالا پیاده‌سازی بهینه را بر اساس رفتار پیش‌فرض، محدودیت‌ها و تمام رهنمودهای بهینه‌سازی طراح می‌سازد. طراح می‌تواند از رهنمودهای بهینه‌سازی برای تغییر و کنترل رفتار پیش‌فرض منطق درونی و درگاه‌های ورودی/خروجی استفاده کند. به این صورت طراح مجاز است که از یک کد منبع C چندین پیاده‌سازی سخت‌افزاری مختلف تولید کند.

برای تشخیص اینکه طراحی انجام شده حداقل‌های لازم را برطرف می‌کند یا نه، طراح می‌تواند معیارهای کارایی موجود در گزارش سنتز تولید شده توسط سنتز سطح بالا را مرور کند. پس از تحلیل

---

۱ Top Function

۲ Modules

۳ Entity

گزارش، طراح می‌تواند از رهنمودهای بهینه‌سازی استفاده کند تا پیاده‌سازی را تصحیح کند. گزارش سنتز شامل اطلاعاتی در مورد معیارهای کارایی زیر است:

- مساحت: مقدار منابع سخت‌افزاری لازم برای پیاده‌سازی طراحی بر اساس منابع موجود در FPGA که شامل جداول جستجو[۱]، ثبات‌ها، بلوک‌های حافظه و DSP48-ها است.
- تاخیر: تعداد چرخه‌های ساعت لازم برای محاسبه‌ی همه‌ی خروجی‌ها توسط تابع
- فرجه‌ی آغاز[۲]: تعداد چرخه‌های ساعت لازم پیش از این که تابع بتواند ورودی بپذیرد
- تاخیر تکرار حلقه: تعداد چرخه‌های ساعتی که طول می‌کشد تا یک تکرار حلقه پایان پذیرد
- فرجه‌ی آغاز حلقه: تعداد چرخه‌های ساعت پیش از این که تکرار بعدی حلقه پردازش داده را شروع کند
- تاخیر حلقه: تعداد چرخه‌های لازم برای اجرای تمام تکرارهای حلقه

۲-۲-۲- تفهیم سنتز سطح بالای ویوادو

ابزار سنتز سطح بالای ویوادو یک تابع C را به یک بلوک مالکیت معنوی سنتز می‌کند که قابل استفاده و جادهی در یک طراحی سخت‌افزاری است. این ابزار با سایر ابزارهای طراحی زایلینکس[۳] کاملا هماهنگ است و برای ساخت پیاده‌سازی بهینه از الگوریتم C از ویژگی‌های لازم و جامع پشتیبانی می-کند.

طبق [۱۲] جریان طراحی سنتز سطح بالای ویوادو به صورت زیر است:

---

۱ Look-up Tables

۲ Initiation Intervall

۳ Xilinx

۱- کامپایل، اجرا (شبیه‌سازی) و عیب‌یابی الگوریتم و کد C

۲- سنتز الگوریتم C به پیاده‌سازی زبان انتقال ثبات با استفاده از رهنمودهای بهینه‌سازی اختیاری طراح

۳- تولید گزارش‌های جامع و تحلیل طراحی

۴- اعتبارسنجی پیاده‌سازی زبان انتقال ثبات با استفاده از جریان دکمه‌ی فشاری

۵- بسته‌بندی پیاده‌سازی زبان انتقال ثبات به مجموعه‌ای از قالب‌های مالکیت‌معنوی

۲-۲-۲-۱- ورودی‌ها و خروجی‌ها

موارد زیر طبق [۱۲] ورودی‌های سنتز سطح بالای ویوادو هستند:

- تابع C نوشته شده به زبان‌های C، C++، SystemC یا یک هسته‌ی رابط کاربری برنامه-نویسی OpenCL

  این مورد ورودی اصلی به ابزار است و تابع می‌تواند سلسله مراتبی از زیر توابع باشد.

- محدودیت‌ها

  محدودیت‌ها شامل دوره‌ی ساعت، عدم اطمینان ساعت و FPGA مورد هدف می‌شود.

- رهنمودها

  رهنمودها اختیاری هستند و سنتز را برای پیاده‌سازی یک رفتار یا بهینه‌سازی بخصوص هدایت می‌کنند.

- نیمکت آزمون[۱] C و فایل‌های مرتبط با آن

  ابزار از نیمکت آزمون C برای شبیه‌سازی عملکرد تابع C پیش از سنتز و همین‌طور برای اعتبارسنجی خروجی زبان انتقال ثبات با استفاده از شبیه‌سازی مشترک C و RTL استفاده می‌کند.

---

[۱] Test Bench

طراح می‌تواند این ورودی‌ها را به دو روش به یک پروژه سنتز سطح بالای ویوادو اضافه کند. روش اول استفاده از رابط کاربری گرافیکی و روش دوم استفاده از دستورات TCL در خط فرمان است.

موارد زیر طبق [۱۲] خروجی‌های سنتز سطح بالای ویوادو هستند:

- فایل‌های پیاده‌سازی زبان انتقال ثبات در قالب زبان توصیف سخت‌افزار

به کمک سنتز ویوادو می‌توان زبان انتقال ثبات را به پیاده‌سازی سطح دروازه و فایل جریان بیت FPGA تبدیل کرد. زبان انتقال ثبات در قالب‌های استاندارد زیر موجود است:

- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)

ابزار سنتز سطح بالای ویوادو فایل‌های پیاده‌سازی را به عنوان یک بلوک مالکیت معنوی بسته‌بندی می‌کند که می‌توان از آن در سایر ابزارهای جریان طراحی زایلینکس استفاده کرد. با سنتز منطقی می‌توان این بلوک بسته‌بندی شده را به جریان بیت FPGA تبدیل کرد.

- فایل‌های گزارش

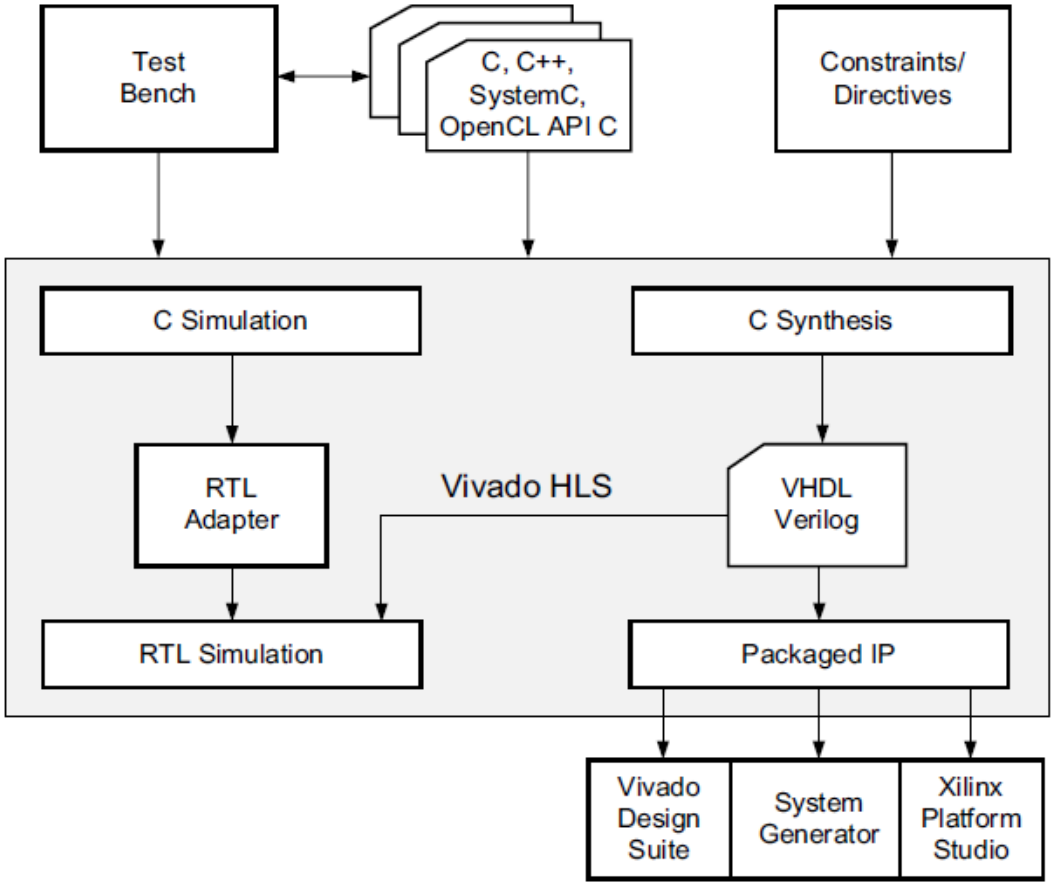
شکل ۶ جریان ورودی و خروجی در ابزار سنتز سطح بالای ویوادو را نشان می‌دهد.

۲-۲-۲-۲- نیمکت آزمون و پشتیبانی زبان

در هر برنامه‌ی C تابع بالا main() نامیده می‌شود. در ابزار سنتز سطح بالای ویوادو هر زیرتابعی از main() را می‌توان به عنوان تابع بالا برای سنتز مشخص کرد. البته خود تابع main() را نمی‌توان سنتز کرد. همچنین قوانین زیر نیز باید مورد توجه قرار گیرند [۱۲]:

- تنها تعریف یک تابع به عنوان تابع بالا برای سنتز مجاز است.
- تمام زیرتوابع در یک سلسله مرتبه‌ی زیر تابع بالا نیز سنتز می‌شوند.

- اگر طراح بخواهد که توابعی که داخل سلسله‌مرتبه‌ی زیر تابع بالا نیستند هم سنتز شوند، باید این توابع را در داخل تابع بالا جا دهد.
- جریان اعتبارسنجی برای هسته‌های رابط برنامه‌نویسی کاربردی OpenCL به مدیریت‌های خاصی در جریان سنتز سطح بالا نیاز دارد که باید مورد توجه قرار گیرند. این اطلاعات در فصل سوم [۱۲] موجود است.



شکل ۶- جریان طراحی سنتز سطح بالای ویوادو [۱۲]

هنگامی که طراح از جریان طراحی سنتز سطح بالای ویوادو استفاده می‌کند، بسیار زمان‌بر است که یک تابع C با عملکرد غلط را سنتز کند و سپس با تحلیل جزییات پیاده‌سازی تشخیص دهد که چرا

تابع عملکرد مورد انتظار را ندارد. از این رو، برای افزایش بهره‌وری از یک نیمکت آزمون برای سنجش صحت عملکرد تابع C پیش از سنتز استفاده می‌شود.

سنتز سطح بالای ویوادو از استانداردهای زیر برای کامپایل و شبیه‌سازی C پشتیبانی می‌کند:

- ANSI-C (GCC 4.6)
- C++ (G++ 4.6)
- OpenCL API (1.0 embedded profile)
- SystemC (IEEE 1666-2006, version 2.2)

سنتز سطح بالای ویوادو از بسیاری از ساختارهای زبانی C، C++ و SystemC در کنار همه‌ی انواع داده‌ی اصلی برای هر زبان شامل انواع شناور و دوبرابر پشتیبانی می‌کند. هرچند با توجه به [۱۲] سنتز برای برخی از ساختارها مثل موارد زیر پشتیبانی نمی‌شود:

- اختصاص پویای حافظه

یک FPGA مجموعه‌ی ثابتی از منابع دارد و ساخت و آزادسازی پویای منابع حافظه ممکن نیست.

- عملیات سیستم عامل

همه‌ی داده‌ها در FPGA باید از درگاه‌های ورودی خوانده شوند و در درگاه‌های خروجی نوشته شوند. عملیات سیستم عاملی مثل خواندن و نوشتن در فایل یا پرسش‌هایی چون زمان و تاریخ ممکن نیستند. در عوض، نیمکت آزمون C می‌تواند این عملیات را انجام دهد و داده‌ها را برای سنتز به عنوان آرگومان‌های تابع به تابع بدهد [۱۲].

۲-۲-۲-۳- سنتز، بهینه‌سازی و تحلیل

ابزار سنتز سطح بالای ویوادو پروژه‌محور است. هر پروژه می‌تواند چندین راه حل و مجموعه‌ای از کدهای C را شامل می‌شود. هر راه حل می‌تواند محدودیت‌ها و رهنمودهای بهینه‌سازی متفاوتی از راه حل دیگر داشته باشد. مراحل سنتز، بهینه‌سازی و تحلیل در پروسه‌ی طراحی سنتز سطح بالای ویوادو به شکل زیر است [۱۲]:

۱- ساخت یک پروژه با یک راه حل اولیه

۲- اعتبارسنجی اجرای صحیح و بدون خطای شبیه‌سازی C

۳- اجرای سنتز برای به دست آوردن نتایج

۴- تحلیل نتایج

پس از تحلیل نتایج می‌توان یک راه حل جدید با محدودیت‌ها و رهنمودهای بهینه‌سازی جدید ساخت و راه حل جدید را سنتز کرد. این روند را می‌توان آنقدر تکرار کرد تا طراحی به ویژگی‌ها و کارایی‌های مورد انتظار برسد.

با استفاده از ابزار سنتز سطح بالای ویوادو، طراح می‌تواند رهنمودهای بهینه‌سازی مختلفی را به طراحی اعمال کند که شامل موارد زیر هستند [۱۲]:

- رهنمود اجرای خط لوله‌ای به یک کار؛ که اجازه می‌دهد تا نسل بعدی اجرای آن کارها پیش از پایان اجرای نسل فعلی آغاز شوند.
- تخصیص یک تاخیر برای پایان توابع، حلقه‌ها و مناطق کد
- اعمال محدودیت روی تعداد منابع مورد استفاده
- حذف وابستگی‌های ضمنی و ارثی در کد و صدور اجازه‌ی استفاده از عملیات خاص: برای مثال اگر می‌توان مقادیر اولیه داده‌ها را نادیده گرفت (مثلا در یک خط جریان ویدیو)، اجازه داده شود که اگر کارایی بهبود می‌یابد، حافظه پیش از نوشتن خوانده شود.
- انتخاب پروتکل‌های ورودی/خروجی: برای اطمینان از این موضوع که طراحی نهایی می‌تواند به سایر بلوک‌های سخت‌افزاری با همان پروتکل ورودی/خروجی متصل شود.

۲-۲-۳- بهینه‌سازی طراحی

در این بخش به خلاصه‌ای از روش‌های بهینه‌سازی که به کمک آن‌ها می‌توان ابزار سنتز سطح بالای ویوادو را هدایت کرد تا طبق معیارهای کارایی به اهداف مورد نظر برسد، می‌پردازیم.

در جدول ۱ خلاصه‌ای از رهنمودهای بهینه‌سازی در HLS را می‌بینیم.

رهنمود	توضیح
ALLOCATION	محدودیتی بر روی تعداد عملیات، هسته‌ها یا توابع مورد استفاده اعمال می‌کند. این رهنمود می‌تواند ابزار را مجبور به به اشتراک گزاری منابع سخت-افزاری کند و ممکن است تاخیر را افزایش دهد.
ARRAY_MAP	چند آرایه‌ی کوچک را به هم چسبانده و تبدیل به یک آرایه بزرگ می‌کند تا استفاده از بلوک‌های حافظه را کاهش دهد.
ARRAY_PARTITION	آرایه‌های بزرگ را به چند آرایه‌ی کوچک‌تر یا چند ثبات منفرد تقسیم می‌کند تا دسترسی به داده را بهبود بخشد و تنگنای بلوک حافظه را از بین ببرد.
ARRAY_RESHAPE	شکل یک آرایه را از تعداد زیاد عناصر با عرض کلمه‌ی کوچک به تعداد کمتر عناصر با عرض کلمه‌ی بزرگ تغییر می‌دهد. این کار دسترسی به بلوک حافظه را بدون استفاده از بلوک‌های بیشتر بهبود می‌بخشد.
DATA_PACK	بخش‌های داده‌ی یک ساختار را داخل یک اسکالر با عرض کلمه‌ی بزرگ بسته‌بندی می‌کند.
DATAFLOW	خط لوله سازی در سطح کارها را فعال می‌کند و اجازه می‌دهد توابع و حلقه‌ها به صورت هم‌روند اجرا شوند. برای کاهش تاخیر استفاده می‌شود.

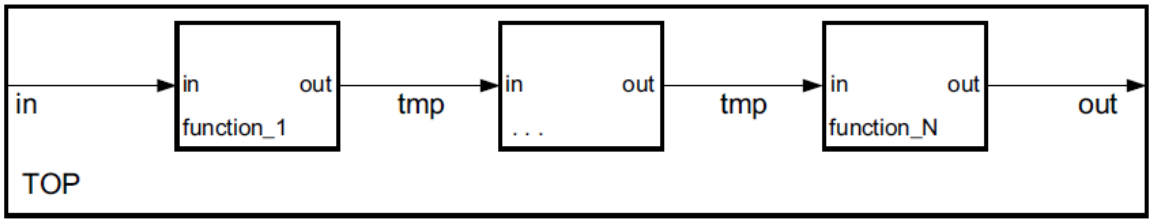
DEPENDENCE	برای ارائه‌ی اطلاعات اضافی که می‌توانند بر وابستگی‌های حمل حلقه فائق آیند، استفاده می-شود و اجازه می‌دهد تا حلقه‌ها خط لوله‌ای شوند.
EXPRESSION_BALANCE	اجازه می‌دهد تا متعادل‌سازی خودکار عبارات خاموش شود.
FUNCTION_INSTANTIATE	اجازه می‌دهد تا نمونه‌های متفاوت یک تابع به صورت محلی بهینه شوند.
INLINE	با حذف سلسله مراتب، یک تابع را درخط می‌کند. برای فعال کردن بهینه‌سازی منطق در فرای مرزهای تابع و بهبود تاخیر و وقفه با کاهش سربار فراخوانی تابع استفاده می‌شود.
INTERFACE	مشخص می‌کند درگاه‌های زبان انتقال ثبات چگونه از توصیف تابع ساخته شوند.
LATENCY	یک محدودیت حداقلی و حداکثری برای تاخیر مشخص می‌کند.
LOOP_FLATTEN	حلقه‌های تودرتو را به یک حلقه‌ی منفرد تبدیل می‌کند تا تاخیر را بهبود دهد.
LOOP_MERGE	حلقه‌های متوالی را ترکیب می‌کند تا تاخیر کلی را کاهش دهد، به اشتراک گزاری را افزایش دهد و بهینه‌سازی منطق ر بهبود بخشد.
LOOP_TRIPCOUNT	برای حلقه‌هایی که متغیرهای محدوده‌ای دارند استفاده می‌شود. یک پیش‌بینی از تعداد تکرارهای حلقه به دست می‌دهد. این رهنمود تاثیری در سنتز ندارد و فقط گزارش را تغییر می‌دهد.
OCCURRENCE	هنگامی که توابع یا حلقه‌ها خط لوله می‌شوند استفاده می‌شود تا مشخص کند کد در یک موقعیت مکانی خاص با نرخ کمتری از کد پایانی تابع یا حلقه اجرا می‌شود.

PIPELINE	تاخیر اولیه را با فراهم کردن امکان اجرای هم‌روند عملیات در داخل یک حلقه یا تابع، کاهش می‌دهد.
PROTOCOL	قسمتی از کد را به عنوان منطقه‌ی پروتکل مشخص می‌کند. از یک منطقه‌ی پروتکل می‌توان برای درست کردن دستی یک پروتکل رابط استفاده کرد.
RESET	این رهنمود برای اضافه یا حذف کردن بازنشانی بر روی یک متغییر حالت استفاده می‌شود.
RESOURCE	مشخص می‌کند که یک کتابخانه‌ی منبع خاص برای پیاده‌سازی یک متغییر (آرایه، عملیات ریاضی یا آرگومان تابع) در زبان انتقال ثبات استفاده شده است.
STREAM	مشخص می‌کند که یک آرایه خاص به صورت اول داخل اول خارج (FIFO) یا کانال حافظه‌ی RAM در طول بهینه‌سازی جریان داده، پیاده‌سازی شود.
UNROLL	حلقه‌های for را باز می‌کند تا به جای یک مجموعه از عملیات، چند عملیات مستقل بسازد.

جدول ۱- رهنمودهای بهینه‌سازی سنتز سطح بالای ویوادو

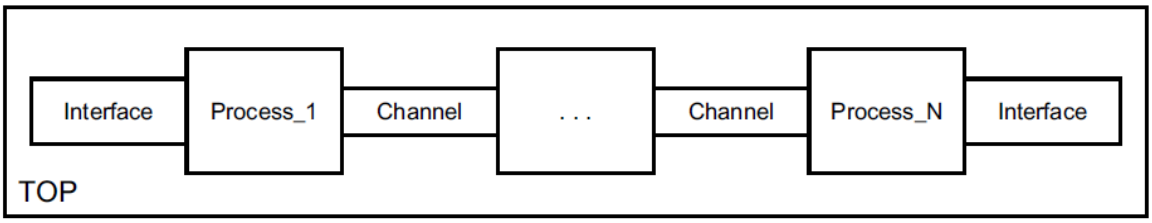
۲-۲-۳-۱- خط لوله‌سازی سطح کار: بهینه‌سازی جریان داده

بهینه‌سازی جریان داده با یک مجموعه از کارهای ترتیبی (توابع، حلقه‌ها یا هردو) هم‌چون شکل ۷ شروع می‌شود.



شکل ۷- توصیف عملکردی ترتیبی [۱۲]

با استفاده از این سری کارهای ترتیبی، بهینه‌سازی جریان داده یک معماری پردازشی موازی هم‌چون شکل ۸ می‌سازد. بهینه‌سازی جریان داده روشی قدرتمند برای بهبود گذردهی طرح است.

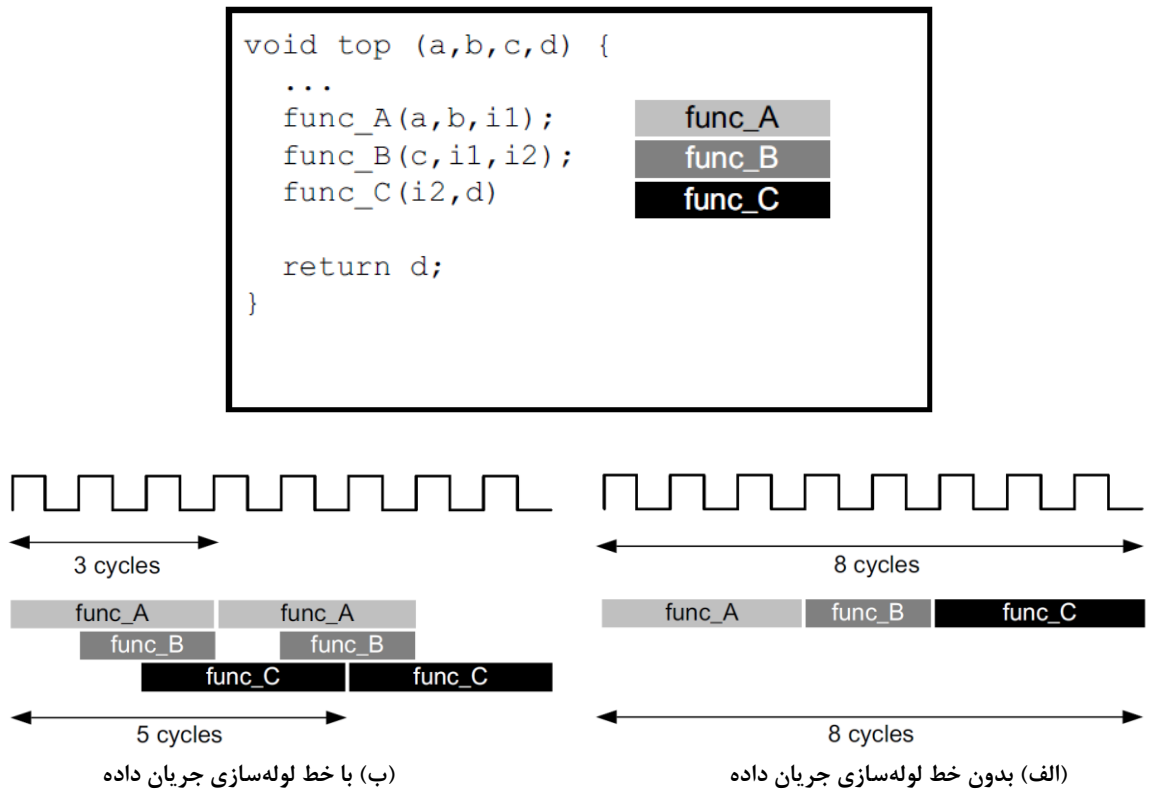


شکل ۸- معماری پردازش موازی [۱۲]

کانال‌های نمایش داده شده در شکل ۸ تضمین می‌کنند که لازم نیست یک کار، برای شروع شدن، تا پایان همه‌ی عملیات کار قبلی منتظر بماند. شکل ۹ نشان می‌دهد که بهینه‌سازی جریان داده چگونه اجازه می‌دهد تا اجزای کارها هم‌پوشانی داشته باشند و گذردهی کلی طرح افزایش و تاخیر آن کاهش یابد.

در مثال شکل ۹ بدون خط لوله‌سازی جریان داده (الف)، پیاده‌سازی ۸ چرخه نیاز دارد تا یک ورودی جدید بتواند توسط تابع func_A پردازش شود و ۸ چرخه هم نیاز است تا خروجی توسط تابع func_C نوشته شود.

در همین مثال با خط لوله‌سازی جریان داده (ب)، تابع func_A هر ۳ چرخه‌ی ساعت می‌تواند پردازش یک ورودی جدید را آغاز کند (که به معنی تاخیر آغازی کمتر است) و حالا تنها ۵ چرخه‌ی ساعت نیاز دارد تا مقدار نهایی خروجی را آماده کند (که به معنی تاخیر کمتر است).



شکل ۹- بهینه‌سازی جریان داده [۱۲]

بهینه‌سازی جریان داده محدودیت‌هایی هم دارد. برای اینکه این بهینه‌سازی بتواند کار کند، داده باید در طول طرح از کاری به کار دیگر در جریان باشد. شیوه‌های کدنویسی زیر جلوی بهینه‌سازی جریان داده را می‌گیرند:

- عدم رعایت تک تولیدکننده تک مصرف‌کننده
- پرش از روی کارها
- بازخورد بین کارها
- اجرای شرطی کارها
- ناحیه‌های حلقه با متغییرهای محدوده‌ای

برای این که ابزار سنتز سطح بالای ویوادو بتواند بهینه‌سازی جریان داده را انجام دهد، همه‌ی عناصر انتقالی بین کارها باید از مدل تک تولیدکننده تک مصرف کننده پیروی کنند. هر متغییر باید توسط یک کار تولید و تنها توسط یک کار مصرف شود.

تنها وقتی می‌توان از این بهینه‌سازی در حلقه‌ها استفاده کرد که محدوده‌ی حلقه ثابت باشد. یعنی اگر متغییرها محدوده‌ی حلقه را مشخص کنند نمی‌توان از این بهینه‌سازی استفاده کرد [۱۲].

۲-۲-۳-۲- بازکردن حلقه برای بهبود خط لوله‌سازی

به صورت پیش‌فرض حلقه‌ها در سنتز سطح بالای ویوادو بسته نگه داشته می‌شوند؛ به این معنی که حلقه‌ها هم‌چون یک موجودیت تنها در نظر گرفته می‌شوند. در این حالت تمام عملیات داخل حلقه با استفاده از منابع سخت‌افزاری یکسان برای هر تکرار حلقه پیاده‌سازی می‌شوند.

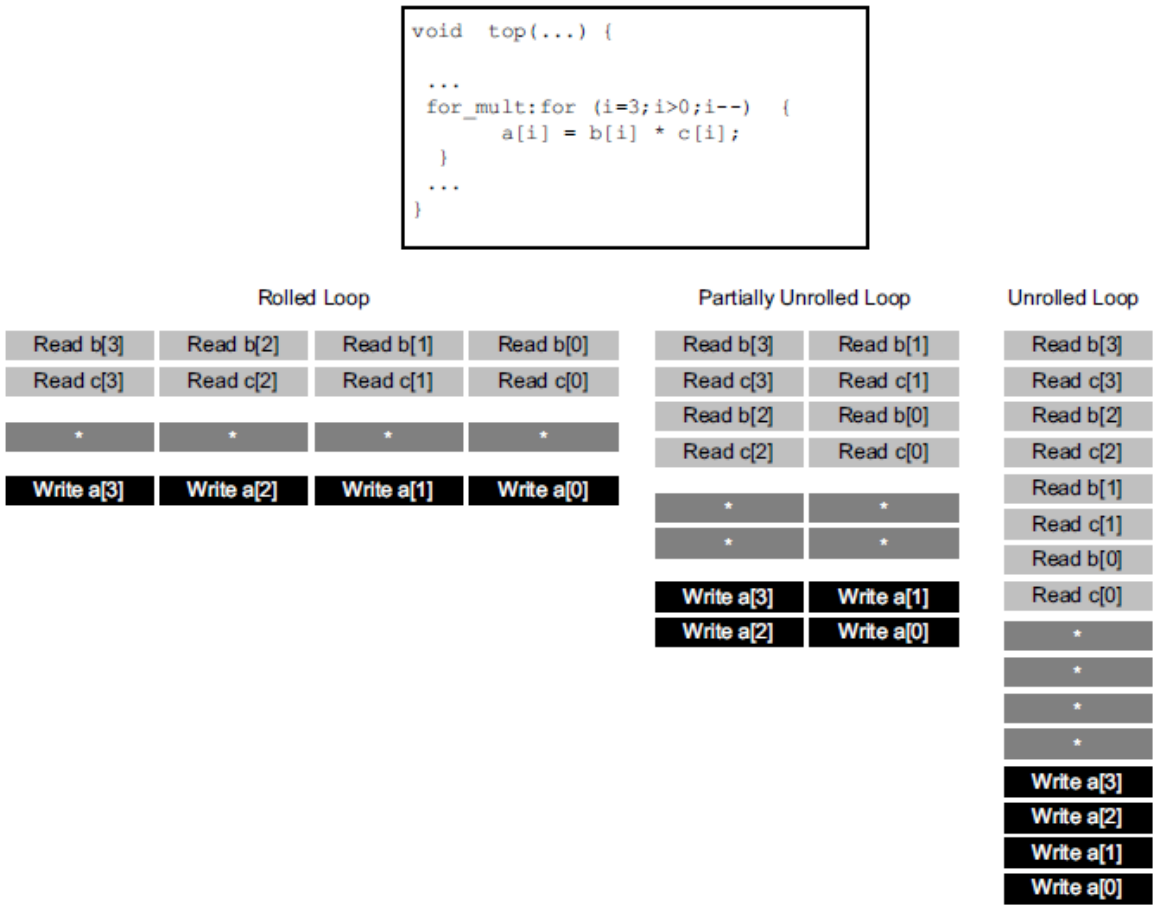
با استفاده از رهنمود UNROLL ابزار سنتز سطح بالای ویوادو قادر است که حلقه‌های for را به صورت کامل یا نیمه‌کاره باز کند.

شکل ۱۰ منافع قدرتمند باز کردن حلقه و مواردی که باید برای باز کردن حلقه در نظر گرفت را نشان می‌دهد. در این مثال فرض شده است که آرایه‌های a[i]، b[i] و c[i] به بلوک‌های حافظه RAM نگاشت شده‌اند. می‌بینیم که به آسانی می‌توان پیاده‌سازی‌های زیاد و متفاوتی را به کمک کاربرد باز کردن حلقه‌ها ساخت.

- حلقه‌ی بسته: وقتی که حلقه بسته باشد، هر تکرار در یک چرخه‌ی ساعت جدا اجرا می‌شود. این پیاده‌سازی چهار چرخه‌ی ساعت طول می‌کشد و تنها به یک ضرب‌کننده نیاز دارد. همچنین هر بلوک RAM می‌تواند تک‌درگاهه باشد.
- حلقه‌ی نیمه‌باز: در این مثال، حلقه نیمه‌کاره و با فاکتور ۲ باز شده است. این پیاده‌سازی دو ضرب‌کننده و RAM-های دودرگاهه نیاز دارد تا بتواند از دو خواندن یا دو نوشتن به هر بلوک RAM در یک چرخه‌ی ساعت پشتیبانی کند. این پیاده‌سازی دو چرخه‌ی ساعت

طول می‌کشد تا تمام شود. به این شکل تاخیر آغازی و تاخیر نسبت به حلقه‌ی بسته نصف می‌شوند.

- حلقه‌ی باز: در نسخه‌ی کاملا باز، همه‌ی عملیات حلقه می‌توانند تنها در یک چرخه‌ی ساعت انجام شوند. در عوض، این پیاده‌سازی به چهار ضرب‌کننده نیاز دارد. از آن مهم‌تر، در این پیاده‌سازی باید امکان چهار خواندن و چهار نوشتن هم‌زمان در یک چرخه‌ی ساعت ممکن باشد. از آنجا که یک بلوک حافظه‌ی RAM نهایتاً دو درگاه دارد، در این پیاده‌سازی آرایه‌ها باید تقسیم شوند.



شکل ۱۰- جزییات باز کردن حلقه [۱۲]

برای باز کردن حلقه می‌توان از رهنمود UNROLL روی هر حلقه‌ی جدا در طرح استفاده کرد یا رهنمود را بر روی یک تابع اعمال کرد که همه‌ی حلقه‌های داخل ناحیه‌ی آن تابع را باز می‌کند.

اگر حلقه‌ای کاملا باز شود و وابستگی‌های داده‌ای اجازه دهند، همه‌ی عملیات آن به صورت موازی اجرا می‌شوند. اگر عملیات یک تکرار از حلقه به نتیجه‌ی تکرار قبلی نیاز داشته باشد، نمی‌توانند موازی اجرا شوند اما به محض اینکه داده آماده شود اجرا می‌شود. باز شدن کامل حلقه به معنی رونویسی شدن منطق بدنه‌ی حلقه است.

۲-۳- برد زیبو (Zybo)

نام برد از حروف اول دو کلمه‌ی Zynq و Board آمده است. این برد غنی-ویژگی[۱]، آماده‌ی استفاده و یک پلتفورم توسعه‌ی مدار دیجیتال و نرم‌افزار نهفته سطح ورود[۲] است که بر پایه‌ی کوچک-ترین عضو خانواده‌ی زینک ۷۰۰۰ زایلینکس یعنی Z-7010 ساخته شده است. این تراشه بر پایه‌ی معماری قابل برنامه‌نویسی کامل سامانه بر تراشه استوار[۳] است. این برد پردازنده دو هسته‌ای ARM Cortex-A9 را با منطق سری هفتم آرایه‌ی دروازه‌ی قابل برنامه‌ریزی میدان زایلینکس ادغام کرده است. با مجهز بودن به مجموعه‌ی کاملی از لوازم جانبی ارتباطی و مولتی‌مدیا، این برد می‌تواند یک طراحی کامل سامانه را پشتیبانی کند [۱۳].

۲-۳-۱- ویژگی‌ها

طبق منبع [۱۳] برد زیبو ویژگی‌های زیر را دارد:

- پردازنده‌ی دو هسته‌ای Cortex-A9 با فرکانس کاری ۶۵۰ مگاهرتز
- کنترل‌کننده‌ی حافظه‌ی DDR3 با ۸ کانال DMA

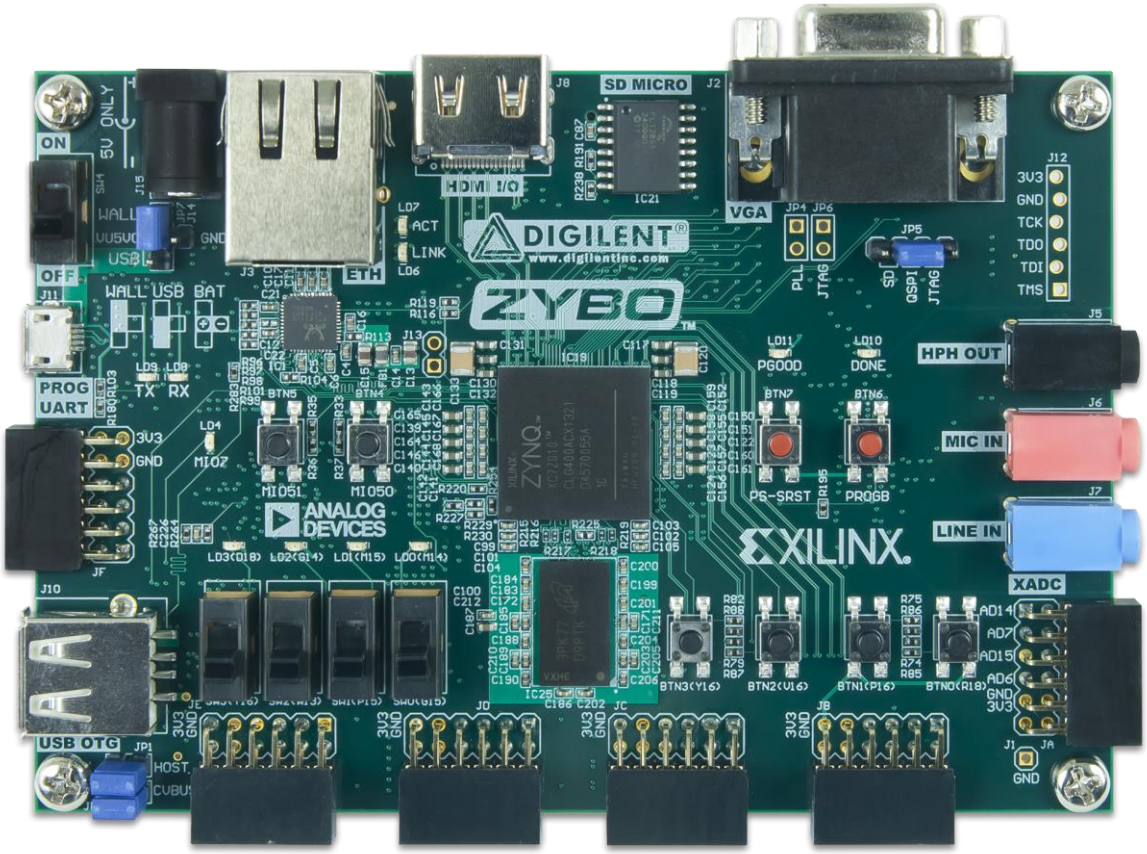
---

۱ Feature-rich

۲ Entry-level

۳ All Programmable System-on-Chip

- کنترل‌کننده‌ی لوازم جانبی با پهنای باند بالا: Ethernet یک گیگابایتی، USB 2.0، SDIO
- کنترل‌کننده‌ی لوازم جانبی با پهنای باند کم: SPI، UART، I2C،CAN
- منطق بازبرنامه‌پذیر معادل Artix-7 FPGA:
  - ۴۴۰۰ تکه‌ی منطقی، هرکدام با ۴ جدول جستوجوی ۶-ورودی و ۸ فلیپ-فلاپ
  - ۲۴۰ کیلوبایت بلوک RAM سریع
  - دو کاشی مدیریت ساعت
  - ۸۰ تکه‌ی DSP
  - ساعت داخلی با فرکانس بیشتر از ۴۵۰ مگاهرتز
  - مبدل آنالوگ به دیجیتال روی برد (XADC)
- ۵۱۲ مگابایت حافظه‌ی x32 DDR3 با پهنای باند ۱۰۵۰ مگابیت بر ثانیه
- درگاه HDMI دو نقشه
- درگاه منبع VGA ۱۶ بیت بر پیکسل
- درگاه Ethernet PHY سه حالته (۱ گیگابیت/ ۱۰۰ مگابیت/ ۱۰ مگابیت)
- درگاه حافظه‌ی MicroSD
- درگاه OTG USB 2.0
- EEPROM خارجی
- مفسر صوتی با خروجی هدفون و میکروفون
- رابط سری ۱۲۸ مگابیتی Flash
- مبدل UART به USB و JTAG روی برد
- ۶ دکمه‌ی فشاری، ۴ سوییچ و ۵ لامپ LED



شکل ۱۱- تصویر برد زیبو [۱۳]

۲-۴- جمع‌بندی

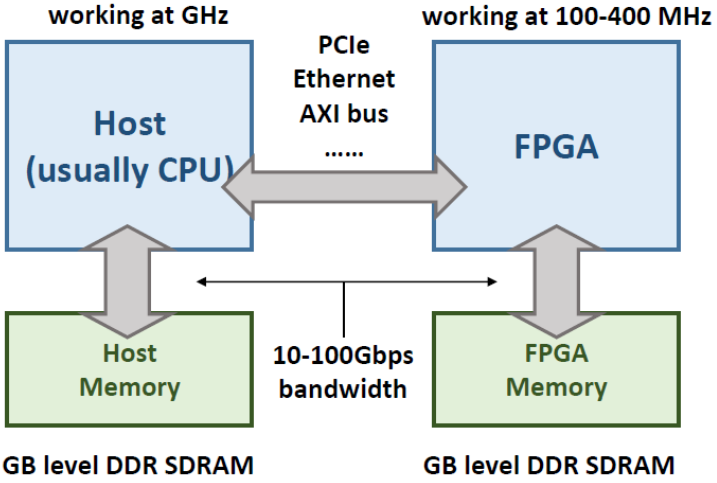
در این فصل ابتدا به مفاهیم پایه‌ی یادگیری ماشین پرداختیم و سپس با شبکه‌های عصبی و به خصوص شبکه‌های عصبی کانولوشنی آشنا شدیم و اجزای آن‌ها را بررسی کردیم. در ادامه به معرفی ابزار سنتز سطح بالای ویوادو پرداختیم و چرخه‌ی طراحی به کمک این ابزار را تشریح کردیم. با برخی روش-های پایه‌ای بهینه‌سازی گذردهی و تاخیر در این ابزار آشنا شدیم و مثال‌هایی از آن‌ها دیدیم. در نهایت نیز به معرفی برد زیبو و ویژگی‌های آن پرداختیم.

فصل سوم

مرور کارهای مرتبط

مرور کارهای مرتبط

در شتاب‌دهنده‌های مبنتی بر FPGA برای شبکه‌های عصبی معمولا از یک معماری مطابق شکل ۱۲ استفاده می‌کنند. سیستم معمولا از یک واحد پردازش مرکزی به عنوان میزبان و یک بخش FPGA تشکیل می‌شود. میزبان و FPGA می‌توانند با حافظه‌ی خارجی خود کار کنند و یا از طریق اتصال موجود به حافظه‌ی یک‌دیگر دسترسی پیدا کنند. در اکثر موارد شتاب‌دهنده‌ی شبکه‌ی عصبی بر روی بخش FPGA پیاده‌سازی می‌شود و با استفاده از نرم‌افزار روی میزبان کنترل می‌شود [۱].



شکل ۱۲- یک معماری معمول برای یک شتاب‌دهنده‌ی مبتنی بر FPGA [۱]

تراشه‌های معمول FPGA واحدهای ذخیره‌سازی بزرگی مثل رجیسترها و SRAM-ها دارند اما این حافظه‌ها با توجه به اندازه‌ی شبکه‌های عصبی مطرح بسیار کوچک هستند. مدل‌های رایج ۱۰۰ تا ۱۰۰۰ مگابایت پارامتر دارند در حالی که بزرگ‌ترین تراشه‌های FPGA موجود ۵۰ مگابایت حافظه‌ی SRAM روی تراشه دارند. برای پوشش این فاصله لازم است تا از حافظه‌های خارجی مثل DDR SDRAM استفاده شود اما پهنای باند و مصرف انرژی این حافظه‌ها کارایی سیستم را محدود می‌کند.

ظرفیت محاسباتی FPGA نسبتاً زیاد است و بزرگ‌ترین تراشه‌ها قادر به محاسبه‌ی تا حداکثر ۱۰ ترافلاپ بر ثانیه (عملیات نقطه‌ی شناور بر ثانیه) هستند. اما در تراشه‌های رده‌پایین این عدد به ۲۰

گیگافلاپ در ثانیه کاهش می‌یابد که برای پشتیبانی از پردازش فیلم بی‌درنگ برای کاربردهای قابل‌حمل کافی نیست.

با وجود این چالش‌ها، محققین روش‌های بهینه‌سازی جالبی را در سطوح الگوریتم و معماری برای طراحی شتاب‌دهنده‌های شبکه‌های عصبی با کارایی بالا روی FPGA ارائه داده‌اند که در این بخش ابتدا به معرفی آن‌ها می‌پردازیم و پس از آن نیز دو مورد از کارهای اخیری را که از FPGA برای شتاب-دهی فاز استنتاج در شبکه‌های عصبی استفاده کرده‌اند، معرفی می‌کنیم؛ که در مورد نخست از Intel FPGA SDK و در مورد دوم از Vivado HLS برای پیاده‌سازی شبکه استفاده شده است.

۳-۱- روش‌های بهینه‌سازی در طراحی شتاب‌دهنده‌های شبکه‌های عصبی

۳-۱-۱- فشرده‌سازی سخت‌افزار-محور مدل

یکی از راهکارهای طراحی شتاب‌دهنده‌های سریع و با کارایی بالا، بهینه‌سازی مدل‌های شبکه-های عصبی است. یک مدل بزرگ‌تر معمولا به دقت بیشتر می‌انجامد و در نتیجه یک مصالحه بین دقت سیستم و سرعت و توان مصرفی سخت‌افزار وجود دارد. کارهای اخیر تلاش می‌کنند تا به طور مستقیم تاخیر پردازش را با پیدا کردن یک ساختار خوب برای شبکه بهینه کنند [۱۴] یا از برخی لایه‌ها در زمان اجرا برای صرفه‌جویی در محاسبات صرفه نظر کنند [۱۵]. البته این روش‌ها خیلی در طراحی سخت‌افزار تاثیرگذار نیستند. سایر روش‌ها تلاش می‌کنند تا با فشرده‌سازی مدل‌های موجود شبکه‌های عصبی به مصالحه‌ی مورد نظر برسند. آن‌ها تلاش می‌کنند تا تعداد وزن‌ها یا تعداد بیت‌های هر وزن یا فعال‌ساز را کم کرده و به این شکل پیچیدگی محاسباتی و حافظه‌ای را کاهش دهند. در ادامه، این روش‌ها را معرفی می‌کنیم.

۳-۱-۱-۱- گسسته‌سازی داده

یکی از راهکارهای رایج فشرده‌سازی مدل گسسته کردن وزن‌ها و فعال‌سازهاست. این مقادیر در شبکه‌های عصبی معمولا به صورت اعداد ممیز شناور نمایش داده می‌شوند. روش‌های گسسته‌سازی داده سعی می‌کنند این نمایش را با بازنمایی ممیز ثابت با بیت‌های کم یا حتی مجموعه‌ای کوچک از مقادیر حاصل از آموزش جایگزین کنند. از یک طرف استفاده از بیت‌های کمتر برای هر فعال‌ساز یا وزن نیازمندی پهنای باند و حافظه‌ی سیستم را کاهش می‌دهد و از طرف دیگر، استفاده از بازنمایی ساده‌تر هزینه‌ی سخت‌افزاری هر عمل محاسباتی را کم می‌کند. محققین در [۱۶ و ۱۷ و ۱۸] از این روش استفاده کرده‌اند.

۳-۱-۱-۲- کاهش وزن‌ها

راهکار فشرده‌سازی دیگر کاهش تعداد وزن‌هاست. یک روش تقریب زدن ماتریس وزن‌ها با یک بازنمایی با رتبه‌ی کم است. در [۱۹] نویسندگان ماتریس وزن‌های یک لایه‌ی کاملا متصل را به روش تجزیه مقدار منفرد فشرده کردند که در آن با کاهش ۶۴ درصدی اندازه‌ی بزرگ‌ترین لایه‌ی کاملا متصل در شبکه‌ی VGG دقت طبقه‌بندی تنها ۰٫۰۴ درصد کاهش پیدا کرد. در [۲۰] نویسندگان از روش مشابهی در لایه‌های کانولووشنی استفاده کردند که سرعت چهار برابر بهترین مدل‌های CNN بر روی مجموعه‌ی دادگان ImageNet را تنها با ۰٫۹ کاهش در دقت به ارمغان آورد. روش دیگر کاهش وزن‌ها هرس کردن است که در آن وزن‌های با مقادیر صفر یا مقادیر مطلق کوچک کنار گذاشته می‌شوند. نویسندگان در [۲۱ و ۲۲] به این روش پرداخته‌اند که با حذف تقریبا ۹۰ درصد وزن‌ها دقت کمتر از ۱ درصد کاهش داشته است.

۳-۱-۲- طراحی سخت‌افزار: معماری کارآمد

روش دیگر بهینه‌سازی در شتاب‌دهنده‌های شبکه‌های عصبی، استفاده از تکنیک‌های سخت-افزاری برای رسیدن به کارایی بالا و توان مصرفی کم است. این شیوه‌ها به سه سطح تقسیم می‌شوند: سطح واحد محاسبه، سطح باز کردن حلقه و سطح سیستم.

۳-۱-۲-۱- سطح واحد محاسبه

طراحی سطح واحد محاسبه، اوج عملکرد شتاب‌دهنده‌ی شبکه‌ی عصبی را تحت تاثیر قرار می-دهد. منابع موجود در FPGA محدود هستند؛ در نتیجه طراحی کوچک‌تر واحد محاسبه به معنی تعداد بیشتری واحد محاسبه و در نتیجه افزایش اوج عملکرد خواهد بود. طراحی دقیق آرایه واحد محاسبه همچنین می‌تواند فرکانس کاری سیستم را افزایش دهد. برخی شیوه‌ها که در این گروه قرار می‌گیرند در زیر آمده‌اند:

- واحد محاسبه با پهنای بیت کم
- روش کانولووشن سریع
- روش‌های بهینه‌سازی فرکانس

۳-۱-۲-۲- سطح باز کردن حلقه

لایه‌های کاملا متصل و لایه‌های کانولووشنی اکثر نیازمندی‌های محاسباتی و حافظه‌ای در شبکه‌های عصبی را ایجاد می‌کنند. تابع کانولووشن در پیاده‌سازی به شکل حلقه‌های تو در تو در می‌آید. لایه‌ی کاملا متصل را نیز می‌توان با یک لایه‌ی کانولووشنی با اندازه‌ی نقشه‌ی ویژگی و هسته‌ی ۱×۱ نشان داد. علاوه بر این حلقه‌ها، موازی‌سازی با فراخوانی دسته‌ای ورودی‌ها در یک حلقه‌ی بیرونی نیز ممکن است. در نتیجه فنون باز کردن حلقه‌ها را می‌توان در شتاب‌دهنده‌های این نوع شبکه‌ها به کار برد. دو فاکتور مهم در این سطح در زیر آمده‌اند:

- انتخاب پارامترهای باز کردن حلقه

تعداد تکرارهای موازی شده روی سخت‌افزار «پارامتر باز کردن حلقه» نام دارد. انتخاب نامناسب این پارامتر ممکن است به کاهش شدید بهینگی استفاده از سخت‌افزار منجر شود. در مدل‌های CNN ابعاد حلقه‌ها در لایه‌های مختلف متفاوت است و در کنار مشکل کاهش بهینگی استفاده از سخت‌افزار، مسیر داده و طراحی حافظه‌ی روی تراشه نیز تحت تاثیر فنون باز کردن حلقه قرار می‌گیرند.

- انتقال داده و طراحی حافظه‌ی روی تراشه

در کنار موازی‌سازی، سیستم حافظه‌ی روی تراشه باید به طور کارا و بهینه دادگان لازم را به همه‌ی واحدهای محاسبه در هر چرخه برساند.

۳-۱-۲-۳- سطح طراحی سیستم

در سطح سیستم مباحث زیر مورد توجه هستند:

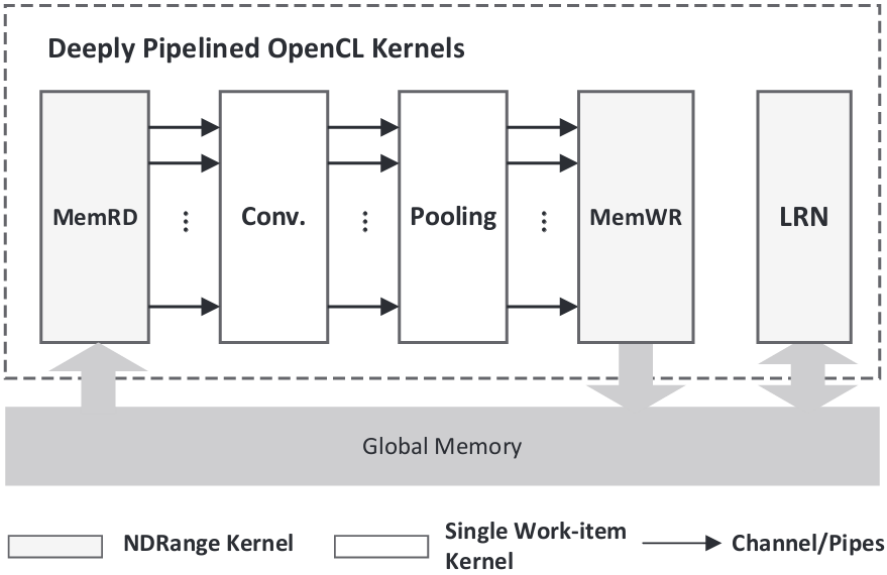
- مدل روف‌لاین (مدل سقفی)
- تبادل و کاشی‌کاری حلقه
- برنامه‌ریزی بین لایه‌ای
- تنظیم الگوی دسترسی به داده‌ها

۳-۲-دو نمونه از پیاده‌سازی‌های شتاب‌دهنده‌های شبکه عصبی روی FPGA

۳-۲-۱- PipeCNN [۲۳]

PipeCNN یک پیاده‌سازی منبع‌باز از یک شتاب‌دهنده‌ی CNN است که از Intel FPGA SDK for OpenCL استفاده می‌کند. دستاورد اصلی این کار یک ساختار بهینه از هسته‌های خط لوله-ای برای شبکه‌های عصبی کانولووشنی در مقیاس بزرگ است. شبکه‌های AlexNet و VGG در این معماری پیاده‌سازی و تست شدند. معماری طراحی شده در این روش به صورت شکل ۱۳ است. این معماری دو هسته‌ی MemRD و MemWR دارد که کارکرد آن‌ها انتقال داده و وزن‌ها بین حافظه‌ی سراسری و FPGA است. انتقال داده در بین این دو هسته با توسعه‌ی کانال و روی تراشه انجام می‌شود.

هسته‌ی هنجارسازی پاسخ محلی از دیگر هسته‌ها جدا است زیرا ممکن است به چند الگوی دسترسی به حافظه نیاز داشته باشد.



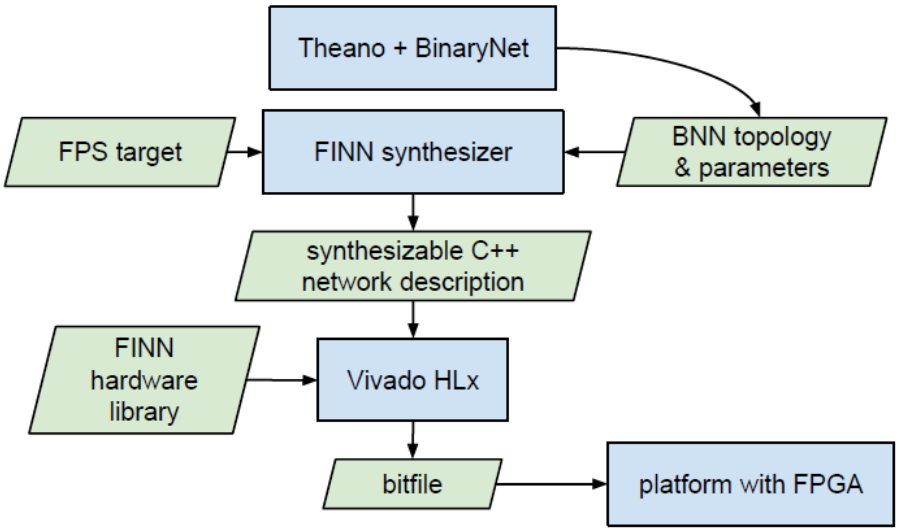
شکل ۱۳- معماری PipeCNN [۲۳]

نویسندگان در این کار گزارش کرده‌اند که کوتاه‌ترین زمان طبقه‌بندی برای شبکه AlexNet ۴۳ میلی ثانیه و برای شبکه‌ی VGG-16 ۷۱۸ میلی ثانیه بوده است.

۳-۲-۲- FINN [۲۴]

FINN یک فریم‌ورک برای استنتاج سریع و مقیاس‌پذیر در شبکه‌ی عصبی باینری است. در حالی که PipeCNN از دقت استاندارد شناور ۳۲ استفاده می‌کرد، در این کار شبکه‌ی باینری به کار گرفته شده است. محققین از ابزار Vivado HLS از شرکت Xilinx [۲۵] برای پیاده‌سازی استفاده کرده‌اند. شکل ۱۴ نحوه‌ی استفاده از یک شبکه‌ی باینری آموزش دیده و توپولوژی آن را برای توصیف C++ یک معماری جریان ناهمگن نشان می‌دهد.

در این کار، محققین با استفاده از یک پلتفورم FPGA ZC706 با مصرف توان کمتر از ۲۵ وات توانستند به گذردهی ۱۲,۳طبقه‌بندی در ثانیه با تاخیر ۰,۳۱ میکروثانیه و دقت ۹۵,۸ درصد بر روی مجموعه‌ی دادگان MNIST برسند.



شکل ۱۴- ساخت یک شتاب‌دهنده‌ی FPGA از یک شبکه‌ی باینری [۲۴]

۳-۳- جمع‌بندی

در این فصل به طور اختصاصی به شتاب‌دهی شبکه‌های عصبی بـه کمـک FPGA پـرداختیم و برخی چالش‌های آن را مطرح کردیم. معماری کلی این نوع طراحی‌ها را بررسی کردیم و روش‌های کلـی بهینه‌سازی در این شتاب‌دهنده‌ها را مرور کردیم. در پایان دو نمونه از پیاده‌سازی‌های شبکه‌های عصـبی بر روی FPGA را بررسی کردیم و دیدیم که با بهینه‌سازی و طراحی خط لوله‌ای به نتایج قابل توجهی از نظر تاخیر و مصرف توان برای شبکه‌های مطرح مثل AlexNet و VGG دست یافته‌اند.

فصل چهارم

پیاده‌سازی

پیاده‌سازی

همان‌طور که پیشتر اشاره کردیم، یکی از مهم‌ترین و پرکاربردترین شبکه‌های عصبی موجود شبکه‌ی عصبی کانولووشنی است. به کمک FPGA-ها، معماری شتاب‌دهنده و عرض مسیر داده می‌تواند دقیقا متناسب با شبکه‌ی هدف طراحی شود که مزیتی نسبت به استفاده از واحدهای پردازش گرافیکی یا طراحی مدارهای ASIC است. همچنین قابلیت بازپیکربندی در FPGA-ها اجازه می‌دهد تا طرح یک شتاب‌دهنده با ترکیب کردن یافته‌های جدید علمی و راهکارهای نوین (مثل قابلیت دست‌یابی به دقت[۱] تشخیص زیاد سیستم با دقت ۲ بیتی[۲] داده [۲۶]) سازگار باشد [۴].

اخیراً سنتز سطح بالا یک روش پیاده‌سازی نسبتاً بالغ برای FPGA-ها است که اجازه می‌دهد یک توصیف نرم‌افزاری به سخت‌افزار سنتز شود. سنتز سطح بالا با افزایش سطح انتزاع در طراحی و اشکال-زدایی، هزینه‌های یک‌باره‌ی مهندسی[۳] را کاهش می‌دهد [۴].

در این پروژه، هدف پیاده‌سازی تابع کانولووشن و کاهش بعد (ادغام) مورد استفاده در شبکه‌های عصبی کانولووشنی برای برد زیبو بوده است. برای این پیاده‌سازی همچون [۲۴] از ابزار سنتز سطح بالای Vivado استفاده شده است. در این روش، مدل رفتاری مورد نظر برای سنتز روی FPGA توسط یک زبان کنترل جریان مثل C/C++ توصیف می‌شود و ابزار سنتز سطح بالا، توصیف رفتاری ما را به سخت-افزار نگاشت کرده و کد زبان توصیف سخت‌افزار را تولید می‌کند. البته نکته‌ی قابل توجه در این پروژه این است که هدف تنها به پیاده‌سازی این دو لایه‌ی شبکه‌ی عصبی خلاصه نشده است؛ بلکه سعی شده تا این پیاده‌سازی با توجه به نکات و راهکارهای بهینه‌سازی اشاره شده در فصل‌های دوم و سوم بهینه و هوشمندانه باشد. همچنین در انتها تحلیلی از تاثیر این راهکارهای بهینه‌سازی بر عملکرد سیستم ارائه شده است.

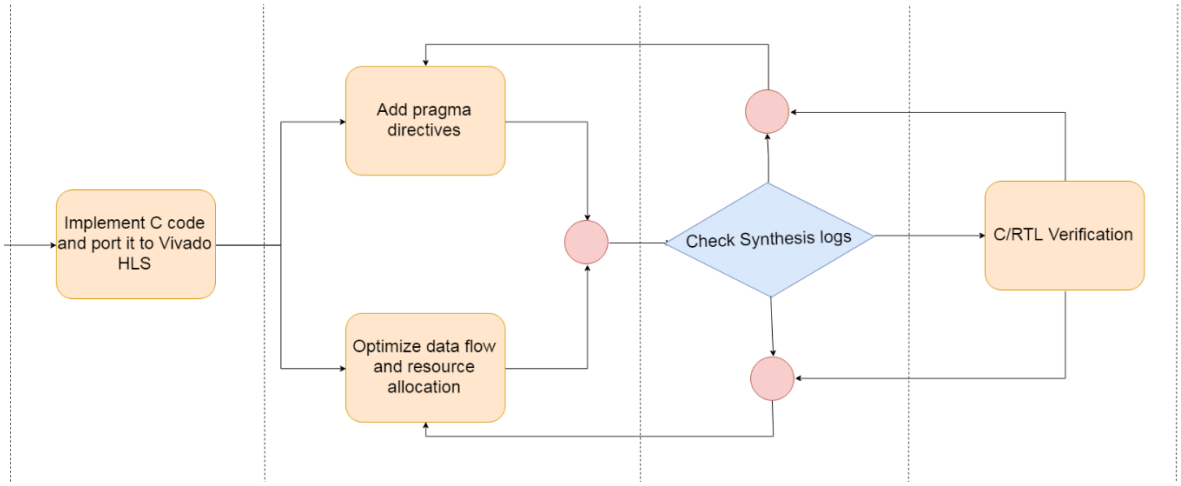
معماری پروژه به این صورت است که ۳ فیلتر ۳×۳ به ماتریس تصویر اعمال شده، کانولووشن محاسبه شده، سپس کاهش بعد انجام شده و نتایج در ۳ ماتریس ذخیره می‌شود.

---

[۱] Accuracy

[۲] 2-bit precision

[۳] Non-Recurring Engineering costs



شکل ۱۵- چرخه‌ی کار در پروژه

۴-۱- معماری پروژه

در این فصل به تشریح پیاده‌سازی هسته‌ی مالکیت معنوی و بخش‌های مختلف آن می‌پردازیم و انتها روش آزمایش این هسته را هم به صورت شبیه‌سازی و هم اجرا بر روی خود برد زیبو توضیح می‌دهیم.

۴-۱-۱- فیلترهای اعمال شده

در این پروژه برای اعمال کانولوشن روی تصویر اولیه از سه فیلتر تشخیص لبه، منبت‌کاری و تیزکردن استفاده شده است که در ادامه به معرفی آن‌ها می‌پردازیم.

۴-۱-۱-۱- فیلتر تشخیص لبه سوبل[۱]

این فیلتر در پردازش تصویر در الگوریتم‌های تشخیص لبه بسیار پرکاربرد است و تصویری می‌سازد که بر لبه‌ها تاکید می‌کند. اسم این فیلتر از روی نام ایروین سوبل[۲] از محققان آزمایشگاه هوش مصنوعی

---

۱ Sobel

۲ Irwin Sobel

دانشگاه استنفورد گرفته شده است. در هر نقطه از عکس نتیجه‌ی این فیلتر یا بردار گرادیان متناظر یا نرم این بردار است. این فیلتر بر اساس کانولوشن عمل می‌کند.

ماتریس این فیلتر راه در زیر می‌بینیم:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

۴-۱-۱-۲- فیلتر منبت‌کاری[۱]

با اعمال این فیلتر بر یک عکس، هر پیکسل تصویر یا با یک برجسته‌سازی یا با یک سایه جایگزین می‌شود. این جایگزینی بر اساس روشنایی و تاریکی تصویر اصلی مشخص می‌شود. نواحی با کنتراست[۲] کم با یک پس‌زمینه‌ی طوسی جایگزین می‌شوند. تصویر فیلتر شده نرخ تغییر رنگ در هر نقطه از تصویر اصلی را نشان می‌دهد. تصویر حاصل از اعمال این فیلتر بر عکس اولیه معمولا یک منبت‌کاری فلزی از تصویر اصلی را تداعی می‌کند و علت این نام‌گذاری هم همین است.

ماتریس این فیلتر را در زیر می‌بینیم:

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

---

۱ Emboss

۲ Contrast

۴-۱-۱-۳- فیلتر تیز کردن[۱] (شفافیت)

اعمال این فیلتر بر عکس ورودی، تصویری می‌سازد که از تصویر اصلی شفاف‌تر (کمتر تار) است. تصویر حاصل هرچند که شفاف‌تر است اما در نمایش موضوعات تصویر به درستی تصویر اصلی نیست. این فیلتر لبه‌ها، اختلال‌ها و لکه‌ها را تشدید می‌کند و همچنین ممکن است در نواحی یکدست مثل آسمان و سطح دریا اختلال اضافه کند.

ماتریس این فیلتر را در زیر می‌بینیم:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

۴-۱-۲- ورودی‌ها و خروجی‌ها

مالکیت معنوی مد نظر در این پروژه می‌بایست تصویر ورودی را دریافت کرده و سه تصویر خروجی حاصل از اعمال عملیات کانولوشن و کاهش بعد را در خروجی بگذارد. از همین رو، ورودی به شکل یک آرایه IN_H×IN_W و سه خروجی به شکل آرایه‌های OUT_H×OUT_W انتخاب شدند. در نتیجه معرفی[۲] تابع اصلی پروژه به صورت زیر خواهد بود.

```
void conv(uint8_t image_in[IN_H*IN_W], uint8_t image_out1[OUT_H*OUT_W], uint8_t i
mage_out2[OUT_H*OUT_W], uint8_t image_out3[OUT_H*OUT_W]);
```

۱ Sharpen

۲ Declaration

۴-۱-۳- اندازه‌ی تصاویر

با توجه به ابعاد نمایشگر سامانه‌ای که توسعه‌ی پروژه بر روی آن انجام شده است ابتدا از تصاویر بزرگ مطابق با وضوح نمایشگر این سامانه یعنی اندازه‌ی ۷۶۸×۱۳۶۶ استفاده شد. سپس با توجه به منابع موجود روی برد هدف (زیبو) و در نظر گرفتن این نکته که بهینه‌سازی تاخیر مدار به افزایش مساحت آن می‌انجامد، اندازه‌های متفاوت مورد آزمایش قرار گرفتند. در انتها نیز همان‌طور که در بخش ۴-۳ می‌بینیم اندازه تصویر ورودی برای اجرا بر روی برد ۱۰×۱۰ در نظر گرفته شد که اندازه تصاویر خروجی تابع با توجه به اعمال کاهش بعد ۲×۲ برابر با ۵×۵ خواهد بود.

برای کاهش منابع حافظه‌ای مورد استفاده و تمرکز بر عملکرد توابع کانولوشن و کاهش بعد، تصاویر به صورت سیاه سفید در نظر گرفته شده‌اند. از آنجایی که هر پیکسل در یک تصویر سیاه سفید مقداری بین ۰ تا ۲۵۵ دارد که میزان روشنایی آن پیکسل را مشخص می‌کند، در این معماری برای بازنمایی تصاویر از آرایه‌هایی از مقادیر بدون علامت صحیح با طول ۸ بیت استفاده کردیم که بهینه‌ترین بازنمایی ممکن برای کار مورد نظر ما است؛ زیرا مقادیر بین ۰ تا ۲۵۵ را می‌توان با ۸ بیت نمایش داد. البته برخی مقادیر مورد استفاده در فیلترها علامت منفی دارند که به همین جهت برای تعریف هسته‌ی این فیلترها از مقادیر با علامت استفاده شده است.

۴-۱-۴- تابع بالا[1] (اصلی)

برای انجام عملکرد سیستم و استفاده از توابع از پیش نوشته شده و داده‌ساختارهای مناسب در روش برنامه‌نویسی سنتز سطح بالای ویوادو، از کتابخانه‌های hls_video و stdint استفاده شده است.

روند کلی این تابع به این صورت است که ابتدا آرایه‌ی ورودی تابع را به کمک توابع موجود در کتابخانه‌ی hls_video به داده‌ساختار Mat که در پروژه‌های پردازش تصویر به کمک سنتز سطح بالا استفاده از آن رایج است، تبدیل می‌کنیم. این داده‌ساختار در واقع ماتریسی است که اکثر توابع کتاب‌خانه-های سنتز سطح بالای ویوادو برای عمل بر روی آن نوشته شده‌اند. در همین ابتدا این ماتریس را در دو

[1] Top Function

نمونه‌ی دیگر رونوشت می‌کنیم تا محدودیت‌های خواندن از یک حافظه بر عملکرد سیستم تاثیر منفی نگذارد. البته این موضوع در پروژه‌های بزرگ‌تر و پیچیده‌تر می‌تواند حائز اهمیت باشد. علت دیگر این کار این است که توابع موجود در کتابخانه‌های HLS معمولا داده‌های موجود در داده‌ساختار ماتریسی ذکر شده را مصرف می‌کنند و پس از فراخوانی آن‌ها این داده‌ساختارها تخلیه شده و برای فراخوانی بعدی قابل استفاده نیستند. در گام بعد، سه بافر پنجره‌ای ۳×۳ برای قرار گیری فیلترها تعریف می‌کنیم و عناصر ماتریس‌های معرفی شده در بخش ۴-۱-۱ را در آن‌ها قرار می‌دهیم. همچنین به یک نقطه تحت عنوان لنگر[۱] نیاز داریم که در حین اعمال فیلترها بر تصویر اولیه، مشخص می‌کند هسته‌ی فیلتر نسبت به پیکسلی که در حال پردازش آن است چگونه قرار بگیرد. به طور معمول مرکز هسته را بر روی پیکسل تحت پردازش قرار می‌دهند که در این پروژه نیز همین کار انجام شده است. در این مرحله ماتریس ورودی، ماتریسی برای ذخیره‌ی خروجی، بافر پنجره‌ای و نقطه‌ی لنگر را به تابع فیلتر دو بعدی از کتابخانه‌ی hls_video می‌دهیم تا محاسبه‌ی کانولوشن را انجام دهد. در گام آخر تصویر فیلتر شده را به تابع کاهش بعد می‌دهیم تا ابعاد تصویر را کاهش دهد.

۴-۱-۵- تابع کاهش بعد

آرگومان‌های این تابع یک آرایه به عنوان ورودی و یک آرایه با اندازه‌ی یک چهارم آن برای ذخیره‌ی خروجی هستند. عملکرد این تابع بسیار ساده است. دو حلقه‌ی تو در تو (برای شبیه‌سازی حرکت روی ماتریس دو بعدی) روی آرایه ورودی حرکت می‌کنند. هر بار چهار خانه تشکیل دهنده‌ی یک مربع در ماتریس تصویر خوانده می‌شود و بیشینه‌ی آن‌ها در خانه‌ی متناظر در ماتریس خروجی نوشته می‌شود.

[۱] Anchor

۴-۲- بررسی عملکرد سامانه و نیمکت آزمون در شبیه‌سازی

۴-۲-۱- نیمکت آزمون C

برای بررسی عملکرد سامانه نیاز به یک نیمکت آزمون بود تا ورودی‌ها را بـه تـابع اصـلی بدهـد و خروجی‌ها را دریافت کند.

روند کلی این نیمکت آزمون به این صورت است که ابتدا یک عکس را از حافظه‌ی سخت سامانه‌ی توسعه با توابع OpenCV می‌خواند و داده‌ساختار Mat را به عنوان خروجی دریافت مـی‌کنـد. سـپس بـه کمک توابع رونوشت حافظه[۱]، عناصر این داده‌ساختار را در آرایه‌ی قابل ارائه به تابع اصـلی رونویسـی مـی-کنیم. حال تابع اصلی را فراخوانی کرده و پس از اتمام آن خروجی‌ها را بار دیگر بـه داده‌سـاختار ماتریسـی تبدیل کرده و در نهایت با توابع OpenCV تصاویر خروجی را در حافظه‌ی سخت می‌نویسیم.

تصاویر زیر برای ارائه در این گزارش با اندازه‌های بزرگ ورودی ۱۰۰۰×۵۶۲ به سامانه داده شـده-اند.

شکل ۱۶- تصویر ورودی به نیمکت آزمون با اندازه‌ی ۱۰۰۰×۵۶۲

[۱] memcpy

شکل ۱۷- تصویر ۵۰۰×۲۸۱ فیلتر شده با هسته‌ی تشخیص لبه

شکل ۱۸- تصویر ۵۰۰×۲۸۱ فیلتر شده با هسته‌ی منبت‌کاری

شکل ۱۹- تصویر ۵۰۰×۲۸۱ فیلتر شده با هسته‌ی تیزکردن (شفاف‌سازی)

تصویر شکل ۱۶ به عنوان ورودی به تابع بالا داده شده است و همان‌طور که در اشکال ۱۷، ۱۸ و ۱۹ می‌بینیم فیلترهای اشاره شده بر روی آن‌ها اعمال شده است. نمونه‌ی دیگری از اجرای شبیه‌سازی بر روی یک تصویر ورودی را در ادامه در تصاویر ۲۰ تا ۲۳ می‌بینیم.

شکل ۲۰- تصویر ورودی به نیمکت آزمون با اندازه‌ی ۱۰۰۰×۵۶۲

شکل ۲۱- تصویر ۵۰۰×۲۸۱ فیلتر شده با هسته‌ی تشخیص لبه

شکل ۲۲- تصویر ۵۰۰×۲۸۱ فیلتر شده با هسته‌ی منبت‌کاری

شکل ۲۳- تصویر ۵۰۰×۲۸۱ فیلتر شده با هسته‌ی تیزکردن (شفاف‌سازی)

در ادامه خروجی‌های تصویر شکل ۱۶ را با سایز طراحی کوچک‌تر (ورودی ۱۶۴×۹۲ و خروجی ۸۲×۴۶) می‌بینیم.

(الف) خروجی تشخیص لبه

(ب) خروجی منبت‌کاری

(ج) خروجی تیزکردن

شکل ۲۴- خروجی‌های طرح نهایی

۴-۲-۲- پیاده‌سازی کد نرم‌افزاری

همان‌طور که در بخش قبل دیدیم، خروجی‌های تولید شده همان خروجی‌های مورد انتظار هستند اما برای اثبات این موضوع نیاز داریم تا ماتریس خروجی را با کد نرم‌افزاری مشابه مقایسه کنیم و از یکسان بودن آن‌ها اطمینان حاصل کنیم. به همین جهت کدی مشابه برای انجام دو عمل کانولوشن و کاهش بعد به زبان C++ و با نرم‌افزار Visual Studio نوشتیم که به کمک توابع معادل در کتابخانه‌های OpenCV به انجام اعمال مورد نظر ما تنها با استفاده از پردازنده‌ی مرکزی می‌پردازد. این برنامه را بر روی سامانه‌ی توسعه که مبتنی بر ویندوز[۱] ۱۰ است اجرا کردیم. از ۱۰۰ تصویر از مجموعه داده‌ی MNIST [۱۰] به عنوان ورودی‌های این برنامه استفاده شد. ابتدا این تصاویر را به نیمکت آزمون نوشته شده برای ماژول سخت‌افزاری دادیم و خروجی‌های تولید شده توسط شبیه‌سازی را در حافظه‌ی سیستم توسعه نوشتیم که شامل سه تصویر خروجی برای هر ورودی و مجموعاً ۳۰۰ تصویر خروجی بودند. برنامه‌ی نرم‌افزاری نوشته شده ابتدا همان ۱۰۰ تصویر ورودی یاد شده از مجموعه داده‌ی MNIST را خوانده و عمل کانولوشن و کاهش بعد را با همان فیلترهای استفاده شده در ماژول سخت‌افزاری انجام می‌دهد. سپس فایل‌های تصاویر حاصل از نیمکت آزمون را نیز به این برنامه می‌دهیم و برنامه آن‌ها را از روی حافظه‌ی سیستم توسعه خوانده و عناصر ماتریس‌های آن‌ها را تک به تک با عناصر ماتریس‌های تصاویر خروجی که خود به صورت نرم‌افزاری به دست آورده است مقایسه می‌کند و تعداد پیکسل‌های مغایر در کل مجموعه‌ی ۳۰۰ تصویر را در خروجی نمایش می‌دهد. پس از اجرای این روند مشاهده شد که که تصاویر خروجی حاصل از شبیه-سازی نیمکت آزمون کد HLS و خروجی حاصل از اجرای برنامه‌ی نرم‌افزاری زبان C++ دقیقاً یکسان هستند.

۴-۲-۳- شبیه‌سازی توأم C و RTL

از طرفی دیگر برای اینکه مطمئن باشیم که خروجی‌های حاصل از کدهای HDL حاصل از سنتز ماژول سخت‌افزاری در شبیه‌سازی پس از سنتز نیز درست هستند، از شبیه‌سازی توأم زبان C و RTL

[۱] Windows

استفاده کردیم. نیمکت آزمون را مطابق نیاز تغییر دادیم تا خروجی ماژول سخت‌افزاری را با داده‌های طلایی مقایسه کند. منظور از داده‌های طلایی خروجی‌های درست و مورد انتظار است. این خروجی‌ها را از کد نرم‌افزاری ذکر شده در پاراگراف قبل به دست آورده و فایل‌های آن‌ها را به نیمکت آزمون می‌دهیم. در نیمکت آزمون پس از فراخوانی تابع بالا (اصلی) و تولید خروجی‌ها توسط این تابع، خروجی‌ها را برای همه‌ی ۳۰۰ مورد یاد شده با هم مقایسه می‌کنیم و اگر مغایرتی در حداقل یکی از پیکسل‌های تصاویر خروجی تابع اصلی (که همان خروجی شبیه سازی RTL است) با داده‌های طلایی وجود داشت مقدار «یک» و در غیر این صورت مقدار «صفر» را برمی‌گردانیم. بخش شبیه‌سازی توأم C و RTL در ابزار سنتز سطح بالای ویوادو تنها هنگامی گزارش موفقیت‌آمیز بودن شبیه‌سازی را می‌دهد که تابع main در نیمکت آزمون مقدار صفر را برگرداند. تصمیم‌گیری بالا نیز به همین دلیل بوده است. روشی که در اینجا برای سنجش صحت عملکرد استفاده شده است مطابق با مثال‌های خود شرکت Xilinx است که در هنگام نصب مجموعه‌ی نرم‌افزاری ویوادو می‌توان نسبت به نصب این مثال‌ها نیز اقدام کرد. در شکل ۲۵ می‌بینیم که نتیجه‌ی شبیه‌سازی هم‌زمان موفقیت‌آمیز بوده است.

Cosimulation Report for 'conv'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	3250	3250	3251	3250	3250	3251

شکل ۲۵- نتیجه‌ی شبیه‌سازی توأم C و RTL

در انتها برای اطمینان کامل از عملکرد سامانه پس از نشستن بر روی برد، خروجی‌ها را برای چند تصویر ورودی حاصل از اجرای ماژول بر روی خود برد می‌بینیم. در بخش بعدی به این موضوع پرداخته شده است.

۴-۳- بررسی عملکرد سامانه بر روی برد

برای سنجش صحت عملکرد هسته‌ی مالکیت معنوی پیاده‌سازی شده در بخش ۴-۱ پس از تمام مراحل سنتز و بر روی سخت‌افزار برد (مشابه حالتی که در نهایت و در عمل باید از آن استفاده کرد)، نیاز داریم تا این هسته را در یک طراحی جامع در خود نرم‌افزار ویوادو مورد استفاده قرار دهیم تا بتوانیم جریان بیت تولید کنیم و پس از برنامه‌ریزی برد با جریان بیت تولیدی، خروجی‌های حاصل از اجرای هسته بر روی برد را ببینیم.

۴-۳-۱- طرح ویوادو

در راستای طراحی این طرح جامع که شامل ماژول نوشته شده است، دو چالش بزرگ وجود داشت که مورد اول نحوه‌ی رد و بدل کردن داده بین هسته‌ی مالکیت معنوی طراحی شده و یک حافظه‌ی قابل خواندن و نوشتن بود و مورد دوم به اندازه‌ی تصاویر ورودی و خروجی با توجه به منابع موجود بر روی برد مربوط می‌شد. برای عبور از این چالش‌ها به منظور اجرای برنامه بر روی برد از یک ماتریس تصویر ورودی قرار گرفته در یک فایل سرآیند[۱] برای دادن ورودی به ماژول استفاده کردیم. خروجی‌های ماژول را در طرح پیاده شده در ویوادو به هسته‌های اشکال‌زدایی[۲] که در واقع کاوشگر[۳] امواج عبوری از یک نقطه‌ی مدار هستند، متصل کردیم تا بتوانیم سیگنال‌های خروجی از ماژول را ببینیم. همچنین برای اینکه بتوانیم از ترفندهای بهینه‌سازی زمان اجرای کار استفاده کنیم (که این بهینه‌سازی‌ها به صورت طبیعی منجر به افزایش مساحت ماژول می‌شوند) و همچنین برای اینکه بتوانیم خروجی‌های گرفته شده در هسته‌ی اشکال‌زدایی را ساده‌تر ببینیم و بخوانیم، از تصاویر کوچک با اندازه‌ی ورودی ۱۰×۱۰ و خروجی ۵×۵ استفاده کردیم.

برای سنجش عملکرد بر روی برد، یک پروژه در نرم‌افزار ویوادو ساختیم و یک طرح بلوکی برای آن ایجاد کردیم. در این طرح بلوکی، هسته‌ی مالکیت معنوی ساخته شده توسط ابزار HLS را در کنار یک

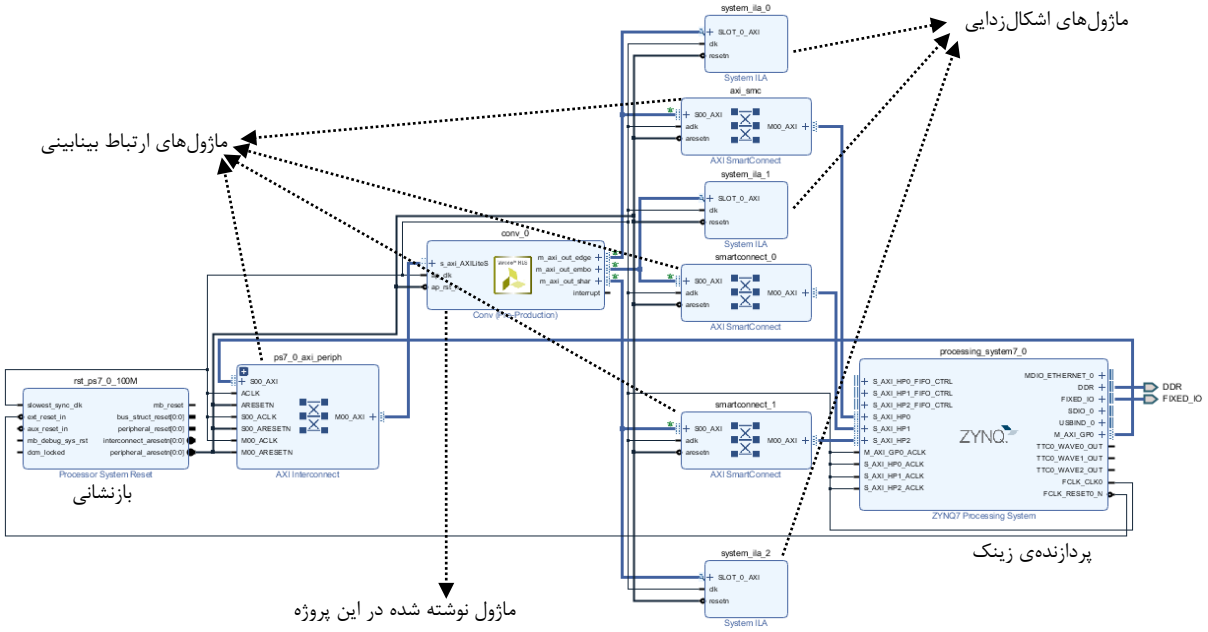
---

۱ Header File

۲ Debugging

۳ Probe

پردازنده‌ی نرم Zynq و چند ماژول ارتباط بینابینی، ماژول بازنشانی و ماژول اشکال‌زدایی قرار دادیم. شکل این طرح بلوکی را در شکل ۲۶ می‌بینید.



شکل ۲۶- تصویر طرح بلوکی فضای آزمون IP بر روی برد

برای راه‌اندازی هسته‌ی مالکیت معنوی و دادن کلاک به آن از پردازنده‌ی Zynq استفاده کرده‌ایم. در طراحی‌های پیچیده‌تر و به طور مثال به منظور ساخت یک شبکه‌ی عصبی کامل می‌توان از این پردازنده برای مدیریت و کنترل کل طرح استفاده کرد و سامانه را به یک هم‌طراحی[۱] پیچیده‌ی نرم‌افزار و سخت‌افزار تبدیل کرد. نیاز به ماژول بازنشانی نیز واضح است. به‌کارگیری ماژول‌های ارتباط بینابینی نیز برای ارتباط دادن درگاه‌های ارباب در خروجی ماژول سخت‌افزاری به درگاه‌های برده در پردازنده هستند. دقت شود که عرض داده برای درگاه‌های برده با کارایی بالا در پردازنده‌ی Zynq به صورت پیش‌فرض ۶۴ بیتی هستند که آن‌ها را به ۳۲ تغییر دادیم تا با خروجی ماژول سخت‌افزاری هم‌سان باشند. همچنین ماژول UART را در پردازنده فعال کردیم تا با دیدن یک علامت در این درگاه، از راه‌اندازی ماژول سخت‌افزاری مطمئن شویم.

[۱] Co-Design

پس از قرار دادن عناصر طراحی در کنار هم با استفاده از ابزار صحت‌سنجی طرح[1] در نرم‌افزار ویوادو مطمئن شدیم که اتصالات به درستی برقرار هستند، فضاهای آدرس‌دهی برای درگاه‌های مختلف هم‌پوشانی ندارند و به طور کلی طرح عیبی ندارد. سپس یک بسته‌بندی خودکار زبان توصیف سخت‌افزار[2] ساختیم تا پین[3]های طرح را بر روی پین‌های برد نگاشت کند. سپس طرح را سنتز و پیاده‌سازی کرده و در نهایت جریان بیت را تولید کرده و سخت‌افزار تولید شده را استخراج کردیم.

در این مرحله، بسته‌ی توسعه نرم‌افزار زایلینکس را باز کردیم و یک پروژه در آن ساختیم تا بتوانیم برای پردازنده کد بنویسیم و از این طریق ماژول سخت‌افزاری را راه‌اندازی کنیم.

در ویوادو ابزار مدیریت سخت‌افزار را باز کرده و برد را از طریق کابل USB به سیستم توسعه متصل کردیم. برد را برنامه‌ریزی کرده و پایانه‌ی بسته‌ی توسعه‌ی نرم‌افزاری زایلینکس را به درگاه USB-ی که برد به آن متصل است وصل کردیم و سپس برنامه‌ی راه‌اندازی را بر روی سخت افزار اجرا کردیم. به این شکل طرح به راه افتاده و ماژول‌های اشکال‌زدایی شروع به کار می‌کنند. در این زمان می‌توانیم با حساس کردن این ماژول‌ها به تغییرات مورد نظر ما در سیگنال‌های کنترلی خروجی ماژول مثل WVALID که صحیح بودن مقدار نوشته شده توسط یک درگاه خروجی خاص را نشان می‌دهد، داده‌های خروجی از برد را ببینیم.

۴-۳-۲- اجرا بر روی برد

همان‌طور که شرح دادیم برنامه را بر روی برد برده و برای چند ماتریس ورودی اجرا کردیم که خروجی‌های متناسب با آن‌ها حاصل شدند و در ادامه آن‌ها را می‌بینیم.

---

[1] Validate Design

[2] HDL Wrapper

[3] Pin

۴-۳-۲-۱- نمونه‌ی اول

ورودی اول در شکل ۲۷ آورده شده است. همچنین خروجی‌های مورد انتظار حاصل از اجرای کد نرم‌افزاری اشاره شده در بخش ۴-۲-۲ را برای این ورودی در شکل ۲۸ می‌بینیم.

$$\begin{bmatrix} 205 & 166 & 118 & 88 & 74 & 81 & 89 & 106 & 157 & 215 \\ 169 & 117 & 94 & 86 & 79 & 75 & 63 & 90 & 113 & 177 \\ 139 & 105 & 95 & 75 & 58 & 73 & 79 & 80 & 98 & 142 \\ 118 & 100 & 95 & 78 & 67 & 78 & 80 & 79 & 82 & 115 \\ 96 & 93 & 80 & 80 & 74 & 75 & 58 & 84 & 96 & 90 \\ 88 & 71 & 51 & 58 & 76 & 80 & 92 & 105 & 91 & 87 \\ 106 & 89 & 73 & 71 & 97 & 86 & 97 & 92 & 82 & 102 \\ 150 & 93 & 105 & 72 & 65 & 56 & 57 & 64 & 65 & 158 \\ 215 & 133 & 77 & 82 & 65 & 54 & 56 & 54 & 125 & 236 \\ 232 & 237 & 176 & 112 & 68 & 56 & 91 & 170 & 249 & 252 \end{bmatrix}$$

شکل ۲۷- ماتریس تصویر ورودی اول به برد

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 39 & 26 & 0 \\ 12 & 0 & 84 & 77 & 16 \\ 255 & 144 & 0 & 0 & 255 \\ 255 & 255 & 46 & 255 & 255 \end{bmatrix}$$

(الف) خروجی مورد انتظار از فیلتر تشخیص لبه

$$\begin{bmatrix} 205 & 14 & 64 & 255 & 255 \\ 54 & 13 & 99 & 126 & 179 \\ 90 & 108 & 153 & 170 & 127 \\ 255 & 137 & 97 & 110 & 255 \\ 255 & 135 & 97 & 255 & 255 \end{bmatrix}$$

(ب) خروجی مورد انتظار از فیلتر منبت‌کاری

$$\begin{bmatrix} 255 & 148 & 102 & 132 & 255 \\ 198 & 122 & 95 & 106 & 222 \\ 118 & 110 & 85 & 186 & 133 \\ 243 & 210 & 187 & 178 & 255 \\ 255 & 255 & 56 & 255 & 255 \end{bmatrix}$$

(ج) خروجی مورد انتظار از فیلتر شفافیت

شکل ۲۸- خروجی‌های مورد انتظار با ورودی شکل ۲۷

خروجی‌های دیده شده از ماژول‌های اشکال‌زدایی یا همان امواج میله‌های کاوشگر حاصل از اجرای ماژول بر روی برد برای این ورودی به صورت زیر بوده است:

مقادیر قرار گرفته بر روی درگاه تشخیص لبه:

00000000, 00, 000b27, 1a00, 0c00, 544d10, ff, 900000ff, ffff2eff, ff

حاصل تبدیل این مقادیر به مبنای ده:

0,0,0,0,0,0,11,39,26,0,12,0,84,77,16,255,144,0,0,255,255,255,46,255,255

مقادیر قرار گرفته بر روی درگاه منبت‌کاری:

cd0e40ff, ff, 360d63, 7eb3, 5a6c, 99aa7f, ff, 89616eff, ff8761ff, ff

حاصل تبدیل این مقادیر به مبنای ده:

205,14,64,255,255,54,13,99,126,179,90,108,153,170,127,255,137,97,110,255,255,135,97,255,255

مقادیر قرار گرفته بر روی درگاه شفافیت:

ff946684, ff, c67a5f, 6ade, 766e, 55ba85, f3, d2bbb2ff, ffff38ff, ff

حاصل تبدیل این مقادیر به مبنای ده:

255,148,102,132,255,198,122,95,106,222,118,110,85,186,133,243,210,187,178,255,255,255,56,255,255

همان‌طور که مشاهده می‌کنیم مقادیر دقیقاً با آرایه‌ی یک‌بعدی شده‌ی ماتریس‌های خروجی مورد انتظار یکسان هستند و نتیجه می‌گیریم که ماژول به درستی بر روی برد کار کرده است.

۴-۳-۲-۲- نمونه‌ی دوم

ورودی دوم در شکل ۲۹ آورده شده است. همچنین خروجی‌های مورد انتظار حاصل از اجرای کد نرم‌افزاری اشاره شده در بخش ۴-۲-۲ را برای این ورودی در شکل ۳۰ می‌بینیم.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 6 & 136 & 203 & 188 & 28 & 0 & 0 & 0 \\ 0 & 0 & 55 & 176 & 34 & 200 & 73 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 67 & 222 & 18 & 0 & 0 & 0 \\ 0 & 0 & 0 & 15 & 217 & 77 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 142 & 176 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 33 & 240 & 41 & 0 & 0 & 15 & 45 & 15 \\ 0 & 0 & 31 & 223 & 204 & 214 & 175 & 189 & 176 & 65 \\ 0 & 0 & 0 & 18 & 38 & 63 & 43 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

شکل ۲۹- ماتریس تصویر ورودی دوم به برد

$$\begin{bmatrix} 55 & 255 & 255 & 255 & 0 \\ 0 & 0 & 82 & 14 & 0 \\ 33 & 255 & 138 & 75 & 120 \\ 31 & 221 & 255 & 255 & 255 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(الف) خروجی مورد انتظار از فیلتر تشخیص لبه

$$\begin{bmatrix} 116 & 255 & 255 & 0 & 0 \\ 61 & 228 & 255 & 0 & 0 \\ 66 & 255 & 255 & 105 & 105 \\ 95 & 255 & 252 & 255 & 255 \\ 0 & 76 & 90 & 0 & 0 \end{bmatrix}$$

(ب) خروجی مورد انتظار از فیلتر منبت‌کاری

$$\begin{bmatrix} 0 & 255 & 255 & 0 & 0 \\ 0 & 255 & 255 & 119 & 0 \\ 0 & 255 & 255 & 0 & 0 \\ 0 & 255 & 255 & 255 & 255 \\ 0 & 0 & 20 & 0 & 0 \end{bmatrix}$$

(ج) خروجی مورد انتظار از فیلتر شفافیت

شکل ۳۰- خروجی‌های مورد انتظار با ورودی شکل ۲۹

خروجی‌های دیده شده از ماژول‌های اشکال‌زدایی یا همان امواج میله‌های کاوشگر حاصل از اجرای ماژول بر روی برد برای این ورودی به صورت زیر بوده است:

مقادیر قرار گرفته بر روی درگاه تشخیص لبه:

37ffffff, 00, 000052, 0e00, 21ff, 8a4b78, 1f, ddffffff, 00000000, 00

حاصل تبدیل این مقادیر به مبنای ده:

55,255,255,255,0,0,0,82,14,0,33,255,138,75,120,31,221,255,255,255,0,0,0,0,0

مقادیر قرار گرفته بر روی درگاه منبت‌کاری:

74ffff00, 00, 3de4ff, 0000, 42ff, ff6969, 5f, fffcffff, 004c5a00, 00

حاصل تبدیل این مقادیر به مبنای ده:

116,255,255,0,0,61,228,255,0,0,66,255,255,105,105,95,255,252,255,255,0,76,90,0,0

مقادیر قرار گرفته بر روی درگاه شفافیت:

00ffff00, 00, 00ffff, 7700, 00ff, ff0000, 00, ffffffff, 00001400, 00

حاصل تبدیل این مقادیر به مبنای ده:

0,255,255,0,0,0,255,255,119,0,0,255,255,0,0,0,255,255,255,255,0,0,20,0,0

همان‌طور که مشاهده می‌کنیم مقادیر دقیقاً با آرایه‌ی یک‌بعدی شده‌ی ماتریس‌های خروجی مورد انتظار یکسان هستند و نتیجه می‌گیریم که ماژول به درستی بر روی برد کار کرده است.

۴-۴- جمع‌بندی

در این فصل توانستیم به کمک مطالبی که در فصل‌های گذشته بیان کردیم، یک ماژول سخت-افزاری پیاده‌سازی کنیم تا محاسبه‌ی کانولوشن و انجام عمل ادغام را بر روی FPGA انجام دهیم. روند اشاره شده در بخش ۴-۳-۲ را برای ماتریس‌های ورودی با اندازه‌ی ۹۲×۱۶۴ و خروجی با اندازه‌ی ۴۶×۸۲ و همین‌طور برای ماتریس‌های ورودی با اندازه‌ی ۲۸×۲۸ و خروجی با اندازه‌ی ۱۴×۱۴ نیز تکرار کردیم که در آن موارد نیز سخت‌افزار به درستی عمل کرد. در این پیاده‌سازی از دو رهنمود بهینه‌سازی جریان داده و خط لوله‌سازی حلقه‌ها نیز استفاده کردیم که موجب کاهش تاخیر شدند. در فصل آینده به تفصیل به نتایج حاصل و تاثیر این بهینه‌سازی‌ها خواهیم پرداخت.

فصل پنجم

جمع‌بندی و کارهای آینده

جمع‌بندی و کارهای آینده

۵-۱- جمع‌بندی

در این پروژه، هدف پیاده‌سازی تابع کانولوشن و تابع کاهش بعد بر روی FPGA به کمک ابزار سنتز سطح بالای ویوادو بوده است. این دو تابع از توابع اصلی و حیاتی شبکه‌های عصبی کانولوشنی هستند و پیاده‌سازی بخش‌های پرهزینه از یک سامانه‌ی نرم‌افزاری بر روی FPGA، هم‌چون کاری که در این پروژه انجام شد، که شتاب‌دهی خوانده می‌شود از مباحث روز علوم کامپیوتر است. در این پروژه توانستیم دو تابع کانولوشن و کاهش بعد را با پیاده سازی سخت‌افزاری شتاب‌دهی کنیم. برای این پیاده-سازی از ابزار قدرتمند سنتز سطح بالای ویوادو استفاده کردیم که نسبت به پیاده‌سازی با زبان انتقال ثبات بسیار سریع‌تر و کم‌هزینه‌تر است.

همان‌طور که در بخش‌های ۴-۲ و ۴-۳ مشاهده کردیم، ماژول سخت‌افزاری پیاده‌سازی شده به روش‌های مختلف شبیه‌سازی و اجرا بر روی برد مورد آزمایش قرار گرفت و دیدیم که به درستی عمل کرده و خروجی‌های مورد انتظار را تولید می‌کند.

۵-۱-۱- تحلیل کارایی و مشاهده‌ی تاثیر روش‌های بهینه‌سازی

در این بخش می‌خواهیم به طور خلاصه به تحلیل کارایی مالکیت معنوی طراحی شده بپردازیم. برای این منظور روی تصاویر با اندازه‌ی ورودی ۱۶۴×۹۲ و خروجی ۸۲×۴۶ کار می‌کنیم.

طرح اولیه بدون اعمال هرگونه بهینه‌سازی غیر خودکار (آنچه که طبق رهنمودهای نویسنده به ابزار دیکته شود) دارای میانگین تاخیر ۲۰۳۳۳۶ چرخه‌ی کلاک برای آماده‌سازی خروجی‌هاست. نتیجه-ی گزارش زمان و کارایی سنتز به صورت زیر است:

Summary

Latency		Interval		
min	max	min	max	Type
203336	203336	203337	203337	none

جدول ۲- تاخیر طراحی پیش از بهینه‌سازی

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	633
FIFO	0	-	40	160
Instance	0	12	3040	3668
Memory	24	-	10	3
Multiplexer	-	-	-	239
Register	-	-	305	-
Total	24	12	3395	4703
Available	120	80	35200	17600
Utilization (%)	20	15	9	26

جدول ۳- میزان استفاده از منابع برد پیش از بهینه‌سازی

برنامه‌ی اشاره شده در بخش ۴-۲-۲ را به گونه‌ای تغییر می‌دهیم تا زمان آماده شدن خروجی‌ها بر روی سامانه‌ی توسعه (که یک رایانه‌ی HP با پردازنده Core i7 و ۸ گیگابایت حافظه‌ی رم است) را محاسبه کند. میانگین زمان اجرا بر روی همان تصویر ورودی ۱۶۴×۹۲ که به ماژول سخت‌افزاری داده-ایم با ۱۰۰ مرتبه اجرا برابر ۷۱۰۳۱۴۰ نانوثانیه است. از طرفی طول دوره‌ی چرخه‌ی ساعت پیش‌بینی شده برای سخت‌افزار طراحی شده ۱۰ نانوثانیه است که با توجه به تعداد کلاک پیش‌بینی شده در سنتز، زمان اجرای برنامه‌ی سخت‌افزاری بدون هیچ‌گونه بهینه‌سازی ۲۰۳۳۳۷۰ نانوثانیه پیش‌بینی می‌شود که حدود ۵ میلیون نانوثانیه از برنامه‌ی نرم‌افزاری سریع‌تر است. البته نکته‌ی دیگری که قابل توجه است این است که ممکن است بتوان کلاک طرح را کمتر از ۱۰ نانوثانیه قرار داد که در آن صورت این اختلاف بیشتر نیز می‌شود. از طرف دیگر معمولا در کاربردهای پردازش تصویر با افزایش اندازه‌ی ورودی عملکرد پردازنده نسبت به روش‌های موازی مثل FPGA و محاسبات گرافیکی بدتر خواهد شد.

در پیاده‌سازی این ماژول سخت‌افزاری، جهت بهینه‌سازی تاخیر و کارایی طرح، طبق مطالبی که در بخش ۲-۲-۳ به آن‌ها اشاره شد، از دو رهنمود DATAFLOW و PIPELINE استفاده شد.

رهنمود DATAFLOW به کل تابع اصلی برنامه اعمال می‌شود و باعث می‌شود تا خط لوله-سازی در سطح کارها فعال شود و حلقه‌ها یا توابع مختلف موجود در طراحی به صورت هم‌روند کار کنند. برای مثال در طرح ما عملیات مربوط به محاسبه‌ی کانولوشن با فیلتر دوم ربطی به عملیات مربوط به محاسبه‌ی کانولوشن با فیلتر اول ندارد؛ در نتیجه این دو عملیات می‌توانند به صورت هم‌روند و با هم اجرا شوند. در مورد فیلتر سوم نیز همین شرایط برقرار است.

از طرفی عملیات مربوط به تابع کاهش بعد به نحوی نوشته شده‌اند که محاسباتی که بـه تولیـد هر یک از پیکسل‌های تصویر خروجی می‌انجامند هیچ‌گونه وابستگی به عملیات مربوط به سایر پیکسل‌ها ندارند و در نتیجه می‌توان تمام پیکسل‌های خروجی را به صورت موازی محاسبه کـرد و در واقـع نیـازی اجرای به ترتیب حلقه وجود ندارد. به همین دلیـل، روی هـر دو حلقـه‌ی تـابع کـاهش بعـد از رهنمـود PIPELINE استفاده شد تا اجرای تکرارهای مختلف حلقه‌ها هم‌روند شود.

اضافه کردن این رهنمودها خود به توجه و اعمال برخی تغییرات نیاز داشـت. مـثلاً بـرای اینکـه اجرای هم‌روند کارها برای سه فیلتر مورد نظر ممکن شود، به سه نسخه‌ی یکسان از آرایه‌ی ورودی نیـاز داشتیم. این موضوع خود باعث سه برابر شدن متغیرهای درون ماژول می‌شد که در نتیجه به تعـداد سـه برابر حافظه‌ی BRAM بر روی برد نیاز داشتند و در نتیجه طرح برای اندازه‌های بزرگ‌تـر بـر روی بـرد زیبو جا نمی‌شد. تبدیل کردن متغیرهای میانی به نوع ایستا این مشکل را کمتر می‌کند زیرا باعـث مـی-شود یک بار و برای کل طول عمر برنامه تعریف شوند؛ در نتیجه چند نمونه از هر یـک از آن‌هـا سـاخته نشود. از آنجا که تابع تبدیل جریان آرایه‌ی ورودی به داده‌ساختار Mat داده‌ی ورودی را مصرف می‌کند، نمی‌توان سه بار روی یک جریان ورودی آن را فراخوانی کرد. در نتیجه بایـد بـه روش دیگـری از ورودی رونویسی می‌کردیم که به کمک تابع Duplicate ممکن شد. ورودی این تابع که پـیش از ایـن، دسـتور نوشتن بر روی آن از منبع جریان ورودی را داده‌ایم حال باید توسط تابع Duplicate خوانده شود. یعنی نیاز داشته‌ایم تا هم در این متغیر (آرایه) بنویسیم و هم از آن بخوانیم؛ چنـین امـری روی یـک آرایـه‌ی ایستا در ابزار HLS ممکن نیست. در نتیجه بار دیگر مجبور شدیم تا آرایه‌هایی که به این تابع به عنـوان ورودی می‌دهیم را از حالت ایستا خارج کنیم. از طرفی به‌کارگیری رهنمود خط لوله‌سازی حلقه‌ها باعـث می‌شود تا تعداد جدول‌های جست‌وجو مورد نیاز بیشتر شود. برای مثال همان‌طور که در جـدول ۵ مـی-بینید، استفاده از این رهنمود در طرح با اندازه‌ی فعلی موجب می‌شود که ماژول ما بر روی برد زیبـو جـا نشود.

Summary

Latency		Interval		
min	max	min	max	Type
23152	23152	16602	16895	dataflow

جدول ۴- تاخیر طرح پس از بهینه‌سازی

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	76
FIFO	0	-	100	379
Instance	8	1	6503	24603
Memory	24	-	0	0
Multiplexer	-	-	-	90
Register	-	-	15	-
Total	32	1	6618	25148
Available	120	80	35200	17600
Utilization (%)	26	1	18	142

جدول ۵- میزان استفاده از منابع برد پس از بهینه‌سازی

با مشاهده‌ی جدول ۴ و مقایسه‌ی آن با جدول ۲ می‌بینیم که تنها با اعمال دو نمونه بهینه‌سازی بر روی طرح اولیه توانستیم زمان تاخیر آماده‌سازی خروجی را از ۲۰۳۳۳۶ چرخه‌ی ساعت به تنها ۲۳۱۵۲ چرخه‌ی ساعت کاهش دهیم. این بدین معناست که با این بهینه‌سازی ۸٫۷ برابر سرعت را نسبت به ماژول سخت‌افزاری اولیه و ۳۰ برابر نسبت به برنامه‌ی نرم‌افزاری نوشته شده بر روی سامانه‌ی توسعه افزایش داده‌ایم. باید توجه شود که پیاده‌سازی اولیه بر روی سخت‌افزار نسبت به نرم‌افزار تنها حدود ۳ برابر سریع‌تر بوده است. ممکن است با تلاش برای بهینه‌سازی خود برنامه‌ی نرم‌افزاری مثلا با اعمال تغییر روی آن برای اجرا بر روی رشته‌های متفاوت روی پردازنده‌ی مرکزی نیز بتوانیم به چنین تسریعی برسیم. از این رو ما در این تحقیق به این تسریع بسنده نکرده و تلاش کردیم تا ماژول سخت‌افزاری طراحی شده با اختلاف قابل توجهی نسبت به نرم‌افزار سریع‌تر باشد.

لازم به ذکر است از آنجا که اندازه‌ی طرحی که در بخش ۴-۳-۲ بر روی برد اجرا کردیم بسیار کوچک‌تر بود (ورودی ۱۰×۱۰)، مشکل کمبود جداول جست‌وجو پس از اعمال بهینه‌سازی که در جدول ۵ می‌بینیم در آن طرح وجود نداشت.

در انتها دیدیم که توانستیم به کمک پیاده‌سازی دو عمل کانولوشن و کاهش بعد بر روی FPGA به شتاب‌دهی این عملیات بپردازیم. از این پروژه می‌توان به عنوان هسته‌ی محاسبات کانولوشن و کاهش بعد در یک شبکه‌ی عصبی کانولوشنی کامل استفاده کرد و به سرعت بالاتری در اجرای این شبکه‌ها دست یافت.

۵-۲-کارهای آینده

شتاب‌دهی سخت‌افزاری در کاربردهای هوش مصنوعی و شبکه‌های عمیق و به خصوص با استفاده از ابزارهای سنتز سطح بالا از تحقیقات به‌روز و لبه‌ی علم در زمینه‌های هوش مصنوعی و سخت-افزار است که جای خالی کارهای بسیار پیچیده‌تر و حرفه‌ای‌تر از آنچه در این پروژه‌ی کارشناسی انجام شد، در سطح پروژه‌های کارشناسی ارشد و دکتری در سطح دانشکده خالی بود. به‌روز بودن این تکنولوژی و دشوارتر بودن طراحی‌های سخت‌افزاری علت اصلی نبود منابع ساده‌ی آموزشی برای انجام کارهای مشابه است. نحوه‌ی استفاده از ابزار ویوادو و هسته‌های مالکیت معنوی از پیش نوشته‌شده، تحلیل نتایج و رفع مشکلات، استفاده‌ی درست از روش‌های بهینه‌سازی، چگونگی کار با انواع توابعی که ابزار پس از سنتز برای استفاده در مجموعه‌ی توسعه‌ی نرم‌افزار تولید می‌کند، تخصیص آدرس‌های سخت‌افزاری به ماژول‌های مختلف موجود در یک طرح و کار کردن با این آدرس‌ها و موارد دیگر همگی چالش‌هایی هستند که در این نوع پروژه‌ها با آن‌ها درگیر می‌شویم.

استفاده از طراحی‌های توأم سخت‌افزار و پردازنده در کنارهم می‌تواند به شتاب‌دهی بسیار خوب محاسبات در شبکه‌های عصبی و کاربردهای پردازش تصویر بیانجامد و این امر در کارهای انجام شده در منابع [۲۳] و [۲۴] مشهود است. در نتیجه تلاش برای عبور از چالش‌های اشاره شده می‌تواند به تولید محصولاتی بسیار ارزشمند بیانجامد. محصولاتی که نسبت به روش‌های دیگر توان مصرفی کمتری دارند، گذردهی بیشتری دارند و حجم آن‌ها کوچک است و می‌توانند در سامانه‌های نهفته مورد استفاده قرار گیرند.

کار بر روی پروژه‌ی ارائه شده در همین گزارش را می‌توان ادامه داد. استفاده از دیگر روش‌های بهینه‌سازی که در بخش ۳-۲-۲ برخی از آن‌ها را بررسی کردیم اولین قدم در این مسیر است. استفاده از این ماژول در یک طراحی توأم در کنار یک پردازنده و کنترل و رد و بدل کردن داده بین پردازنده و ماژول از کارهایی است که به عنوان یکی از با اهمیت‌ترین کارهای آینده به شدت توصیه می‌شود؛ زیرا پیش‌نیازیست که رفع آن موجب باز شدن مسیر برای ایجاد طراحی‌های جامع و کامل شبکه‌های عصبی و به کلی شتاب‌دهی محاسبات بر روی FPGA می‌شود. یکی از روش‌های مطرح برای انجام این کار استفاده از هسته‌های مالکیت معنوی دسترسی مستقیم به حافظه است. فراهم کردن امکان خواندن

داده‌ها از یک حافظه‌ی خارجی مانند حافظه‌ی SD موجود روی برد زیبو و نوشتن خروجی بر روی آن، گام بعدی در راستای تولید یک سامانه‌ی کامل و یک محصول نهایی‌ست.

پس از طی کردن این مسیر می‌توان به سایر چالش‌های موجود در شتاب‌دهی سخت‌افزاری با FPGA پرداخت و علاوه بر بهینه‌سازی معماری طرح بر روی سخت‌افزار، طراحی و معماری خود شبکه-ی عصبی و داده‌ساختارهای موجود روی آن را تغییر داد و بهینه کرد. مشکلات مربوط به بازنمایی وزن-های شبکه‌های عصبی به گونه‌ای که به فضای ذخیره‌سازی کمتری نیاز داشته باشد و بر روی منابع محدود یک تراشه‌ی FPGA قرار بگیرد از این دست چالش‌هاست.

از دیگر نکات قابل توجه این است که برای شتاب‌دهی یک پروژه ابتدا باید قسمتی از برنامه که بیشترین پیچیدگی محاسباتی را دارد یا فراوانی فراخوانی آن بسیار زیاد است و بیشتر زمان اجرا را به خود اختصاص می‌دهد، پیدا کرد. ما در این پروژه به سخنان تحقیقات گذشته اکتفا کرده و به تسریع قسمت کانولوشن پرداختیم؛ اما در یک طرح بزرگ بهتر است تا ابتدا برنامه‌ی نرم‌افزاری با ابزارهای پروفایلینگ مورد تحلیل قرار گیرد و سپس نسبت به انتخاب بخش‌هایی که به شتاب‌دهی نیاز دارند، اقدام شود.

منابع و مراجع

[1] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, Huazhong Yan, "[DL] A Survey of FPGA-Based Neural Network Inference Accelerator," *Tsinghua University*, 2018.

[2] Ole Martin Skafså, "FPGA implementation of a Convolutional Neural Network for 'Wake up word' detection," *Norwegian University of Science and Technology*, 2017.

[3] Riccardo Albertazzi, "Implementation of a Binary CNN on FPGA with High-Level Synthesis Tools," *University of Bologna*, 2018.

[4] Jin Hee Kim, Brett Grady, Ruolong Lian, John Brothersy, Jason H. Anderson, "FPGA-Based CNN Inference Accelerator Synthesized from Multi-Threaded C Software," *University of Toronto*, 2018.

[5] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM Journal of Research and Development*, pp. 210–229, 1959.

[6] T. M. Mitchell, "Machine Learning," *McGraw-Hill Science/Engineering/Math*, 1997.

[7] C. M. Bishop, "Pattern Recognition and Machine Learning," *Springer*, 2006.

[8] M. Nielsen, "Neural Networks and Deep Learning," [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html. [Accessed May, 2019].

[9] Stanford, "CS231n Convolutional Neural Networks for Visual Recognition," [Online]. Available: http://cs231n.github.io. [Accessed May, 2019].

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, 86(11), pp. 2278–2324, 1998.

[11] Xilinx, "Introduction to FPGA Design with Vivado High-Level Synthesis UG998," 2019.

[12] Xilinx, “Vivado Design Suite User Guide High-Level Synthesis UG902,” 2015.

[13] DIGILENT, “Zybo Reference Manual,” [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual. [Accessed May, 2019].

[14] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le, “Mnasnet: Platform-aware neural architecture search for mobile,” *arXiv preprint arXiv:1807.11626 (2018)*, 2018.

[15] Xin Wang, Fisher Yu, Zi-Yi Dou, and Joseph E Gonzalez, “Skipnet: Learning dynamic routing in convolutional networks,” *arXiv preprint arXiv:1711.09485 (2017)*, 2017.

[16] Junsong Wang, Qiuwen Lou, Xiaofan Zhang, Chao Zhu, Yonghua Lin, and Deming Chen, “Design Flow of Accelerating Hybrid Extremely Low Bit-width Neural Network in Embedded FPGA,” *arXiv preprint arXiv:1808.04311 (2018)*, 2018.

[17] Fengfu Li, Bo Zhang, and Bin Liu, “Ternary weight networks,” *arXiv preprint arXiv:1605.04711 (2016)*, 2016.

[18] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou, “DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients,” *arXiv preprint arXiv:1606.06160 (2016)*, 2016.

[19] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. “Going deeper with embedded fpga platform for convolutional neural network”, In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, 2016.

[20] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun, “Efficient and accurate approximations of nonlinear convolutional networks,” In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1984–1992, 2015.

[21] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky, “Sparse convolutional neural networks,” In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp.806–814, 2015.

[22] Song Han, Huizi Mao, and William J Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149 (2015)*, 2015.

[23] D. Wang, J. An, and K. Xu, “PipeCNN: An OpenCL-Based FPGA Accelerator for Large-Scale Convolution Neuron Networks,” *CoRR*, abs/1611.02450, 2016.

[24] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, “FINN: A framework for fast, scalable binarized neural network inference,” *CoRR*, abs/1612.07119, 2016.

[25] Xilinx, “Vivado High-Level Synthesis,” [Online]. Available: https://www.xilinx.com/products/designtools/vivado/integration/esl-design.html. [Accessed May, 2019].

[26] G. Venkatesh, E. Nurvitadhi, and D. Marr, “Accelerating deep convolutional networks using low-precision and sparsity,” *arXiv.org*, 2016. [Online]. Available: http://arxiv.org/abs/1610.00324. [Accessed May, 2019].

پیوست‌ها

پیوست ۱- کدهای نوشته شده

۱- تابع اصلی ماژول (Top-Function)

```
void conv(uint8_t image_in[IN_W*IN_H], uint8_t out_edge[OUT_W*OUT_H], uint8_t out_embo[OUT_W*OUT_H], uint8_t out_shar[OUT_W*OUT_H]){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi depth=784 port=image_in
#pragma HLS INTERFACE m_axi depth=196 port=out_shar
#pragma HLS INTERFACE m_axi depth=196 port=out_embo
#pragma HLS INTERFACE m_axi depth=196 port=out_edge
#pragma HLS DATAFLOW

    hls::Mat<IN_H,IN_W,HLS_8UC1> src_tmp1;
    hls::Mat<IN_H,IN_W,HLS_8UC1> src_tmp2;
    static hls::Mat<IN_H,IN_W,HLS_8UC1> src1;
    static hls::Mat<IN_H,IN_W,HLS_8UC1> src2;
    static hls::Mat<IN_H,IN_W,HLS_8UC1> src3;
    static hls::Mat<IN_H,IN_W,HLS_8UC1> dst1;
    static hls::Mat<IN_H,IN_W,HLS_8UC1> dst2;
    static hls::Mat<IN_H,IN_W,HLS_8UC1> dst3;

    hls::Window<3,3,char> kernel1;
    hls::Window<3,3,char> kernel2;
    hls::Window<3,3,char> kernel3;
    hls::Point_<int> anchor = hls::Point_<int>(-1,-1);

    hls::AXIM2Mat<IN_W,uint8_t,IN_H,IN_W,HLS_8UC1>(image_in,src_tmp1);
    hls::Duplicate(src_tmp1,src_tmp2,src1);
    hls::Duplicate(src_tmp2,src2,src3);


    //filter1: edge detector
    const char coefficients1[3][3] = { {-1,-2,-1},{ 0, 0, 0},{ 1, 2, 1} };
    for (int i=0;i<3;i++){
       for (int j=0;j<3;j++){
          kernel1.val[i][j]=coefficients1[i][j];
       }
    }
    hls::Filter2D(src1,dst1,kernel1,anchor);
    uint8_t dst1_arr[IN_H*IN_W];
    hls::Mat2AXIM<IN_W,uint8_t,IN_H,IN_W,HLS_8UC1>(dst1, dst1_arr);
    maxpooling(dst1_arr, out_edge);
    //end of filter1: edge detector


    //filter2: emboss
    const char coefficients2[3][3] = { {-2, -1, 0},{-1, 1, 1},{0, 1, 2} };
    for (int i=0;i<3;i++){
        for (int j=0;j<3;j++){
            kernel2.val[i][j]=coefficients2[i][j];
        }
    }

    hls::Filter2D(src2,dst2,kernel2,anchor);
```

```
    uint8_t dst2_arr[IN_H*IN_W];
    hls::Mat2AXIM<IN_W,uint8_t,IN_H,IN_W,HLS_8UC1>(dst2, dst2_arr);
    maxpooling(dst2_arr, out_embo);
    //end of filter2: emboss


    //filter3: sharpen
    const char coefficients3[3][3] = { {0, -1, 0},{-1, 5, -1},{0, -1, 0} };
    for (int i=0;i<3;i++){
        for (int j=0;j<3;j++){
            kernel3.val[i][j]=coefficients3[i][j];
        }
    }
    hls::Filter2D(src3,dst3,kernel3,anchor);
    uint8_t dst3_arr[IN_H*IN_W];
    hls::Mat2AXIM<IN_W,uint8_t,IN_H,IN_W,HLS_8UC1>(dst3, dst3_arr);
    maxpooling(dst3_arr, out_shar);
    //end of filter3: sharpen

}
```

۲- تابع کاهش بعد

```
void maxpooling(uint8_t input_mat[IN_H*IN_W], uint8_t output_mat[OUT_H*OUT_W]){
    int i,j =0;
    uint8_t a, b, c, d, max = 0;
    for(j=0; j<=(IN_H-2); j=j+2){
#pragma HLS PIPELINE
        for(i=0; i<=(IN_W-2); i=i+2){
#pragma HLS PIPELINE
            a = input_mat[j*IN_W + i];
            b = input_mat[j*IN_W + i+1];
            c = input_mat[(j+1)*IN_W + i];
            d = input_mat[(j+1)*IN_W + i+1];
            max = findMax(a,b,c,d);
            output_mat[(j/2)*OUT_W + i/2] = max;
        }
    }
}
```

۳- تابع محاسبه‌ی بیشینه

```
uint8_t findMax(uint8_t a, uint8_t b, uint8_t c, uint8_t d){
    uint8_t max;
    max = a;
    if(b > max){
        max = b;
    }
    if(c > max){
        max = c;
    }
    if(d > max){
```

```
        max = d;
    }
    return max;
}
```

۴- لیست تعاریف و کتاب‌خانه‌های مورد استفاده در ۳ تابع بالا

```
#include <stdint.h>
#include <hls_video.h>
//#include "imagexvec.h"
using namespace std;

#define IN_W 92
#define IN_H 164
#define OUT_W 46
#define OUT_H 82
```

۵- کد نیمکت آزمون

```
#include <stdint.h>
#include <stdio.h>
#include <hls_opencv.h>
#include <string.h>
using namespace cv;

#define IN_W 28
#define IN_H 28
#define OUT_W 14
#define OUT_H 14

void conv(uint8_t * image_in, uint8_t * out_edge, uint8_t * out_embo, uint8_t * o
ut_shar);

int main(){

    int k=0, t=0;
    static uint8_t image_in[IN_W*IN_H];
    //static Mat im;
    Mat im;
    Mat golden_im_edge, golden_im_embo, golden_im_shar; //code for c/rtl cosimula
tion
    uint8_t giEdgeArr[OUT_W*OUT_H], giEmboArr[OUT_W*OUT_H], giSharArr[OUT_W*OUT_H
]; //code for c/rtl cosimulation
    static uint8_t out_edge[OUT_W*OUT_H];
    static uint8_t out_embo[OUT_W*OUT_H];
    static uint8_t out_shar[OUT_W*OUT_H];
    static Mat out1, out2, out3;

    for (k=0; k<100; k++){
        im = imread(std::to_string(k) + ".png",CV_LOAD_IMAGE_GRAYSCALE);
        memcpy(image_in,im.data,sizeof(uint8_t)*IN_W*IN_H);
        conv(image_in, out_edge, out_embo, out_shar);
```

```
        /*
        out1 = Mat(OUT_H,OUT_W,CV_8UC1,out_edge);
        imwrite("outs/" + std::to_string(k) + "edge.png", out1);
        out2 = Mat(OUT_H,OUT_W,CV_8UC1,out_embo);
        imwrite("outs/" + std::to_string(k) + "embo.png", out2);
        out3 = Mat(OUT_H,OUT_W,CV_8UC1,out_shar);
        imwrite("outs/" + std::to_string(k) + "shar.png", out3);
         */
        // new codes for c/rtl co simulation
        golden_im_edge = imread(std::to_string(k) + "edge.png", CV_LOAD_IMAGE_GRAYSCALE);
        golden_im_embo = imread(std::to_string(k) + "embo.png", CV_LOAD_IMAGE_GRAYSCALE);
        golden_im_shar = imread(std::to_string(k) + "shar.png", CV_LOAD_IMAGE_GRAYSCALE);
        memcpy(giEdgeArr,golden_im_edge.data,sizeof(uint8_t)*OUT_W*OUT_H);
        memcpy(giEmboArr,golden_im_embo.data,sizeof(uint8_t)*OUT_W*OUT_H);
        memcpy(giSharArr,golden_im_shar.data,sizeof(uint8_t)*OUT_W*OUT_H);
        for(t=0; t<25; t++){
            if((giEdgeArr[t] != out_edge[t]) || (giEmboArr[t] != out_embo[t]) || (giSharArr[t] != out_shar[t])){
                return 1;
            }
        }
    }


    return 0;
}
```

۶- کدهای مربوط به پیاده‌سازی نرم‌افزاری به زبان C++ در محیط Visual Studio

```
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui.hpp>
#include <iostream>
#include <string.h>
#include <cstdlib>
//#include "imagexvec.h"
//using namespace cv;
using namespace std;

#define IN_W 10
#define IN_H 10
#define OUT_W 14
#define OUT_H 14

uint8_t findMax(uint8_t a, uint8_t b, uint8_t c, uint8_t d) {
    uint8_t max;
    max = a;
    if (b > max) {
        max = b;
    }
    if (c > max) {
        max = c;
```

```
25.     }
26.     if (d > max) {
27.         max = d;
28.     }
29.     return max;
30. }
31.
32.
33. void maxpooling(uint8_t input_mat[IN_H*IN_W], uint8_t output_mat[OUT_H*OUT_W]) {

34.     int i, j = 0;
35.     uint8_t a, b, c, d, max = 0;
36.     for (j = 0; j <= (IN_H - 2); j = j + 2) {
37.         for (i = 0; i <= (IN_W - 2); i = i + 2) {
38.             a = input_mat[j*IN_W + i];
39.             b = input_mat[j*IN_W + i + 1];
40.             c = input_mat[(j + 1)*IN_W + i];
41.             d = input_mat[(j + 1)*IN_W + i + 1];
42.             max = findMax(a, b, c, d);
43.             output_mat[(j / 2)*OUT_W + i / 2] = max;
44.             //printf("%d ", output_mat[(j / 2)*OUT_W + i / 2]);
45.             //printf("j=%d ", j); printf("i=%d,", i);
46.         }
47.     }
48. }
49.
50.
51. double module(int filename) {
52.     int i = 0;
53.     cv::Mat image;
54.     //image = cv::imread("img/testing/raw/" + std::to_string(filename) + ".png",
    cv::IMREAD_GRAYSCALE); // Read the file
55.     image = cv::imread("image4.jpg", cv::IMREAD_GRAYSCALE);
56.     if (image.empty()) // Check for invalid input
57.     {
58.         cout << "Could not open or find the image" << std::endl;
59.         //return -1;
60.     }
61.     auto start = chrono::steady_clock::now();
62.     uint8_t imageArr[IN_H*IN_W];
63.     memcpy(imageArr, image.data, sizeof(uint8_t) * IN_H * IN_W);
64.
65.     cv::Point anchor = cv::Point(-1, -1);
66.
67.     // edge filter
68.     int8_t kernel1_arr[9] = { -1,-2,-1, 0, 0, 0, 1, 2, 1 };
69.     cv::Mat kernel1 = cv::Mat(3, 3, CV_8SC1);
70.     memcpy(kernel1.data, kernel1_arr, sizeof(int8_t) * 9);
71.     cv::Mat dst1;
72.     cv::filter2D(image, dst1, image.depth(), kernel1, anchor);
73.     uint8_t dst1_arr[IN_H * IN_W];
74.     memcpy(dst1_arr, dst1.data, sizeof(uint8_t) * IN_H * IN_W);
75.     uint8_t out_edge[OUT_H*OUT_W];
76.     maxpooling(dst1_arr, out_edge);
77.     // end of edge filter
78.
79.     // emboss filter
80.     int8_t kernel2_arr[9] = { -2,-1, 0, -1, 1, 1, 0, 1, 2 };
81.     cv::Mat kernel2 = cv::Mat(3, 3, CV_8SC1);
82.     memcpy(kernel2.data, kernel2_arr, sizeof(int8_t) * 9);
83.     cv::Mat dst2;
```

```
84.     cv::filter2D(image, dst2, image.depth(), kernel2, anchor);
85.     uint8_t dst2_arr[IN_H * IN_W];
86.     memcpy(dst2_arr, dst2.data, sizeof(uint8_t) * IN_H * IN_W);
87.     uint8_t out_embo[OUT_H*OUT_W];
88.     maxpooling(dst2_arr, out_embo);
89.     // end of emboss filter
90.
91.     // sharpen filter
92.     int8_t kernel3_arr[9] = { 0,-1, 0, -1, 5, -1, 0, -1, 0 };
93.     cv::Mat kernel3 = cv::Mat(3, 3, CV_8SC1);
94.     memcpy(kernel3.data, kernel3_arr, sizeof(int8_t) * 9);
95.     cv::Mat dst3;
96.     cv::filter2D(image, dst3, image.depth(), kernel3, anchor);
97.     uint8_t dst3_arr[IN_H * IN_W];
98.     memcpy(dst3_arr, dst3.data, sizeof(uint8_t) * IN_H * IN_W);
99.     uint8_t out_shar[OUT_H*OUT_W];
100.            maxpooling(dst3_arr, out_shar);
101.            // end of sharpen filter
102.            /*
103.            cv::Mat hlsEdge = cv::imread("img/testing/hlsres/" + std::to_string(f
    ilename) + "edge.png", cv::IMREAD_GRAYSCALE);
104.            cv::Mat hlsEmbo = cv::imread("img/testing/hlsres/" + std::to_string(f
    ilename) + "embo.png", cv::IMREAD_GRAYSCALE);
105.            cv::Mat hlsShar = cv::imread("img/testing/hlsres/" + std::to_string(f
    ilename) + "shar.png", cv::IMREAD_GRAYSCALE);
106.            uint8_t hlsEdgeArr[OUT_H*OUT_W], hlsEmboArr[OUT_H*OUT_W], hlsSharArr[
    OUT_H*OUT_W];
107.            memcpy(hlsEdgeArr, hlsEdge.data, sizeof(uint8_t) * OUT_H * OUT_W);
108.            memcpy(hlsEmboArr, hlsEmbo.data, sizeof(uint8_t) * OUT_H * OUT_W);
109.            memcpy(hlsSharArr, hlsShar.data, sizeof(uint8_t) * OUT_H * OUT_W);
110.            int edgeErr = 0, emboErr = 0, sharErr = 0;
111.            for (int k = 0; k < OUT_H*OUT_W; k++) {
112.                if (abs(hlsEdgeArr[k] - out_edge[k]) > 5)
113.                    edgeErr++;
114.                if (abs(hlsEmboArr[k] - out_embo[k]) > 5)
115.                    emboErr++;
116.                if (abs(hlsSharArr[k] - out_shar[k]) > 5)
117.                    sharErr++;
118.            }
119.            printf("edge err: %d, embo err: %d, shar err:%d\n", edgeErr, emboErr,
     sharErr);
120.            return edgeErr + emboErr + sharErr;
121.            */
122.            auto end = chrono::steady_clock::now();
123.            auto diff = end - start;
124.            //printf("\nTime it took:\n");
125.            //cout << chrono::duration <double, nano>(diff).count() << " ns" << e
    ndl;
126.            double nanoseconds = std::chrono::duration<double, std::nano>(diff).c
    ount();
127.            //cout << nanoseconds << " ns";
128.            return nanoseconds;
129.        }
130.
131.
132.        void reader() {
133.            cv::Mat image;
134.            image = cv::imread("1-
    10x10.png", cv::IMREAD_GRAYSCALE); // Read the file
135.            if (image.empty()) // Check for invalid input
136.            {
```

```
137.                cout << "Could not open or find the image" << std::endl;
138.                //return -1;
139.            }
140.            uint8_t imageArr[IN_H*IN_W];
141.            memcpy(imageArr, image.data, sizeof(uint8_t) * IN_H * IN_W);
142.            for (int i = 0; i < IN_W*IN_H; i++) {
143.                printf("%d,", imageArr[i]);
144.            }
145.        }
146.
147.        int main(int argc, char* argv)
148.        {
149.            double t = 0;
150.            for (int i = 0; i < 100; i++) {
151.                t = t + module(0);
152.            }
153.            double avg = t / 100;
154.            cout << avg << " ns\n";
155.
156.
157.            //reader();
158.            /*
159.            int err = 0;
160.            for (int i = 0; i < 100; i++) {
161.                err = err + module(i);
162.            }
163.            printf("Total Error: %d\n", err);
164.            */
165.            system("pause");
166.            return 0;
167.        }
```

۷- کد نوشته شده برای اجرا روی پردازنده‌ی زینک

```
1.  #include <stdio.h>
2.  #include "platform.h"
3.  #include "xil_printf.h"
4.  #include <xconv.h>
5.
6.  XConv myip;
7.
8.  int main()
9.  {
10.     init_platform();
11.
12.     xil_printf("Initializing myip\r\n");
13.     XConv_Initialize(&myip, XPAR_CONV_0_DEVICE_ID);
14.     XConv_Start(&myip);
15.
16.     print("Hello World\n\r");
17.
18.     cleanup_platform();
19.     return 0;
20. }
```

Amirkabir University of Technology
(Tehran Polytechnic)

Computer Science and Information Technology

Implementation of a Convolutional and a Pooling Layer of a CNN on FPGA

By
Sina Mahdipour Saravani

Supervisor
Dr. Reza Safabakhsh

June 2019