

«به نام او»



گزارش نهایی پروژه

طراحی سیستم‌های کم‌توان

دکتر اجلالی

۴۰۱۲۱۲۴۴۳ مهدی قصاب

لینک گیت‌هاب:

<https://github.com/MGhassab/LowPower.git>

تابستان ۱۴۰۲

فهرست

۴	بخش اول : مقدمه
۴	شرح کلی پروژه
۸	خلاصهای از کارهای انجام شده در فاز دوم
۱۰	فاز سوم
۱۱	بخش دوم : کارهای پیشین
۱۱	ارتباطات تقریبی برای شبکه‌های روی تراشه
۱۶	تقریب ارتباطات مبتنی بر زمان slack
۲۰	بهینه‌سازی روش کنترل ارتباطات تقریبی
۲۴	بخش سوم : مباحث نظری
۲۴	پردازش تقریبی
۲۶	مدلهای رنگی
۳۰	پردازش تصویر
۳۲	ابزار HLS
۳۵	الگوریتم Sobel
۳۸	الگوریتم Canny
۴۰	الگوریتم Roberts
۴۳	بخش چهارم : نحوه‌ی پیاده‌سازی تشخیص لبه
۴۵	توضیح کد ماژول تشخیص لبه با زبان C
۴۷	ساخت پروژه با ابزار HLS
۴۹	سنتر کد VHDL با استفاده از کد C با ابزار HLS
۵۱	توضیح کد VHDL ماژول تشخیص لبه
۶۲	نتایج شبیه‌سازی ماژول تشخیص لبه
۶۴	گزارش توان

۶۷	گزارش کلاک
۶۸	گزارش منابع
۶۹	نکات تكميلی
۷۰	بخش پنجم : نحوه پياده‌سازی تبدیل فضای رنگی RGB به YCbCr
۷۰	توضیح کد ماژول تبدیل فضای رنگی RGB به YCbCr با زبان C
۷۱	استخراج IP از کد C با ابزار HLS
۷۵	گزارش توان
۷۶	گزارش کلاک و منابع
۷۷	نحوه پیدا کردن مقادیری که باید در واحد memoization ذخیره شوند
۷۸	نتایج شبیه‌سازی ماژول تبدیل فضای رنگی
۸۰	نکات تكميلی
۸۱	بخش ششم : چالش‌های پياده‌سازی
۸۱	وقوع خطا هنگام شبیه‌سازی کد C در نرمافزار HLS
۸۲	وقوع خطا هنگام استخراج IP در نرمافزار HLS
۸۴	استخراج توان ماژول براساس ورودی‌های خاص
۸۶	بخش هفتم : مقایسه نتایج پياده‌سازی با مقاله
۸۶	جدول‌های ۲، ۳، ۴، ۵ و ۶ مقاله : دلیل مطرح کردن ایده
۸۷	جدول‌های ۷ و ۸ مقاله : توان مصرفی، معیار PSPA و سرباز سخت‌افزاری تشخیص لبه
۸۹	شکل‌های ۹ و ۱۰ مقاله : خروجی ماژول تشخیص لبه
۹۰	جدول‌های ۱۱ و ۱۲ مقاله : توان مصرفی، معیار PSPA و سرباز سخت‌افزاری تبدیل فضای رنگی
۹۳	شکل ۱۳ مقاله : خروجی ماژول تبدیل فضای رنگی
۹۴	نمودار توان بر حسب مقدار پیکسل‌های تقریبی
۹۵	مقایسه طراحی‌های مختلف
۹۶	بخش هشتم : نتیجه‌گیری و جمع‌بندی
۹۸	مراجع

بخش اول : مقدمه

در این گزارش، ابتدا در بخش اول یک شرح کلی از پروژه و کارهای انجام شده در فازهای دوم و سوم را بطور خلاصه بیان کردیم، سپس در بخش دوم کارهای پیشین انجام شده در زمینه محاسبات تقریبی را شرح داده و در بخش سوم مباحث نظری و پایه‌ای موردنیاز برای فهم پروژه را شرح دادیم. در بخش چهارم نحوه پیاده‌سازی مقاله را بطور کامل شرح داده و در بخش پنجم چالش‌هایی که در زمینه پیاده‌سازی به آن‌ها برخور迪م را بیان کردیم و در بخش ششم نیز نتایج بدست آمده از پیاده‌سازی را مطرح و با نتایج آورده شده در مقاله مقایسه کردیم. در انتها نیز با توجه به نتایج بدست آمده به نتیجه‌گیری پرداخته و پیشنهاداتی برای کارهای بعدی ارائه دادیم.

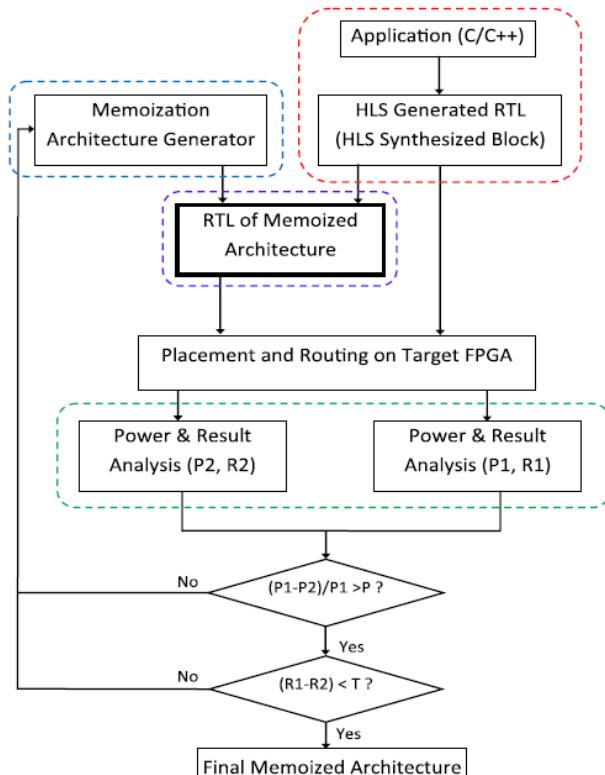
شرح کلی پروژه

در ادامه نحوه‌ی پیاده‌سازی و خروجی نهایی این مقاله را توضیح می‌دهیم تا دید مناسبی از پروژه‌ای که قرار است آن را پیاده‌سازی کنیم، داشته باشیم. در این مقاله، در ساده‌ترین حالت، اگر دو ورودی که به طور زمانی با هم فاصله دارند، با یک آستانه کوچک با یکدیگر تفاوت داشته باشند یا از یک الگوی یکسان پیروی کنند، آنگاه مقدار خروجی محاسبه شده یک ورودی را می‌توان به عنوان خروجی برای ورودی دیگر استفاده کرد که این کار منجر به تقریب می‌شود. اجرای محاسبات تقریبی مبتنی بر حافظه بر روی FPGA با تعدادی چالش مواجه است. از آنجایی که ذخیره‌سازی به مقایسه مقادیر ورودی نیاز دارد، منطق مقایسه و عملیات خواندن و نوشتن در یک بلوک از منبع حافظه، سربار اضافی را متحمل خواهد کرد که باید به وضوح در برابر صرفه‌جویی در انرژی متعادل شود. منابع اضافی مانند اینها همچنین بار اضافی را در شبکه clock FPGA ایجاد می‌کنند و در صورت عدم مدیریت صحیح ممکن است بر حداکثر فرکانس کاری طراحی تأثیر بگذارند. محاسبات تقریبی مبتنی بر حافظه از دیدگاه کاربردی نیز چالش برانگیز است. برای دستیابی به حداکثر صرفه‌جویی در مصرف در حالی که کمترین کیفیت را کاهش می‌دهد، باید معیارهای مقایسه متفاوتی با توجه به نوع کاربرد انتخاب شود.

این مقاله، روند تولید معماری مبتنی بر حافظه را با افزودن یک مرحله پس‌پردازش به جریان‌های HLS موجود، انجام می‌دهد. شکل شماره ۱ نمای کلی روند تولید معماری مبتنی بر حافظه را نشان می‌دهد. این شکل

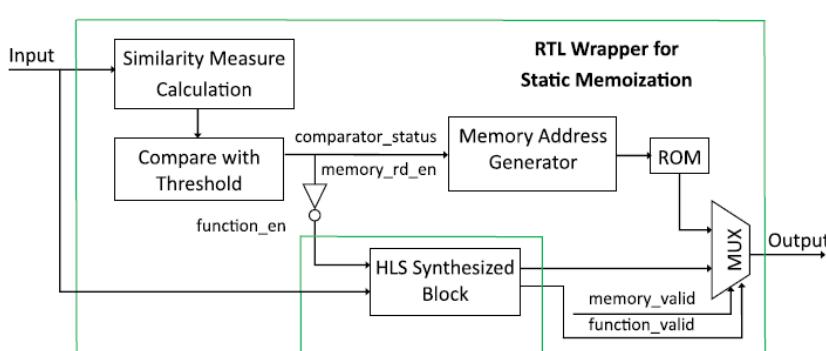
تکه‌هایی از کد یک برنامه با زبان C را نشان می‌دهد که در آن تابعی به نام HLS برای edge_detect مشخص شده است. همچنین یک فایل پیکربندی نمونه (config.tcl) را نشان می‌دهد که تابعی را که برای حافظه‌سازی مشخص شده، نشان می‌دهد. فایل پیکربندی پارامترهای مهم دیگری مانند اندازه‌گیری شباهت، حد آستانه و غیره را نیز به عنوان ورودی می‌گیرد. فایل source ابتدا توسط یک ابزار HLS، مانند LegUp Vivado HLS یا پردازش می‌شود، این عمل منجر به تولید یک فایل RTL به نام edge_detect.v است که توصیفات انتزاعی یک الگوریتم را در یک زبان سطح بالا، مانند C، به ریز معماری دیجیتال تبدیل می‌کند. ابزارهای موجود HLS عبارت‌اند از Catapult-C، LegUp، Vivado HLS و غیره. فایل edge_detect.v به عنوان ورودی توسط سازنده معماری حافظه‌سازی گرفته می‌شود که فایل پیکربندی را نیز می‌خواند. در نتیجه، سازنده معماری حافظه‌سازی فایل‌های Top_edge_detect_Memoize.v و RTL_Wrapper.v را تولید می‌کند.

تولید شده HLS را نمونه‌سازی می‌کند و این دو با هم طراحی سطح بالا Top_edge_detect_Memoize.v و RTL_Wrapper.v را تشکیل می‌دهند. RTL_Wrapper.v شامل بلوک‌های منطقی اضافی (مانند مقایسه‌کننده، حافظه و غیره) است که برای حافظه‌سازی ضروری می‌باشند.



شکل شماره ۱. روند طراحی پیشنهادی برای محاسبات تقریبی مبتنی بر حافظه

اگر یک روند طراحی را برای محاسبات تقریبی مبتنی بر حافظه‌سازی، تکراری فرض کنیم، جزئیات این روند طراحی تکراری در شکل شماره ۱ نشان داده شده است. در اینجا، P1 و P2 به ترتیب به مقادیر توان و R1 و R2 به ترتیب به مقادیر محاسبه‌شده‌ای اشاره دارند که بدون حافظه‌سازی و با حافظه‌سازی به دست می‌آیند. P و T به ترتیب آستانه توان و دقت نتیجه هستند. شکل ۱ نه تنها روند تولید معماری حافظه‌سازی را نشان می‌دهد، بلکه ملاحظات مربوط به توان و دقت نتایج را نیز نشان می‌دهد که باید در نظر گرفته شوند. بلوک قرمز نشان می‌دهد که یک برنامه یا وظیفه توصیف شده در زبان C با استفاده از ابزار HLS، سنتز شده است. بلوک آبی مولد معماری حافظه‌سازی را نشان می‌دهد که ماژول RTL را ایجاد می‌کند تا بلوک سنتز شده HLS را با بلوک‌های مدار مرتبط با حافظه‌سازی، ترکیب کند. در نتیجه این ترکیب، یک طراحی RTL از معماری حافظه‌دار تولید می‌شود (بلوک بنفش)، یعنی ماژول سطح بالا که حاوی طرح RTL و بلوک سنتز شده HLS است. پس از قرار دادن و مسیریابی بر روی FPGA هدف یا شبیه‌سازی با استفاده از یک ابزار مسیریابی خاص (مانند Xilinx ISE)، تحلیل توان دینامیکی مبتنی بر شبیه‌سازی هر دو طرح سنتز شده HLS و معماری حافظه‌سازی به طور جداگانه برای ارزیابی صرفه‌جویی در توان (سبز) انجام می‌شود. درصد اختلاف بین P1 و P2 باید بیشتر از آستانه تعریف شده P توسط کاربر باشد. اتلاف توان ناشی از طراحی سنتز HLS (P1) و اتلاف توان ناشی از معماری حافظه‌دار (P2) با استفاده از ابزارهایی مانند تحلیلگر Xilinx Xpower (XPA) محاسبه می‌شود. سپس نتایج شبیه‌سازی پردازش مجموعه داده‌ها، یعنی R1 (نتیجه کاملاً صحیح، بدون تقریب) و R2 (نتیجه تقریبی به دلیل حافظه‌گذاری)، مقایسه می‌شوند تا بینیم آیا تفاوت آنها در محدوده قابل تحمل تعریف شده برای دقت (T) می‌باشد یا خیر. اگر هر یک از این شرایط ناموفق باشد، کاربر باید شرایط حافظه‌سازی (آستانه، اندازه‌گیری شباهت، و غیره) را برای مولد ذخیره‌سازی تنظیم کند تا طرح دیگری را برای ارزیابی بازسازی کند.



شکل شماره ۲. معماری واحد حافظه‌ساز

برای کاهش توان در سیستم‌های نهفته، محاسبات تقریبی^۱ به عنوان جایگزینی برای محاسبات دقیق پیشنهاد شده است. در این روش فرض شده که برنامه‌های مورد بررسی می‌توانند نتایج تقریبی و خطدار را تحمل کنند، از این رو، محاسبه دقیق غیرضروری و بدون استفاده می‌شود. برخی از نمونه‌های کاربردی عبارت‌اند از داده‌کاوی، جستجو، تجزیه و تحلیل و پردازش رسانه (صوتی و تصویری) که در مجموع به عنوان برنامه‌های کاربردی شناسایی، استخراج و جستجو (RMS) شناخته می‌شوند. تقریب می‌تواند به کاهش محاسبات و در نتیجه مصرف انرژی کمتر کمک کند. در واقع، بهره‌وری انرژی یکی از نیروهای محرکه اصلی محاسبات تقریبی است. محاسبات تقریبی در سطوح مختلف، از جمله در سطح پردازنده‌ها، زبان‌های طراحی و بلوک‌های محاسباتی ASIC، مانند جمع‌کننده‌های غیردقیق، مورد مطالعه قرار گرفته‌اند. اگرچه از FPGA به طور گسترده برای تسريع برنامه‌های RMS استفاده می‌شود، با این حال، تنها در تعداد محدودی کار از محاسبات تقریبی به عنوان بستر محاسباتی، استفاده شده است.



تصویر دقیق



تصویر تقریبی

شکل شماره ۳. مقایسه نتیجه محاسبات تقریبی براساس معیار jpeg و خطای ۵ درصد

این مقاله بر توسعه یک واحد سازنده برای بهره‌برداری از مزایای حافظه‌سازی^۲ برای محاسبات تقریبی روی FPGA تمرکز دارد. حافظه‌سازی تکنیکی شامل استفاده مجدد از یک مقدار محاسبه شده برای همان ورودی است که بعداً دوباره وارد می‌شود و بنابراین از محاسبات مکرر جلوگیری می‌کند. به این ترتیب، حافظه‌سازی پتانسیل بهبود

¹ Approximate computing

² Memoization

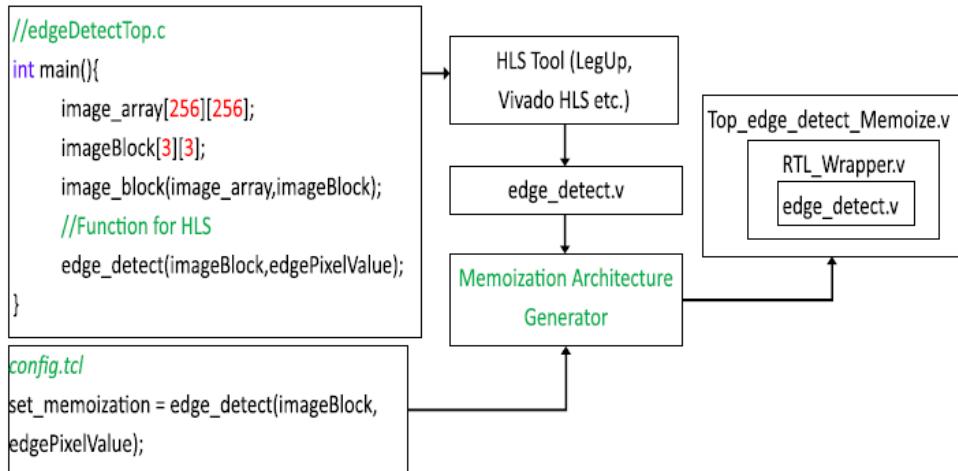
عملکرد و صرفه‌جویی در انرژی را با مبادله محاسبات با چند عملیات حافظه دارد. حافظه‌سازی به طور گستردۀ برای پردازنده‌ها، در مواردی که تمرکز بر اجرای سریع برنامه‌ها از طریق کاهش تعداد محاسبات مورد نیاز با استفاده مجدد jpeg از خروجی ذخیره شده قبلی می‌باشد، مورد مطالعه قرار گرفته است. شکل شماره ۳ نتیجه دقیق را برای معیار با نتیجه تقریبی به دست آمده در هنگام استفاده از چارچوب پیشنهادی مقایسه می‌کند که همانطور که شکل نشان می‌دهد، تفاوت بین دو خروجی ناچیز بوده و با دید انسان قابل تشخیص نیست.

این مقاله، محاسبات تقریبی مبتنی بر حافظه با FPGA را به عنوان بستر محاسباتی از تمام دیدگاه‌های مربوط به انرژی تحلیل می‌کند. همچنین ماهیت و دردسترس‌بودن مجموعه داده‌ها را مورد تجزیه و تحلیل قرار می‌دهد تا روش حافظه‌سازی مناسب را برای کیفیت بهتر نتایج تعیین کند. درواقع این مقاله از یک جریان سنتز سطح بالا (HLS) یکپارچه برای سنتز خودکار طراحی، مبتنی بر حافظه استفاده کرده است. بر اساس نتایج آزمایش‌های انجام شده، روش پیشنهادی این مقاله با استفاده از برنامه‌های پردازش تصویر و هسته‌های محاسباتی رایج با مجموعه داده‌های آن‌ها، صرفه‌جویی قابل توجهی را در مصرف توان (حدود ۲۰ درصد) نشان می‌دهد.

خلاصه‌ای از کارهای انجام شده در فاز دوم

همانطور که شکل شماره ۴ نشان می‌دهد، کار انجام شده در این مقاله از ۵ مرحله‌ی اساسی تشکیل شده است که این ۵ مرحله عبارتند از:

- پیاده‌سازی تابع تشخیص لبه با زبان C++ یا C
- راه اندازی ابزار HLS
- تبدیل کد C یا C++ تابع تشخیص لبه به کد وریلاغ
- پیاده‌سازی تولید‌کننده واحد Memoization
- پیاده‌سازی مازول نهایی و کناره‌م قرار دادن مازول‌های تشخیص لبه و حافظه



شکل شماره ۴. اجزای اصلی پروژه

هدف اصلی ما در این فاز، پیاده‌سازی ۳ مرحله‌ی ابتدایی بود. ما ابتدا با زبان C++ یک برنامه تشخیص لبه با الگوریتم Sobel نوشتیم، این برنامه بطور تصادفی یک تصویر از پوشه ورودی انتخاب و بر اساس الگوریتم Sobel، عمل تشخیص لبه را انجام داده و خروجی را در پوشه Output ذخیره می‌کند. همچنین برای اطمینان از عملکرد درست تابع نوشته شده، ورودی‌ها را به یک اسکریپت تشخیص لبه براساس الگوریتم Sobel در متلب داده و خروجی‌های بدست آمده از متلب را در پوشه Check قرار دادیم. در قدم بعدی، یک مد Test به برنامه اضافه کردیم که در آن خروجی تولید شده توسط برنامه را با خروجی متلب مقایسه کرده و در صورت وجود اختلاف، آن را به کاربر گزارش می‌دهد.

ما در این پروژه، از تصاویر با فرمت RGB استفاده می‌کنیم، زیرا مدل‌های رنگی YUV و YIQ پتانسیل بهتری را برای فشرده‌سازی تصاویر و ویدیوی دیجیتال نسبت به سایر روش‌های کدگذاری فراهم می‌نمایند، دلیل این موضوع این است که در این دو مدل رنگی بر خلاف مدل RGB، تفکیک درخشندگی و مقادیر رنگ از همدیگر امکان‌پذیر می‌باشد.

بعد از نوشتتن برنامه تشخیص لبه با زبان C++ و اطمینان از عملکرد درست تابع تشخیص لبه استفاده شده در آن، در قدم بعدی سراغ راهاندازی ابزار HLS رفتیم. ابزار Vivado HLS به عنوان یک بروزرسانی برای محیط توسعه Vivado HLx ارائه شده است. این ابزار با هدف افزایش سرعت پیاده‌سازی IP‌ها، بلوك‌های شتاب دهنده و الگوریتم‌های

پردازشی در FPGA با استفاده از C و C++ و System C عرضه شده است. این ابزار می‌تواند الگوریتم‌های ارائه شده به زبان C و C++ را به مدارات RTL جهت پیاده سازی مستقیم بر روی تراشه‌های FPGA تبدیل کند. نصب و راهاندازی ابزار HLS، به سادگی نصب خود نرم‌افزار Vivado نیست و نیاز به زمان زیادی دارد. دلیل این موضوع، وجود نسخه‌های زیاد و عدم سازگاری با اکثر سخت‌افزارهای موجود می‌باشد، همچنین فرایند کرک کردن این ابزار بشدت سخت بوده و نیاز به جست‌وجو و صرف زمان دارد. لازم به ذکر است ما در انتهای توانستیم نسخه 2019.1 Vivado را با موفقیت به همراه ابزار HLS نصب کنیم.

بعد از نصب نرم‌افزار Vivado به همراه ابزار HLS، مرحله‌ی بعدی یعنی تبدیل الگوریتم ارائه شده به زبان C++ به مدارات RTL جهت پیاده‌سازی مستقیم بر روی تراشه‌های FPGA را انجام دادیم. این مرحله چالش‌های زیادی را به همراه داشت که حل این چالش‌ها مستلزم صرف زمان زیادی بود. اول از همه ابزار HLS قابلیت تبدیل هرگونه کد C++ را ندارد و کد ما در زبان C++ باید با رعایت یکسری نکات خاص نوشته شود که یادگیری همین نکات و نحوه نوشتمن کد قابل تبدیل، نیاز به مشاهده دوره‌های آموزشی و جست و جوی زیادی دارد. همچنین برای رسیدن به ماژول RTL با عملکرد صحیح، لازم بود تا عمل تبدیل را چندین بار و با تغییر و تست چند باره کد C++ انجام دهیم که همین موضوع باعث شد زمان زیادی صرف این مرحله شود.

فاز سوم

بطور خلاصه ما در فاز دوم توانستیم ماژول تشخیص لبه را با استفاده از ابزار HLS و الگوریتم Sobel که با زبان C++ نوشته بودیم، سنتز کنیم و پایه اصلی پروژه را بسازیم. طبق شکل شماره ۱، کارهای باقی مانده برای فاز سوم شامل پیاده‌سازی واحد سازنده ماژول Memoization و پیاده‌سازی ماژول نهایی و کنار هم قرار دادن ماژول‌های تشخیص لبه و حافظه و بدست آوردن توان مصرفی در هر دو حالت می‌باشد. همچنین در این فاز ما باید کار دوم انجام شده در این مقاله یعنی تبدیل فضای رنگی RGB به YCbCr را با استفاده از محاسبات تقریبی انجام می‌دادیم، برای این منظور ما همانند کار اول از ابزار HLS استفاده کردیم. مهم‌ترین قسمت این فاز بدست آوردن توان مصرفی و نتایج بود که در بخش چهارم بطور کامل نحوه انجام آن را شرح خواهیم داد.

بخش دوم : کارهای پیشین

در این بخش به بررسی کارهای صورت گرفته در حوزه approximation که قابلیت پیاده‌سازی روی FPGA دارند و به کار صورت گرفته در مقاله اصلی نزدیک هستند، پرداخته و بطور کامل آن‌ها را شرح می‌دهیم. تلاش کردیم مقالاتی را انتخاب کنیم که در ژورنال‌های معتبر و در سال‌های اخیر چاپ شده باشند و امکان پیاده‌سازی عملی آن‌ها فراهم باشد.

ارتباطات تقریبی برای شبکه‌های روی تراشه

امروزه به دلیل مشکلات زیاد روش‌های مبتنی بر محیط‌های اشتراکی مثل باس‌ها، شبکه‌های مبتنی بر تراشه تبدیل به راه حلی استاندارد برای برقراری ارتباط بر روی تراشه‌ها شده‌اند. سامانه‌های چندهسته‌ای فعلی شامل چند هزار هسته‌ی پردازشی هستند که با وجود پیشرفت‌های قابل توجه در افزایش کارایی سامانه‌ها از طریق پیاده‌سازی ارتباطات به صورت موازی، شبکه‌های Noc بهزودی به یک گلوگاه در این سامانه‌ها تبدیل خواهند شد. شبکه‌های Noc نقش مهمی در عملکرد و کارایی پردازنده‌های چندهسته‌ای ایفا می‌کنند. با افزایش بار ارتباطی که توسط برنامه‌های محاسباتی موازی در حال ظهرور معرفی شده است، ارتباطات روی تراشه از نظر مصرف انرژی پرهزینه‌تر از محاسبات می‌باشند. مطالعات اخیر نشان می‌دهد که توان مصرفی شبکه‌های Noc می‌تواند تا ۴۰ درصد کل توان مصرفی تراشه برسد. در نتیجه، نیاز به تکنیک‌های نوآورانه کاهش توان و تأخیر برای طراحی‌های Noc وجود دارد. تحقیقات اخیر نشان می‌دهد که برنامه‌های محاسباتی، مانند تشخیص الگو، پردازش تصویر و غیره، می‌توانند خطاهای متوسطی را تحمل کنند و در عین حال نتایج قابل قبولی را به همراه داشته باشند. طراحی‌های معمولی Noc برای پردازنده‌های چندهسته‌ای تمام داده‌ها را بادقت کامل منتقل می‌کنند که برای چنین برنامه‌های محاسباتی تقریبی غیرضروری است. انتقال داده‌ها بادقت بیش از حد باعث مصرف انرژی اضافی و افزایش تأخیر شبکه می‌شود. این مشاهدات فضای طراحی جدیدی را نشان می‌دهد که در آن دقت داده‌ها می‌تواند تا حدی قربانی شود تا عملکرد بهتری در شبکه حاصل شود.

در این مقاله، یک چارچوب ارتباط تقریبی برای NOCs پیشنهاد شده است که در آن داده‌های یک بسته بر اساس میزان تحمل‌پذیری خطا، فشرده می‌شوند تا مصرف انرژی و تأخیر شبکه کاهش یابد. چارچوب ارتباط تقریبی پیشنهادی این مقاله، شامل یک روش کنترل کیفیت مبتنی بر نرمافزار و یک روش تقریب داده مبتنی بر سخت‌افزار است که در رابط شبکه^۳ (NI) پیاده‌سازی شده‌اند. الگوریتم کلی این روش به این صورت است که قبل از اجرای برنامه، روش کنترل کیفیت از تحلیلگر کد برای شناسایی متغیرهای مقاوم در برابر خطا و محاسبه تحمل خطای هر متغیر بر اساس الزامات کیفی برنامه در نتایج استفاده می‌کند. در قدم بعدی، رابط شبکه داده‌ها را فشرده کرده و با استفاده از روش تقریب داده‌های پیشنهادی، یک بسته خطا دار بر اساس تحمل خطای متغیر تولید می‌کند. چارچوب پیشنهادی، مقدار داده‌های ارسال شده را کاهش می‌دهد و درنتیجه بهبود قابل توجهی در مصرف توان و تأخیر شبکه در مقایسه با طراحی‌های NOC معمولی ایجاد می‌کند.

از چالش‌های این روش می‌توان به موارد زیر اشاره کرد:

۱- **تصمیم‌گیری نتیجه:** توانایی کنترل خطاهای ایجاد شده در طول تقریب برای اطمینان از کیفیت نتیجه، مورد نیاز است. تقریب زیاد و بدون محاسبه، می‌تواند منجر به عواقب فاجعه‌آمیزی شود، مانند نتایج اشتباه یا خرابی برنامه. در بعضی از کارهای پیشین، روش‌های کنترل کیفیت از طراحان برنامه می‌خواهد که متغیرهای مقاوم در برابر خطا و تحمل خطای آنها را در برنامه‌های محاسباتی تقریبی مشخص کنند. این روش‌ها برای تصمیم‌گیری در مورد تحمل خطا هر متغیر مقاوم در برابر خطا که می‌تواند منجر به کاهش کیفیت غیرقابل‌پیش‌بینی شود، به عامل انسانی اتکا می‌کنند؛ بنابراین، کنترل کیفیت یک چالش بزرگ برای ارتباطات تقریبی می‌باشد.

۲- **سربار کم منطق تقریب:** تکنیک ارتباطی تقریبی به یک منطق تقریب (از جمله پشتیبانی سخت-افزاری) برای فشرده‌سازی بسته‌ها بر اساس الزامات کیفیت داده‌ها نیاز دارد. چنین منطق تقریبی در مسیر بحرانی قرار می‌گیرد و از نظر تأخیر و مساحت بر روی سربار اثر می‌گذارد. تکنیک تقریب باید

³Network Interface

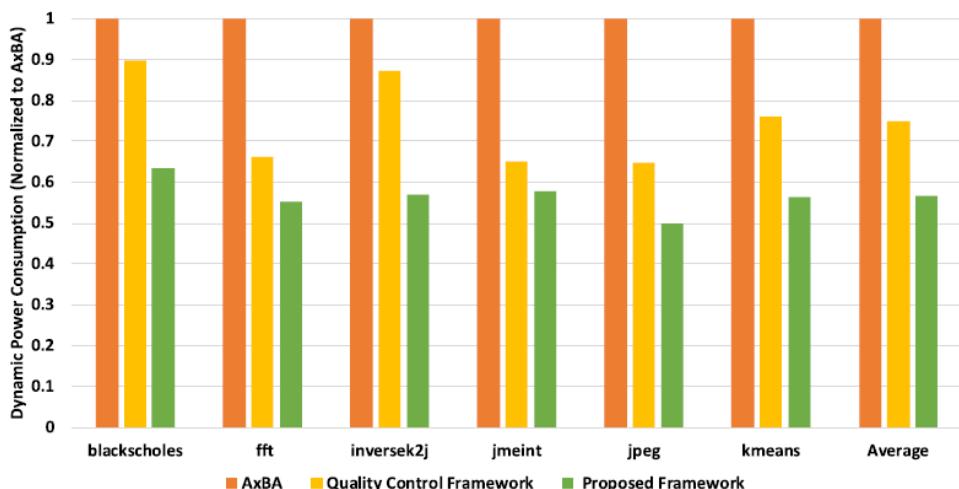
به دقت طراحی و پیاده‌سازی شود تا این هزینه‌های تقریبی از دستاوردهای به دست آمده از طریق ارتباطات تقریبی تجاوز نکند.

جدول شماره ۱ تفاوت عمدۀ بین کار پیشنهادی این مقاله و کارهای قبلی را نشان می‌دهد. تکنیک تقریب داده در DEC-NoC کد کنترل خطا (ECC) را برای مهم‌ترین بیت‌های یک عدد ممیز شناور اعمال می‌کند تا هزینه تصحیح خطا در طول انتقال را کاهش دهد. تکنیک تقریب داده‌ها در چارچوب کنترل کیفیت اندازه شناور را کوتاه می‌کند تا اندازه یک بسته را کاهش دهد. با این حال، این تکنیک‌ها برای پشتیبانی از برنامه‌های مت Shank از اعداد صحیح ناکافی هستند، زیرا هزینه‌های سربار تبدیل اعداد صحیح به اعداد ممیز شناور از نظر مصرف انرژی و تاخیر بهینه نیست. این مقاله یک روش برش داده همراه با فشرده‌سازی الگوی مکرر پیشنهاد می‌کند تا اندازه داده را برای هر دو عدد صحیح و اعداد ممیز شناور کاهش دهد. روش کنترل کیفیت در کارهای قبلی تنها به عامل انسانی برای کشف متغیرهای تقریبی متکی است. بنابراین، تکنیک‌های قبلی مقدار داده‌های تقریبی را در طول ارتباطات روی تراشه محدود می‌کنند. در چارچوب پیشنهادی این مقاله، یک تحلیل‌گر کد طراحی شده تا به طور خودکار تمام متغیرهای احتمالی مقاوم در برابر خطا را در برنامه شناسایی کند تا بتوان از ارتباطات تقریبی نهایت استفاده را برد. در نتیجه، چارچوب پیشنهادی می‌تواند به عملکرد بهتر ارتباطات روی تراشه بدون نقض الزامات کیفی برنامه‌ها دست یابد.

چارچوب	DEC-NoC	Quality Control Framework	روش پیشنهادی این مقاله
روش تقریب داده	حفظ بیت‌های پر ارزش در ممیز شناور	کوتاه کردن بیت‌های کم ارزش در ممیز شناور	فسرده‌سازی با اتلاف اعداد صحیح و ممیز شناور
روش مدیریت کیفیت	تصویر دستی و توسط عامل انسانی	تصویر دستی و توسط عامل انسانی	تصویر خودکار

جدول شماره ۱ . تفاوت کار پیشنهاد شده در این مقاله و کارهای قبلی

شکل ۵ مقادیر توان دینامیکی مصرف شده توسط تکنیکهای ارتباطی تقریبی مختلف را در مقایسه با چارچوب پیشنهادی این مقاله برای کیفیت نتیجه ۹۵ درصد نشان می‌دهد. توان مصرفی دینامیکی شامل توان دینامیکی مصرف شده توسط NI و NoC است. همانطور که در این شکل نشان داده شده است، چارچوب پیشنهادی به کاهش توان دینامیکی متوسط ۴۳ درصد در مقایسه با AxBA دست می‌یابد. بزرگترین کاهش توان دینامیکی در این آزمایش مربوط به معیار jpeg (کاهش ۵۰ درصد) و کمترین بهبود توان دینامیکی برای blackscholes (۳۶ درصد کاهش) می‌باشد. به همان دلیلی که در قسمت قبلی ذکر شد، چارچوب پیشنهادی قادر است به میزان قابل توجهی تعداد Flits در هر بسته را کاهش دهد و در عین حال کیفیت نتیجه را تضمین کند. بنابراین، مصرف توان دینامیکی چارچوب پیشنهادی در مقایسه با AxBA به ترتیب در معیارهای fft, inversek2j, kmeans و jmeint به میزان ۴۴، ۴۳، ۴۲ و ۴۳ درصد کاهش یافته است.

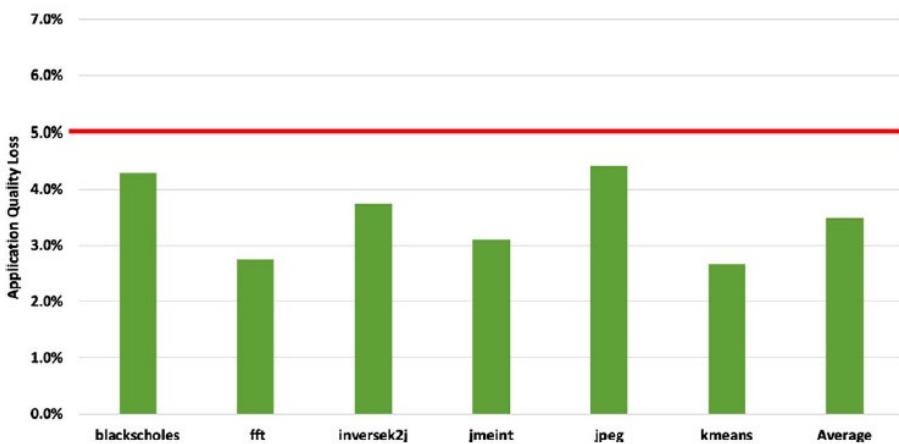


شکل شماره ۵. مقادیر توان مصرفی دینامیکی (تمامی مقادیر بصورت normalized شده

نسبت به روش AxBA بیان شده‌اند)

افت کیفیت برنامه با استفاده از معیارهای متفاوت اندازه‌گیری می‌شود که شکل شماره ۶ افت کیفیت برنامه را برای معیارهای مختلف نشان می‌دهد. خط قرمز در این شکل نشان دهنده حداقل خطاً قابل تحمل برنامه محاسباتی می‌باشد. این شکل نشان می‌دهد که روش کنترل کیفیت پیشنهادی به ترتیب ۴/۲، ۲/۷، ۳/۱، ۴/۴ و ۲/۶ درصد در معیارهای jpeg, jmeint, inversek2j, fft, blackscholes و kmeans کیفیت

از دست می‌دهد. همانطور که در شکل ۶ نشان داده شده است، افت کیفیت برای همه معیارها کمتر از ۵ درصد است و توسط برنامه محاسباتی تقریبی قابل تحمل است.



شکل شماره ۶. مقادیر افت کیفیت در کاربردهای مختلف

نتیجه گیری

در این مقاله، نگارندگان یک چارچوب ارتباط تقریبی متشکل از یک تکنیک تقریب داده و یک روش کنترل کیفیت برای NoC‌ها با توان و زمان بهینه و کم تاخیر پیشنهاد کردند. آن‌ها یک تحلیلگر کد مبتنی بر نرمافزار برای تجزیه و تحلیل بخش‌های کد طراحی کردند. این تحلیلگر کد متغیرهای تقریبی را شناسایی کرده و تحمل خطای آنها را بر اساس الزامات کیفی برنامه محاسبه می‌کند. آن‌ها همچنین یک مکانیزم ارتباطی تقریبی برای NoC‌ها پیشنهاد کردند که در آن اعداد ممیز شناور و اعداد صحیح بر اساس تحمل خطای متغیرها فشرده می‌شوند. چارچوب ارتباط تقریبی پیشنهادی، تعداد بسته‌های تقریبی را افزایش می‌دهد، بنابراین عملکرد شبکه بهتری نسبت به روش‌های تقریبی دیگر به دست می‌آورد، این موضوع در حالی است که اطمینان می‌دهد که کیفیت نتیجه، مطابق با الزامات برنامه است. آن‌ها در انتها چارچوب پیشنهادی خود را با تکنیک‌های ارتباطی تقریبی قبلی به نام AxBA مقایسه کردند. ارزیابی آن‌ها نشان می‌دهد که چارچوب ارتباط تقریبی پیشنهادی مصرف توان پویا و تأخیر شبکه را به ترتیب ۴۳ و ۶۲ درصد در مقایسه با AxBA کاهش می‌دهد.

تقریب ارتباطات مبتنی بر زمان slack

این مقاله، با پیشنهاد تکنیک تقریب slack-aware برای کاهش انرژی مصرف شده توسط NoC‌ها برای محاسبات موازی، به تحقیقات موجود در مورد ارتباطات روی تراشه کمک کرده است. روش تقریبی پیشنهادی ارائه شده در این مقاله، با کاهش اندازه بسته بر اساس زمان slack، هم زمان اجرا و هم مصرف انرژی NoC را کاهش می‌دهد. Slack تعداد چرخه‌هایی است که یک بسته را می‌توان در شبکه به تأخیر انداخت بدون اینکه تأثیری بر زمان اجرا داشته باشد. در این روش، بسته‌های با slack کم برای عملکرد سامانه حیاتی در نظر گرفته شده و اولویت‌بندی این بسته‌ها در حین انتقال، زمان اجرا را به طور چشمگیری کاهش می‌دهد. تکنیک پیشنهادی شامل یک مکانیزم کنترل slack-aware برای شناسایی بسته‌هایی با slack کم و تسريع این بسته‌ها با استفاده از دو مکانیزم تقریب بسته، یعنی یک تقریب درون شبکه (INAP) و یک تقریب رابط شبکه (NIAP) است. مکانیسم INAP بسته‌های با slack کم را در مرحله داوری مسیریاب با تقریب بسته‌ها با slack بالاتر اولویت‌بندی می‌کند. مکانیسم NIAP تأخیر لینک شبکه و سوئیچ پیمایش را با کوتاه‌کردن داده‌ها برای بسته‌های با slack کم کاهش می‌دهد. برای پشتیبانی تکنیک پیشنهادی، نیاز است تا رابط شبکه و مسیریاب با سخت‌افزار تقریب، برای مصرف انرژی و زمان اجرای کمتر پیاده‌سازی شود.

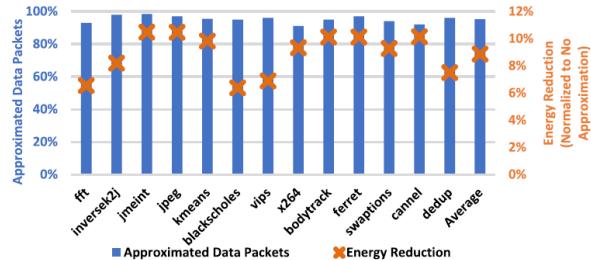
از جمله چالش‌هایی که طراحان تراشه با آن روبرو هستند، می‌توان به کارایی، مصرف توان، مدیریت دما، تغییرات فرایند ساخت و سربار مساحت اشاره کرد. همچنین پیشرفت در روش‌های کوچک‌سازی و ساخت ترانزیستور، امکان ادغام میلیاردی ترانزیستور را در یک تراشه واحد فراهم کرده است که به ایجاد سامانه چندپردازنده‌ای روی تراشه با هزاران هسته منجر می‌شود. برای رفع نیازهای ارتباطی این سامانه‌ها، شبکه روی تراشه به عنوان سوئیچ‌های قابل تنظیم متعدد از لینک‌های فیزیکی، مسیر یاب‌ها و رابطه‌های شبکه معرفی شد که ارتباطات قابل اعتماد، انعطاف‌پذیر، مقیاس‌پذیر و با عملکرد بالایی ارائه می‌دهد. امروزه با توجه به گستردگی پردازنده‌های چندهسته‌ای، مصرف توان شبکه‌های روی تراشه درصد قابل توجهی از کل توان تراشه‌ها را در بر می‌گیرد و از آنجایی که با فرها بیش از نیمی از این توان را تشکیل می‌دهند، با کاهش اندازه بافر یا حذف آن می‌توان میزان مصرفی را کاهش داد. از طرفی، کاهش اندازه بافر می‌تواند بر عملکرد

شبکه روی تراشه تأثیر منفی بگذارد. از روش‌های کاهش توان مصرفی شبکه‌های روی تراشه می‌توان به قطع متناوب توان اشاره کرد که جریان نشتی را با خاموش کردن ترانزیستورها در زمان بیکار بودن قطع می‌کند. از آنجایی که استفاده از این روش در شرایط بارهای ترافیکی سنگین ممکن است منجر به سربار کارایی قابل توجهی شود، برای جبران کاهش کارایی ناشی از قطع متناوب توان، استفاده از ساختار شبکه روی تراشه چندگانه پیشنهاد شده است که در آن هر مسیر یاب به زیر شبکه‌های کوچک‌تری تقسیم می‌شود. از دیگر روش‌هایی که برای به حداقل رساندن مصرف توان استفاده می‌شود، می‌توان به مقیاس‌بندی ولتاژ و فرکانس پویا، GALS و clock gating اشاره کرد. همچنین محدودیت‌های توان طراحی حرارتی (TDP) میزان اتلاف حرارتی مجاز را محدود می‌کند. این موضوع باعث می‌شود که بخش‌هایی از تراشه برای کاهش اتلاف حرارتی در فرکانس پایین‌تر کار کنند یا غیرفعال بمانند. محققان اخیراً در حال بررسی راههایی برای بهره‌برداری از این پدیده که dark silicon نامیده می‌شود، برای کاهش مصرف توان هستند.

مقایسه روش پیشنهادی مقاله با سایر روش‌ها

به دلیل بارهای ارتباطی سنگین NoC در برنامه‌های کاربردی محاسبات موازی فعلی، مانند برنامه‌های کاربردی داده‌های بزرگ و یادگیری ماشین، NoC در حال تبدیل شدن به یک گلوگاه جدی است که بر مصرف انرژی و زمان اجرا تأثیر می‌گذارد. تحقیقات موجود نشان می‌دهد که انرژی مصرف شده برای ارتباطات روی تراشه می‌تواند به راحتی از انرژی مصرف شده برای محاسبه در یک پردازنده چندهسته‌ای تجاوز کند؛ بنابراین، نیاز به تکنیک‌های نوآورانه کاهش انرژی برای طراحی‌های NoC آینده وجود دارد. تحقیقات اخیر بر روی ارتباطات تقریبی از تحمل خطای برنامه‌ها برای کاهش تأخیر شبکه و مصرف توان پویا استفاده می‌کنند. یک چارچوب ارتباطی تقریبی معمولی از دو جزء تشکیل شده است: روش کنترل کیفیت و تکنیک تقریب بسته. روش‌های کنترل کیفیت در نرم‌افزار پیاده‌سازی می‌شوند تا کیفیت نتیجه را با تجزیه و تحلیل هر برنامه و متغیرهای مقاوم در برابر خطای تضمین کنند. تکنیک‌های تقریب بسته در NoC برای کاهش اندازه متغیرهای مقاوم در برابر خطای قبل از ارسال بسته پیاده‌سازی می‌شوند. این کاهش در مقدار داده‌های ارسالی منجر به

بهبود عملکرد NoC از جمله مصرف انرژی پویا کمتر، کاهش تأخیر شبکه و افزایش توان عملیاتی شبکه می‌شود. با این حال، کاهش مصرف انرژی دینامیکی به تنها یکی برای کاهش انرژی کافی نیست. از آنجایی که انرژی محصول توان و زمان اجرا است، کلید ارتباط با انرژی کارآمد بر روی تراشه، کاهش هر دو مقدار با استفاده از روش‌های تقریب است. در اکثر کارهای صورت گرفته در این زمینه از مکانیسم‌های فشرده‌سازی با اتلاف داده‌ها برای دستیابی به یک بسته با اندازه کوچک‌تر برای عملکرد بهتر NoC استفاده شده است، با این حال، روش‌های کنترل کیفیت مبتنی بر نرم‌افزار توانایی محدودی در کاهش زمان اجرا دارند. از آنجایی که نرم‌افزار نمی‌تواند ارتباطات روی تراشه را قبل از اجرا پیش‌بینی کند، بسته‌های کلیدی که در صورت تأخیر در حین انتقال تأثیر قابل توجهی بر زمان اجرا می‌گذارند، نمی‌توانند در طول ارتباط شناسایی و تسریع شوند. در نتیجه، حتی اگر تعداد زیادی بسته تقریبی وجود داشته باشد، تکنیک‌های معرفی شده در کارهای به بهبود نسبتاً کمی در مصرف انرژی منجر می‌شود. همان‌طور که در شکل شماره ۷ نشان‌داده شده است، به طور متوسط ۹۵٪ از بسته‌های داده تقریبی هستند، اما این رویکرد مصرف انرژی را به طور متوسط تنها ۸٪ کاهش می‌دهد.



شکل شماره ۷. مقادیر کاهش انرژی با روش ارتباط تقریبی در معیارهای مختلف

یکی دیگر از تکنیک‌های پرکاربرد برای کاهش انرژی در NoC‌ها، تغییر دینامیکی ولتاژ و فرکانس (DVFS) است. در این تکنیک، NoC ابتدا به چندین حوزه ولتاژ/فرکانس تقسیم می‌شود، سپس، یک سیاست کنترلی برای تنظیم پویا ولتاژ‌های تغذیه و فرکانس اجزای NoC با توجه به وضعیت مشاهده شده یا پیش‌بینی شده در شبکه (به عنوان مثال، استفاده از لینک، ترافیک کش و غیره) ایجاد می‌شود. از طریق کاهش ولتاژ و فرکانس، مصرف کلی توان برای ارتباطات روی تراشه کاهش می‌یابد. با این حال، این کاهش در ولتاژ و فرکانس به طور هم‌زمان باعث می‌شود زمان اجرا طولانی‌تر شود که این موضوع منجر به کاهش اندک انرژی مصرفی

در NoC می‌گردد؛ بنابراین، برای دستیابی به صرفه‌جویی قابل توجه در مصرف انرژی NoC، کاهش توان و زمان اجرا بسیار مهم است.

شرح روش پیشنهادی

در این مقاله، یک تکنیک تقریب slack-aware را برای دستیابی به کاهش مصرف انرژی NoC برای محاسبات موازی پایدار پیشنهاد شده است. slack یک بسته به عنوان تعداد چرخه‌هایی تعریف می‌شود که بسته می‌تواند در شبکه به تأخیر بیفتند بدون اینکه تأثیری بر زمان اجرا داشته باشد. بسته‌هایی با slack کم برای عملکرد سامانه حیاتی در نظر گرفته می‌شوند و تأخیر شبکه برای این بسته‌ها اغلب منجر به توقف پردازنده می‌شود. در طرف مقابل، بسته‌های با slack بالا می‌توانند تأخیر قابل توجهی در شبکه را بدون ایجاد توقف پردازشگر تحمل کنند. روش پیشنهادی این مقاله شامل یک مکانیزم کنترل slack-aware برای شناسایی بسته‌ها با slack کم و تسريع این بسته‌ها برای کاهش مصرف توان NoC و زمان اجرای برنامه‌ها با استفاده از دو تکنیک تقریب بسته است که این دو تکنیک تقریب، تقریب درون شبکه (INAP) و تقریب رابط شبکه (NIAP) می‌باشد. مکانیزم NIAP از روش تقریب داده‌ها (قطع) برای کاهش تأخیر لینک‌ها و سوئیچ‌های شبکه برای بسته‌های با slack کم، استفاده می‌کند. مکانیزم INAP برای کاهش تأخیر داوری در مسیر یاب ها برای بسته‌ها با slack کم طراحی شده است. برای پشتیبانی از تکنیک پیشنهادی، یک رابط شبکه و یک مسیریاب تقریبی پیاده‌سازی شده‌اند. پیاده‌سازی‌های پیشنهادی اندازه بسته را با استفاده از منطق تقریب داده‌های سبک وزن برای مصرف انرژی کمتر، کاهش می‌دهند. کاهش هم‌زمان در زمان اجرا و توان مصرفی منجر به کاهش مصرف انرژی NoC می‌شود.

ارزیابی

تکنیک پیشنهادی مصرف توان NoC را در عین حال که زمان اجرا را بهبود می‌بخشد، کاهش می‌دهد و میزان افت کیفیت را برای برنامه‌ها، به حداقل می‌رساند. ارزیابی آن‌ها نشان می‌دهد که تکنیک تقریب slack-

زمان اجرا را تا ۲۴ درصد و مصرف انرژی را تا ۳۸ درصد در مقایسه با تکنیک‌های ارتباطی تقریبی موجود با ۱/۱ درصد خطای نتیجه کمتر، کاهش می‌دهد.

نتیجه‌گیری

این مقاله به طور خلاصه شامل موارد زیر می‌باشد:

۱- یک تکنیک تقریب slack-aware برای کاهش مصرف انرژی NoCs برای محاسبات موازی پایدار پیشنهاد شده است.

۲- یک مسیریاب تقریبی و یک رابط شبکه تقریبی برای پشتیبانی از سیاست کنترل slack-aware مکانیسم INAP و مکانیسم NIAP برای کاهش تأخیر شبکه برای بسته‌های با slack کم پیاده‌سازی شده‌اند.

بهینه سازی روش کنترل ارتباطات تقریبی

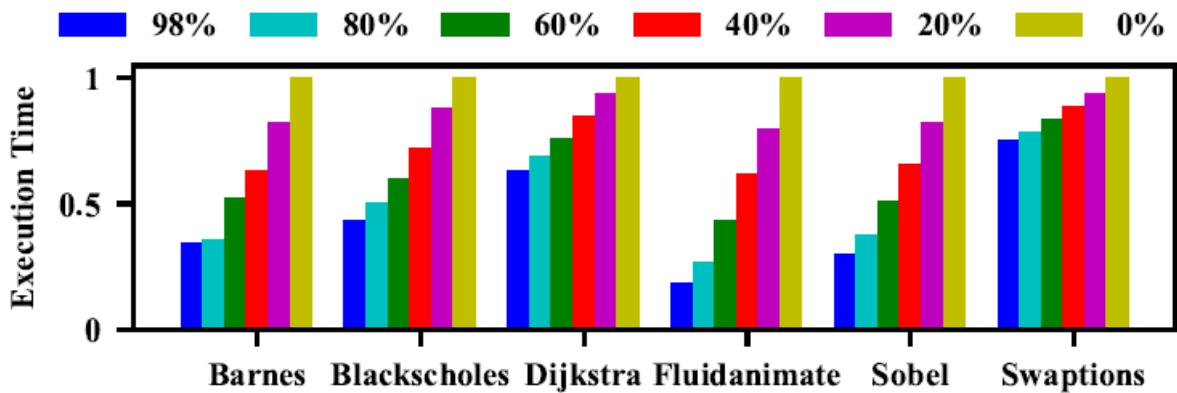
برای بسیاری از برنامه‌های کاربردی، می‌توان با پذیرفتن مقداری خطأ، دقت خروجی برنامه را برای کاهش اعمال محاسباتی/ ارتباطاتی که منجر به سود عملکرد/ انرژی می‌شود، مبادله کرد. از آنجایی که ارتباطات روی تراشه یکی از عوامل اصلی در عملکرد سامانه و مصرف انرژی است، ارتباطات زیربنایی برای دستیابی به بهبود زمان/ انرژی است، با این حال، انجام تقریب کورکرانه باعث کاهش کیفیت غیرقابل قبول می‌شود. در این مقاله، ابتدا یک مسئله بهینه‌سازی برای به حداقل رساندن عملکرد NoC با محدودیت کیفیت موردنیاز، فرمول‌بندی می‌شود و افت کیفیت خروجی برنامه مورد مطالعه قرار می‌گیرد. در قدم دوم، یک روش کنترل کیفیت congestion-aware برای بهبود عملکرد سامانه با حذف شدید داده‌های شبکه بر اساس پیش‌بینی جریان شبکه، پیشنهاد می‌شود. نتایج آورده شده در این مقاله نشان می‌دهد که روش پیشنهادی این مقاله

می‌تواند سرعت اجرا را تا ۲۹/۴۲ درصد افزایش دهد. هر یک از تقریب کننده‌هایی که در واقع محاسبات/ارتباطات تقریبی را انجام می‌دهند با روش کنترل کیفیت کار می‌کنند. یک روش کنترل کیفیت برای استفاده مناسب از تقریب کننده باید به گونه‌ای طراحی شود که افت کیفیت نهایی در محدوده تعريف شده توسط کاربر باشد. اگر یک روش کنترل کیفیت به خوبی طراحی نشده باشد، یا به استفاده کم از تقریب (اجام تقریب به صورت محافظه‌کارانه به‌طوری که بهبود عملکرد به طور کامل مورد بهره‌برداری قرار نگیرد) یا استفاده بیش از حد از تقریب (اجام تقریب بیش از حد تهاجمی که منجر به کاهش کیفیت غیرقابل قبول در خروجی برنامه می‌شود)، باعث اختلال در عملکرد و کارایی نهایی شبکه NoC می‌گردد. در میان اجزای مختلف سخت‌افزاری تراشه، شبکه‌های روی تراشه (NoCs) تأثیر زیادی بر عملکرد ارتباطات و مصرف انرژی دارند؛ بنابراین، ارتباطات تقریبی به یک طراحی جذاب تبدیل شده‌اند که بسته‌های داده را قبل از تزریق به شبکه رها می‌کنند و در مقصد بازیابی می‌کنند تا بار کاری شبکه کاهش یابد.

شرح روش پیشنهادی

در یک ارتباط تقریبی، هر رابط شبکه^۴ مجهز به یک حذف‌کننده و یک بازیاب اطلاعات است که به ترتیب برای حذف و بازیابی داده‌ها استفاده می‌شود. یک مثال آزمایشی در شکل ۱ برای نشان‌دادن امکان‌سنجی بهبود عملکرد سامانه با حذف ترافیک شبکه ارائه شده است. نرخ افت را به عنوان نسبت داده‌هایی که در هر رابط شبکه دور ریخته می‌شوند و هرگز وارد شبکه نمی‌شوند تعریف می‌کنیم. در این مثال، شبکه روی تراشه از یک شبیه‌ساز کامل سامانه چندهسته‌ای برای تغییر اندازه بسته‌های داده به طور یکنواخت با توجه به نرخ افت معین، درست قبل از تزریق به شبکه، استفاده کرده است. نرخ افت از ۰٪ (کاهش) تا ۹۸٪ (تقریباً کاهش یافته) متغیر است. همان‌طور که در شکل شماره ۸ نشان‌داده شده است، حذف داده‌ها می‌تواند زمان اجرا را به میزان قابل توجهی کاهش دهد. به عنوان مثال، حذف ۶۰ درصد داده‌ها باعث افزایش سرعت به میزان ۲۶ درصد در مقایسه با حالتی است که فقط ۲۰ درصد داده‌ها حذف می‌شوند.

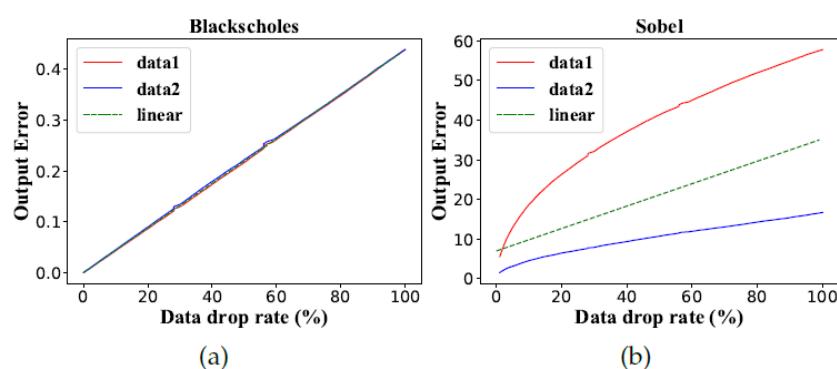
⁴ Network Interface



شکل شماره ۸. زمان اجرا در نرخ‌های مختلف رها کردن داده در برنامه‌های مختلف

مقایسه روش پیشنهادی مقاله با سایر روش‌ها

کارهای قبلی روی NoC‌های تقریبی قادر به حداکثر رساندن افزایش عملکرد معرفی شده توسط تقریب نیستند. تراکم شبکه را در نظر نمی‌گیرند. از آنجایی که تأخیر شبکه به ازدحام شبکه حساس است، حذف داده‌ها بدون آگاهی از تراکم باعث استفاده ناکافی از تقریب می‌شود (انجام تقریب روی بسته‌های غیر بحرانی به گونه‌ای که سود آن کمتر قابل توجه باشد). علاوه بر این، حذف داده‌ها بدون تخمین افت کیفیت خاص برنامه منجر به استفاده بیش از حد از تقریب (خطای غیرقابل قبول) می‌شود. علاوه بر بهینه‌سازی عملکرد، مدل کیفیت دقیقی برای برنامه‌ها ندارند. مدل‌های کیفیت فرض می‌کنند که خطای خروجی با توجه به خطای داده ورودی به صورت خطی رشد می‌کند. با این حال، این فرض خطی‌بودن برای بسیاری از کاربردها که در شکل شماره ۹ نشان‌داده شده است، صادق نیست.



شکل شماره ۹. میزان خطای خروجی تحت نرخ‌های رها سازی مختلف

در شکل شماره ۹، نتایج از دستدادن کیفیت با استفاده از روش حذف داده ABDTR جمع‌آوری شده است، به ترتیب «data1» و «data2» نتایج خطای خروجی با دو داده ورودی متفاوت هستند. خطچین "خطی" با رگرسیون خطی به دست‌آمده. برای Blackscholes، همان‌طور که در کار قبلی بیان شد، تقریباً خطی است. بر عکس، برای Sobel، غیرخطی‌بودن را نشان می‌دهد. علاوه بر این، شکل 7-b نشان می‌دهد که خطای خروجی نه تنها به نرخ افت داده، بلکه به داده‌های ورودی نیز بستگی دارد. کارهای قبلی این عامل مهم را نادیده گرفته‌اند که منجر به عدم دقیق در مدل‌های کیفی آنها می‌شود. در این مقاله ابتدا یک مدل کیفیت دقیق پیشنهاد شده، سپس مسئله بهینه‌سازی عملکرد را برای NOC‌های تقریبی فرموله کرده و به دنبال آن یک مکانیزم ACDC (کنترل ترافیک دینامیکی آگاه از دقت و ازدحام) برای حل آن پیشنهاد کرده است.

نتیجه‌گیری

این مقاله به طور خلاصه شامل موارد زیر می‌باشد:

- ۱- در این مقاله، از دستدادن کیفیت، تراکم شبکه و تأخیر بار، تحلیل و مدل‌سازی می‌شوند. یک مسئله بهینه‌سازی برای به حداقل رساندن تأخیر شبکه، مشروط به محدودیت خطای تعریف شده توسط کاربر، فرموله شده است.
- ۲- یک مکانیسم کنترل برای حل مشکل، پیشنهاد شده است که شامل یک کنترل‌کننده سراسری به طور دوره‌ای و کنترل‌کننده‌های محلی است. بر اساس پیش‌بینی جریان، کنترل‌کننده سراسری ابتدا مشکل کمینه‌سازی تراکم را حل می‌کند و سپس از آن برای کاهش تأخیر بار استفاده می‌کند.
- ۳- در مقایسه با کارهای مشابه، نتایج تجربی نشان می‌دهد که ACDC استفاده بیش از حد یا ناکافی از تقریب را کاهش می‌دهد. برای برخی از کاربردها (به عنوان مثال Sobel، Blackscholes) مکانیزم ACDC سرعت اجرا و صرفه‌جویی در انرژی را بیش از ۲۰٪ نسبت به دو کار اخیر بهبود می‌دهد.

بخش سوم : مباحث نظری

پردازش تقریبی

پردازش تقریبی یک الگوی رو به رشد برای طراحی کم مصرف یا عملکرد بالا است. این موضوع شامل تعداد زیادی از روش‌های محاسباتی است که نتیجه احتمالی (که کاملاً درست نیست) را به جای یک نتیجه دقیق تضمینی بازمی‌گرداند و می‌تواند برای برنامه‌هایی استفاده شود که نتیجه تقریبی برای هدف آن کافی است. برای نمونه می‌توان به موتورهای جستجو اشاره کرد که هیچ پاسخ قطعی و دقیقی نمی‌توان برای یک جستجو خاص در نظر گرفت و بنابراین، بسیاری از پاسخ‌ها ممکن است قابل قبول باشند. به‌طور مشابه، افت گاه به گاه برخی از فریم‌ها در یک برنامه ویدیویی می‌توانند به دلیل محدودیت‌های ادراکی انسان‌ها تشخیص داده نشوند.

محاسبه تقریبی بر اساس این مشاهدات استوار است که در بسیاری از زمینه‌ها، علاوه بر اینکه انجام محاسبات دقیق به منابع زیادی نیاز دارد، تقریب محدود می‌تواند دستاوردهای غیرمنتظره‌ای در عملکرد و انرژی ایجاد کند، در حالی که هنوز به نتایج قابل قبولی دست می‌یابد. به عنوان مثال، در الگوریتم خوشبندی k-means، اگر تنها ۵٪ از دقت طبقه‌بندی را کاهش دهیم، می‌توانیم ۵۰ برابر صرفه جویی انرژی در مقایسه با طبقه‌بندی کاملاً دقیق ایجاد کنیم. نکته اصلی در محاسبات تقریبی این است که فقط در داده‌های غیر اصلی و غیر مهم می‌توان از تقریب استفاده کرد چراکه تقریب داده‌های مهم (به عنوان مثال عملیات کنترل) می‌تواند عواقب فاجعه‌آمیزی ایجاد کند برای مثال می‌تواند باعث خرابی برنامه یا خروجی اشتباه شود. از چندین روش می‌توان برای انجام محاسبات تقریبی استفاده کرد که در ادامه به شرح آن‌ها خواهیم پرداخت.

مدارهای تقریبی

مدارهای حسابی تقریبی: جمع، ضرب و دیگر مدارهای منطقی می‌توانند مخارج کلی سخت‌افزار را کاهش دهند. به عنوان مثال، یک جمع‌کننده چند بیتی تقریبی می‌تواند زنجیره حمل را نادیده بگیرد و بنابراین، به همه اضافات فرعی آن اجزه می‌دهد تا عملیات جمع را به صورت موازی انجام دهند.

ذخیره‌سازی تقریبی

به جای ذخیره دقیق داده‌ها، می‌توان آنها را تقریباً ذخیره کرد، به عنوان مثال، (با) کوتاه کردن بیت‌های پایین در داده‌های نقطه شناور. روش دیگر، پذیرش حافظه کمتر موثق است. برای این منظور، در DRAM و eDRAM، می‌توان تخصیص نرخ تازه سازی را کاهش یا کنترل کرد. در SRAM، می‌توان ولتاژ منبع را کاهش داد یا کنترل کرد. می‌توان از ذخیره‌سازی تقریبی برای کاهش مصرف بالای انرژی نوشتاری MRAM استفاده کرد. به طور کلی، هر مکانیزم تشخیص و تصحیح خطا باید غیرفعال شود.

تقریب در سطح نرم‌افزار

چندین روش برای تقریب در سطح نرم‌افزار وجود دارد. به خاطر سپاری می‌تواند اعمال شود. برخی از تکرارهای حلقه‌ها را می‌توان نادیده گرفت (این روش سوراخ حلقه نامیده می‌شود) تا سریعتر به نتیجه برسیم. برخی از وظایف نیز می‌توانند نادیده گرفته شوند، به عنوان مثال هنگامی که وضعیت زمان اجرا نشان می‌دهد که این کارها مفید نخواهد بود (نادیده گرفتن کار). الگوریتم‌های مونت کارلو و الگوریتم‌های تصادفی صحت را با ضمانت زمان اجرا مبادله می‌کنند. محاسبه را می‌توان با توجه به پارادایم‌هایی که به راحتی امکان تسريع در سخت‌افزارهای تخصصی را می‌دهند (به عنوان مثال واحد پردازش عصبی)، دوباره فرموله کرد.

حوزه‌های کاربرد

از محاسبات تقریبی در حوزه‌هایی که برنامه‌ها در برابر خطاهای تحمیل پذیر هستند استفاده می‌شود، مانند پردازش چند رسانه‌ای، یادگیری ماشین، پردازش سیگنال، محاسبات علمی. به همین دلیل محاسبات تقریبی بیشتر توسط برنامه‌هایی انجام می‌شود که به ادراک (شناخت) انسان مربوط می‌شوند و ممکن است به خودی خود دارای خطای خطا باشد. بسیاری از این برنامه‌ها بر اساس محاسبات آماری یا احتمالی هستند که (مانند تقریب‌های مختلف) می‌توانند با اهداف مطلوب بهتر مطابقت داشته باشند. یکی از کاربردهای قابل توجه در یادگیری ماشین این است که گوگل از این روش در واحدهای پردازش Tensor , ASIC TPU (سفارشی) استفاده می‌کند.

مدل‌های رنگی

مبانی دید انسان از محیط پرامون با تابش نور به اجسام و بازتابش آن به چشم انسان بوجود می‌آید. نور مرئی قسمتی از طیف انرژی الکترومغناطیسی می‌باشد. محدوده‌ی این طیف بیانگر نوع رنگ می‌باشد و چشم ناظر بسته به مکان نور در طیف انرژی رنگ خاصی را مشاهده می‌نماید. مولفه‌هایی که با استفاده از آن می‌توان این انرژی الکترومغناطیسی را تعبیر کرد در ذیل بر مبنای مجمع جهانی استانداردسازی روشنایی CIE بیان شده است.

- ۱- روشنایی^۵: میزان نور دریافتی توسط چشم انسان که باعث تشخیص درخشندگتر بودن یک شی از شی دیگر می‌شود.
- ۲- تهرنگ^۶: احساس انسان از شباهت بین دو ناحیه، توسط این پارامتر تعیین می‌گردد. در اصل تهرنگ بیانگر رنگ غالبی است که چشم بیننده دریافت می‌نماید.
- ۳- پررنگی^۷: میزان کمتر یا بیشتر به چشم آمدن دو ناحیه با ته رنگ یکسان را نشان می‌دهد.
- ۴- درخشندگی^۸: میزان احساس درخشندگی یک ناحیه توسط چشم انسان نسبت به رنگ سفید مرجع.
- ۵- رنگینی^۹: میزان پررنگی یک ناحیه نسبت به درخشندگی رنگ سفید مرجع.
- ۶- اشباع^{۱۰}: میزان پررنگی یک ناحیه نسبت به درخشندگی آن ناحیه یا به عبارتی دیگر، خلوص نسبی ته رنگ آن.

مدل رنگی RGB

یکی از اولین استانداردهای مورد استفاده در حوزه تصاویر دیجیتالی که توسط CIE ارائه شده است را می‌توان استاندارد RGB بیان نمود. این استاندارد بر پایه‌ی سه رنگ اصلی قرمز، سبز و آبی می‌باشد. براساس

⁵ Brightness

⁶ Hue

⁷ Colorfulness

⁸ Lightness

⁹ Chroma

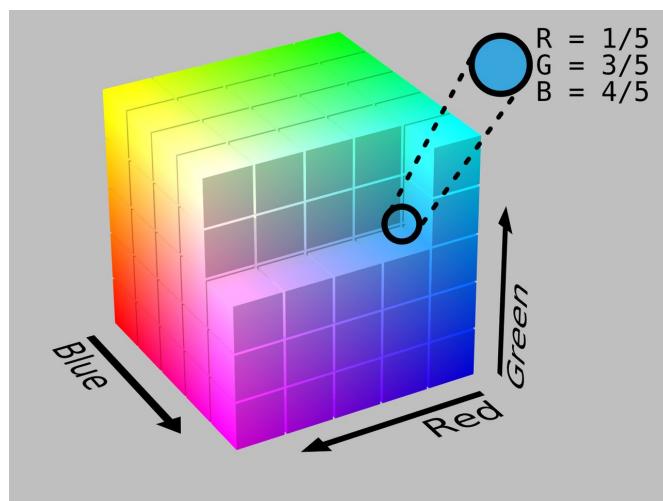
¹⁰ Saturation

این استاندارد، هر رنگ را می‌توان با استفاده از این سه رنگ اصلی تشکیل داد. مطابق این استاندارد برای این سه رنگ اصلی تعریف ذیل ارائه می‌شود.

- رنگ قرمز با طول طیف رنگ ۷۶۶ نانومتر

- رنگ سبز با طول طیف رنگ ۵۴۶۴۱ نانومتر

- رنگ آبی با طول طیف رنگ ۴۳۵۴۸ نانومتر



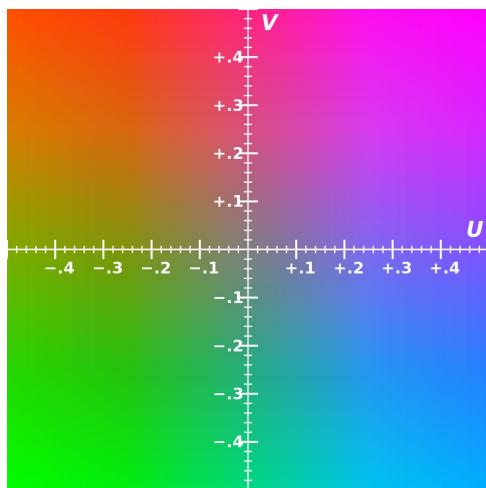
شکل شماره ۱۰. پراکندگی رنگها در استاندارد RGB

در شکل شماره ۱۰، فضای رنگ RGB به نمایش گذاشته شده است. در تصویر فوق بردار x که در راستای چپ افق امتداد داده شده است بیانگر رنگ قرمز و بردار y که در راستای راست افق امتداد داده شده است رنگ آبی و بردار z که عمود بر افق میباشد رنگ سبز را نشان می‌دهد. مرکز مختصات این تصویر که هر سه بردار دارای مقدار صفر می‌باشند را رنگ سیاه و دورترین نقطه تا مرکز مختصات که راس مکعب می‌باشد را رنگ سفید تشکیل داده است. مابقی رنگ‌ها از ترکیب مقادیر مختلف این سه بردار در داخل مکعب رنگی تشکیل می‌شود و بسته به میزان نمونه‌برداری از بردارها تعداد مقادیر مختلف تشکیل شده بیشتر و در نتیجه مکعب توانایی تشکیل رنگ‌های بیشتری را دارا خواهد بود. این مدل رنگ برای ایجاد تصویر در تلویزیون و مانیتورهای CRT, LCD, Plasma بیشتر بکار گرفته می‌شود. در این استاندارد به سه رنگ قرمز، سبز و آبی

رنگ‌های ابتدایی^{۱۱} و به رنگ‌های دیگر که از این سه رنگ مشتق می‌گردند رنگ‌های ثانویه^{۱۲} می‌گویند. تعریف فوق یک تعریف فیزیکی بر حسب طول طیف رنگ‌ها می‌باشد. در ادامه استاندارد دیگری ارائه می‌گردد که با ساختار بینایی انسان شباهت بیشتری را دارد می‌باشد.

مدل رنگی Ycber

مدل RGB بهترین راه برای ارائه یک تصویر رنگی دیجیتال نمی‌باشد. در مدل RGB هر سه مولفه قرمز، سبز و آبی به میزان یکسانی از اهمیت برخوردار بوده و باید با یک دقت یکسان هر سه مولفه را ذخیره نمود. در بعضی از کاربردها مانیازمند تغییر اندازه و مقادیر هر مولفه می‌شویم و برای این منظور از استاندارد استفاده می‌نماییم. در این استاندارد y مولفه‌ی درخشندگی^{۱۳} و از دو مولفه‌ی متفاوت رنگ cr , cb یا V , U استفاده می‌شود. مدل پراکندگی رنگ‌ها بر اساس دو محور cr , cb در شکل شماره ۱۱ نشان داده شده است.



شکل شماره ۱۱. مدل پراکندگی رنگ‌ها در YUV

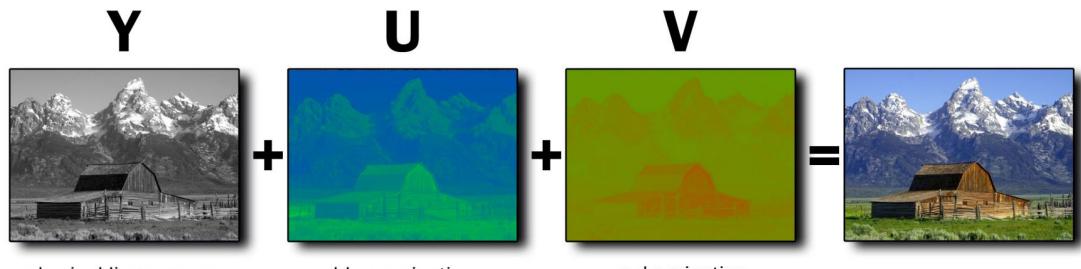
مدل‌های رنگی YUV و YIQ پتانسیل بهتری را برای فشرده سازی تصاویر و ویدیوی دیجیتال نسبت به سایر روش‌های کدگذاری فراهم می‌نمایند، زیرا در این دو مدل رنگی برخلاف مدل RGB تفکیک درخشندگی و رنگینگی از همدیگر امکان‌پذیر می‌باشد. مدل رنگی RGB به راحتی می‌تواند به مدل رنگی

¹¹ Primary

¹² Secondary

¹³ Luminance

RGB یا YUV یا YCbCr باعکس نگاشت گردد. در استاندارد ITU-R BT 601 تبدیلات این مدل به مدل RGB و YUV توصیف می‌کند.



شکل شماره ۱۲. توصیف با مدل رنگی YUV



شکل شماره ۱۳. توصیف با مدل رنگی RGB

مدل رنگی Grayscale

تصاویر با درجه‌بندی رنگ خاکستری، تصاویری هستند که از ۲۵۶ رنگ منحصر به فرد تشکیل شده‌اند.

پیکسل‌های این نوع تصاویر که مقدار آن‌ها از عدد ۰ که رنگ کاملاً مشکی است شروع می‌شود تا پیکسل‌هایی با مقدار عددی ۲۵۵ ادامه دارد که نمایانگر رنگ تماماً سفید است. سایر اعداد موجود در بازه ۰ تا ۲۵۵ رنگ خاکستری را با درجه روشنی و تیرگی مختلف نشان می‌دهند. تصاویر خاکستری و تصاویر سیاه و سفید تنها از یک کanal تشکیل شده‌اند و با دو بُعد می‌توان به مقادیر ماتریس این تصاویر دسترسی داشت. در تصاویر رنگی، از سه ماتریس هماندازه استفاده می‌شود که این ماتریس‌ها، کanal نام دارند. هر یک از این ماتریس‌ها در توالی یکدیگر قرار می‌گیرند و دارای مقادیری در بازه ۰ تا ۲۵۵ هستند.

پردازش تصویر

هوش مصنوعی و یادگیری عمیق در چند سال اخیر تأثیر شگرفی بر حوزه‌های مختلف فناوری داشته است. یکی از داغترین موضوعاتی که در این صنعت مطرح است، پردازش تصویر و بینایی ماشین است: بینایی ماشین یعنی توانایی رایانه‌ها برای «دیدن» و تفسیر دنیای اطرافشان. پردازش تصویر یا image processing یعنی کارهایی که باید انجام دهید تا رایانه تصاویر و محتوای ویدیویی را بتواند بهتر ببیند و تفسیر کند. ماشین‌های خودران، دوربین‌های کنترل جرایم رانندگی، سیستم‌های تشخیص چهره و ... همگی برای اینکه به درستی کار کنند به این دو زمینه مهم هوش مصنوعی متکی هستند. قبل از اینکه به پردازش تصویر پردازیم، ابتدا باید متوجه شویم که دقیقاً چه چیزی یک تصویر را تشکیل می‌دهد. در دنیای فناوری اطلاعات و کامپیوتر، یک تصویر با ابعاد آن (ارتفاع و عرض) بر اساس تعداد پیکسل‌ها نشان داده می‌شود. برای مثال، اگر ابعاد یک تصویر ۵۰۰ در ۴۰۰ باشد، تعداد کل پیکسل‌های تصویر ۲۰۰۰۰۰ است. این پیکسل نقطه‌ای از تصویر است که سایه، تیرگی یا رنگ خاصی را به خود می‌گیرد.

پردازش تصویر مجموعه روش‌هایی است که هدفشان دستکاری یا بهبود تصاویر است. پردازش تصویر به صورت پیکسل به پیکسل است: یعنی الگوریتم‌ها با ویژگی‌های پیکسلی تصویر سروکار دارند، مجموعه‌ای از توابع به ترتیب بر هر پیکسل از یک تصویر اعمال می‌شوند و فقط هنگامی که یک تابع عملیاتی به طور کامل انجام شد، برنامه شروع به انجام تابع دوم و ... می‌کند. پنج نوع اصلی پردازش تصویر وجود دارد:

- تجسم (Visualization): یافتن اشیایی که در تصویر قابل مشاهده نیستند
- تشخیص (Recognition): تشخیص اشیا در تصویر
- اصلاح کردن و بازیابی (Sharpening and restoration): ایجاد یک تصویر بهبودیافته و پیشرفت‌تر از روی تصویر اصلی
- تشخیص الگو (Pattern recognition): شناسایی، گروه بندی و اندازه گیری الگوهای مختلف موجود در تصویر

- بازیابی (Retrieval): مرور و جستجوی تصاویر از یک پایگاه داده بزرگ حاوی تصاویر دیجیتال که مشابه تصویر اصلی است.

کتابخانه OpenCV

در صدر کتابخانه‌های کاربردی پردازش تصویر OpenCV است، که یک کتابخانه منبع باز است که در سال ۲۰۰۰ توسط اینتل توسعه و منتشر شد. OpenCV اغلب برای کارهای بینایی کامپیوتری مانند تشخیص چهره، تشخیص اشیا، تقسیم بندی تصویر و موارد دیگر به کار می‌رود.

OpenCV که به زبان C++ نوشته شده است، در پایتون نیز وجود دارد و می‌تواند در کنار SciPy، NumPy و Matplotlib استفاده شود. یکی از بهترین جنبه‌های OpenCV این است که کتابخانه بینایی کامپیوتر به لطف مشارکت کنندگان زیادی که در گیتهاب دارد، دائماً در حال تکامل است. این کتابخانه پردازش تصویر OpenCV دسترسی به بیش از ۲۵۰۰ الگوریتم پیشرفته و کلاسیک را فراهم می‌کند. کاربران می‌توانند از OpenCV برای انجام چندین کار خاص مانند حذف قرمزی چشم و دنبال کردن حرکات چشم استفاده کنند.

لبه‌یابی تصاویر یکی از مهمترین عملیات در پردازش تصویر به شمار می‌رود. به علت کاربردهای وسیع تصاویر رنگی، لبه‌یابی این تصاویر از اهمیت ویژه‌ای برخوردار است. بطور کلی لبه‌یابی تصاویر رنگی به دو روش برداری^{۱۴} و ترکیبی^{۱۵} انجام می‌شود؛ کیفیت تشخیص لبه و زمان اجرا این الگوریتم‌ها را از یکدیگر متمایز می‌سازد. زمان اجرای الگوریتم‌های لبه‌یابی در کاربردهای واقعی بسیار حائز اهمیت است؛ بدین معنی که استفاده از الگوریتمی که لبه‌های تصویر را با کیفیت مطلوب تشخیص داده اما زمان اجرای بالایی دارد در بسیاری از کاربردها (حساس به زمان)، عمل غیر ممکن است. یکی از روش‌های جدید لبه‌یابی، الگوریتمی است که با استفاده از درخت پوشای مینیمال و در فضای رنگ YUV عملیات لبه‌یابی را انجام می‌دهد. این الگوریتم از کیفیت بالایی برخوردار می‌باشد اما زمان اجرای آن بسیار بالاست.

¹⁴ Vector

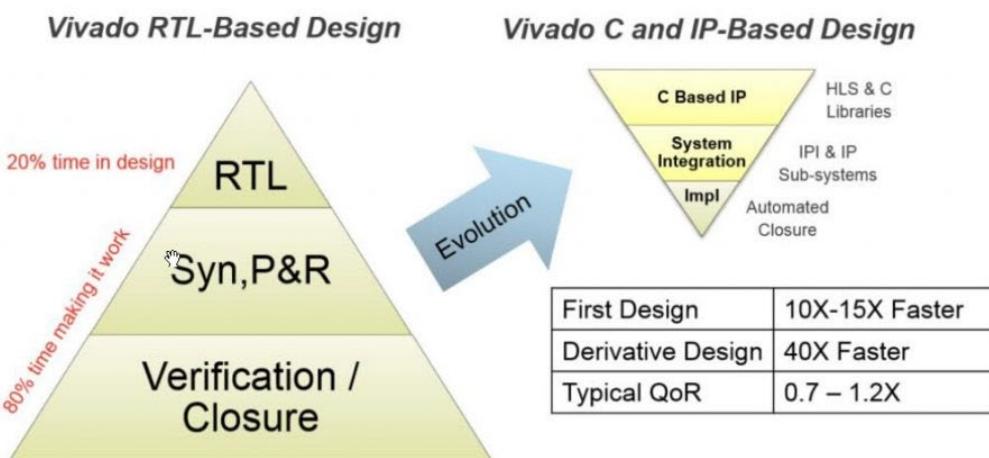
¹⁵ Synthetic

ابزار HLS

زمان بر بودن پروسه طراحی سخت افزار در سطح رجیستر RTL و همچنین انعطاف پذیری پایین در اصلاح یک مدار از پیش طراحی شده از چالش های بزرگی هستند که طراحان سخت افزار در FPGA همواره با آن روبرو هستند، از این رو شرکت های سازنده تراشه های قابل پیکره بندی FPGA همواره در صدد ارائه ابزاری جهت برنامه نویسی سطح بالا برای این تراشه ها بوده اند، در سال های اخیر شرکت Xilinx ابزار سنتر سطح بالای خود به نام Vivado HLS را ارائه داده است و همچنین تلاش های فراوانی جهت توسعه هر چه بیشتر آن انجام داده است. این ابزار می تواند الگوریتم های ارائه شده به زبان C و C++ را به مدارات RTL جهت پیاده سازی مستقیم بر روی تراشه های FPGA تبدیل کند. در حقیقت HLS بعد از زبان های HDL که جایگزین طراحی شماتیک در سطح گیت شدن، گام مهم دیگری در جهت کاهش پیچیدگی های طراحی و همچنین افزایش بهره وری است. به عبارت دیگر، سنتر سطح بالا پلی است میان سخت افزار و نرم افزار که با تمرکز دایی از سخت افزار و به واسطه بکار گیری زبان های سطح بالا، راندمان طراحی را ارتقا می بخشند. در واقع HLS با ایجاد یک حالت انتزاعی، از پیچیدگی های سخت افزاری می کاهد و به طراحان نرم افزار اجازه می دهد بدون درگیر شدن در مفاهیم طراحی سخت افزاری، از تراشه های FPGA جهت شتابدهی الگوریتم هایی که نیاز به توان پردازشی بالا دارند، استفاده کنند.

نکته دیگری که باید به آن اشاره کرد این است که جهت پیاده سازی بهینه الگوریتم در سطح رجیستر، قبل از کد نویسی به زبان های توصیف سخت افزار مانند Verilog و VHDL، باید تمام بهینه سازی های ممکن، به اشتراک گذاری منابع سخت افزاری، تأخیر و همچنین نرخ پردازش در نظر گرفته شود. بهینه سازی در HLS اما رویکردی کاملاً متفاوت دارد. در این ابزار ابتدا الگوریتم به زبان C/C++ نوشته می شود و صحت کارایی آن تایید می گردد. سپس با استفاده از Directive و Pragma، معماری های مختلفی را می توان با تأخیر، نرخ پردازشی و منابع مصرفی متفاوت طراحی کرد و نهایتاً با ایجاد یک مصالحه بین تأخیر و نرخ پردازش مطلوب و همچنین منابع سخت افزاری قابل استفاده در تراشه، معماری مناسب را انتخاب نمود.

علاوه بر تسريع و سهولت در پروسه طراحی و تست، مزیت بسیار مهمی که سنتز سطح بالا نسبت به طراحی در سطح رجیستر دارد، انعطاف‌پذیری بسیار بالا در بهینه‌سازی و تغییر ساختار معماری یک طرح می‌باشد. چنانچه خواسته‌های یک طرح همانند نرخ پردازش اطلاعات ورودی و خروجی تغییر یابد، در HLS با اعمال فرامین کنترلی پرآگما و تنها با کنترل کامپایلر بدون احتمالاً حتی یک خط تغییر در کد C/C++ در زمان کوتاهی می‌توان به خواسته مطلوب دست یافت. اما در طراحی سطح رجیستر، نیاز به باز طراحی معماری طرح می‌باشد که این امر پرسه زمان بربی است. با نگاه به شکل زیر به راحتی می‌توانید چیدمان هرم توسعه محصول در HLS را با HDL مقایسه کنید. این هرم کاملاً وارونه است و بعد از طراحی IP فرایند تجمیع و تست به شکل خیره کننده‌ای سریع‌تر است.



شکل شماره ۱۴. ساختار سلسله مراتبی HLS

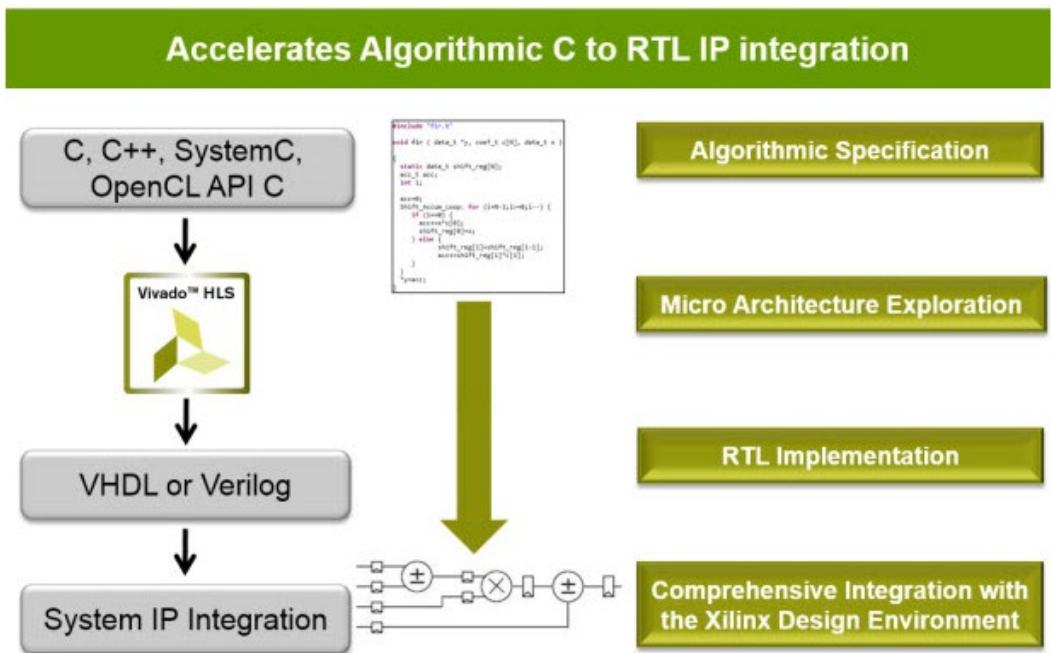
ابزار Vivado HLS به عنوان یک بروزرسانی برای محیط توسعه Vivado HLx ارائه شده است. این ابزار با هدف افزایش سرعت پیاده‌سازی IP ها، بلوک‌های شتاب دهنده و الگوریتم‌های پردازشی در FPGA با استفاده از C و C++ و System C عرضه شده است. برخی از مهم‌ترین ویژگی‌های این ابزار به شرح زیر است:

- سنتز هوشمند کدهای C با توجه به معماری‌های مختلف تراشه‌های FPGA
- تسريع فرایند شبیه‌سازی RTL با تبدیل اتوماتیک تست بنج‌های C

- پشتیبانی از اینترفیس‌های استاندارد صنعتی همچون AXI4-Stream و AXI4
- پشتیبانی از کتابخانه‌های arbitrary precision برای تعریف دیتا تایپ‌های کاملاً سفارشی
- پشتیبانی از دیتا تایپ‌های ممیز ثابت و ممیز شناور
- پشتیبانی از زبان‌های C و C++ و System C
- پشتیبانی از حافظه‌های بلوکی block RAM، بلوک‌های ضرب کننده DSP و سایر عناصر طراحی

Vivado HLS به طراحان امکان می‌دهد که به راحتی و با سرعت بالا به پیاده‌سازی الگوریتم‌های پیچیده برای برنامه‌های کاربردی دیجیتالی بپردازند. با استفاده از این ابزار، طراحان می‌توانند زمان پیاده‌سازی را به شدت کاهش داده و به طور کلی فرآیند طراحی را سریعتر و کارآمدتر کنند. بر اساس آخرین بروز رسانی سایت Xilinx لیست کتابخانه‌های عرضه شده همراه با Vivado HLS به شرح زیر است.

- کتابخانه‌های تایپ‌های سفارشی و محاسبات ممیز ثابت (Arbitrary Precision Data Type)
- کتابخانه پردازش داده‌ها به صورت استریم (HLS Stream)
- کتابخانه‌های توابع ریاضی (HLS Math)
- کتابخانه‌های پردازش تصویر (HLS Video)
- کتابخانه‌های هسته‌های نرم افزاری آماده (HLS IP)
- کتابخانه‌های توابع جبری و ماتریسی (HLS Linear Algebra)
- کتابخانه‌های پردازش سیگنال دیجیتال (HLS DSP)



شکل شماره ۱۵. نحوه تبدیل کد C به RTL در ابزار HLS

الگوریتم Sobel

الگوریتم سوبل (Sobel) یک الگوریتم پردازش تصویر است که برای شناسایی لبه‌ها در تصاویر دیجیتال استفاده می‌شود. این الگوریتم به نام مخترع خود، اروین سوبل، نامگذاری شده است. عملکرد الگوریتم سوبل بر اساس محاسبه گرادیان شدت تصویر در هر پیکسل است. این الگوریتم تقریبی از مشتق اول شدت تصویر را در جهت‌های افقی و عمودی محاسبه می‌کند. برای این منظور، تصویر با مجموعه‌ای از کرنل‌های کوچک و قابل جداسازی جمع شده است. الگوریتم سوبل شامل دو کرنل جداگانه است: یکی برای تشخیص لبه‌های افقی و دیگری برای تشخیص لبه‌های عمودی. این کرنل‌ها ماتریس‌های ۳ در ۳ هستند که با تصویر جمع شده و نتیجه عمل جمع، تخمینی از مشتق تصویر در جهت مورد نظر را ارائه می‌دهد.

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

$$G = \sqrt{G_x^2 + G_y^2} \quad \rightarrow \quad G \approx |G_x| + |G_y|$$

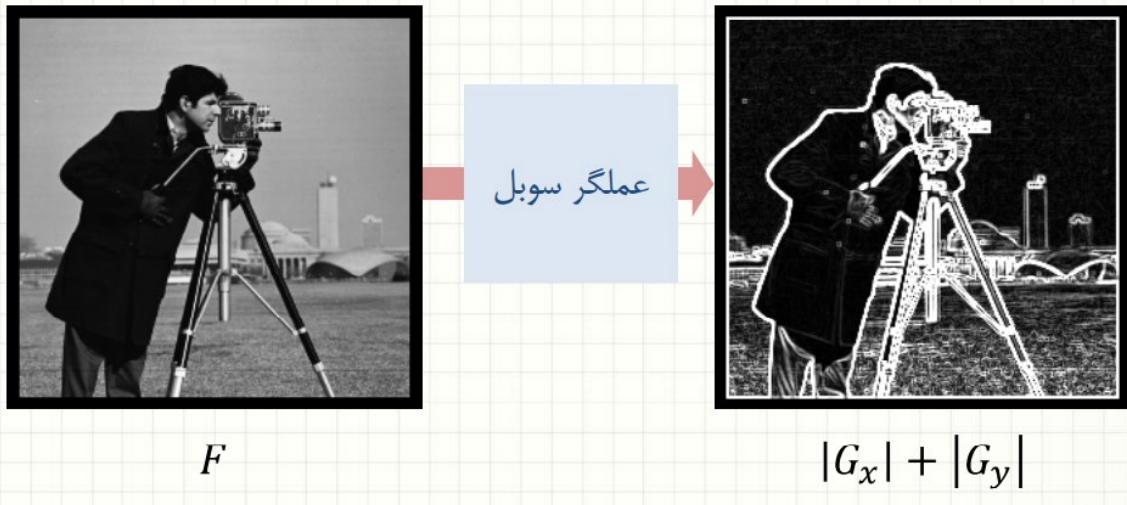
در ادامه الگوریتم سوبل، پس از جمع هسته‌های افقی و عمودی با تصویر، مازول گرادیان محاسبه می‌شود. برای محاسبه مازول گرادیان، ابتدا گرادیان افقی و عمودی در هر پیکسل محاسبه می‌شوند. سپس، مازول گرادیان با استفاده از این دو مقدار محاسبه می‌شود. مازول گرادیان نشان می‌دهد که در هر نقطه از تصویر، شدت و جهت تغییرات سرعت تصویر چقدر است. اگر مازول گرادیان بزرگ باشد، نشان‌دهنده وجود لبه قوی است. بنابراین، با استفاده از این الگوریتم، می‌توان لبه‌های تصویر را تشخیص داد و برجسته‌سازی کرد. استفاده از الگوریتم سوبل در بسیاری از برنامه‌های پردازش تصویر مفید است، از جمله تشخیص لبه، تشخیص شیء، تشخیص تغییرات شدت نور و بسیاری دیگر. الگوریتم سوبل یکی از الگوریتم‌های پرکاربرد در حوزه پردازش تصویر است که در بسیاری از نرم‌افزارها و کتابخانه‌های پردازش تصویر موجود می‌باشد.

مزیت

مزیت الگوریتم سوبل این است که عملکرد سریع و ساده‌ای دارد و نتایج قابل قبولی را در تشخیص لبه‌ها ارائه می‌دهد. با این حال، الگوریتم سوبل به تنها‌ی قادر به تشخیص تمامی انواع لبه‌ها نیست و ممکن است در برخی موارد لبه‌های ناخواسته را نیز تشخیص دهد. برای استفاده از الگوریتم سوبل، ابتدا تصویر ورودی را به سطح خاکستری تبدیل می‌کنیم. سپس کرنل‌های سوبل را با تصویر جمع کرده و مازول گرادیان را محاسبه می‌کنیم. در نهایت، با استفاده از یک آستانه مشخص، لبه‌های تشخیص داده شده را استخراج می‌کنیم. استفاده از الگوریتم سوبل نیازمند بررسی و تنظیم آستانه مناسب است. آستانه میزان شدت لبه‌های تشخیص داده شده را تعیین می‌کند. اگر آستانه بیشتر باشد، تنها لبه‌های قوی‌تر تشخیص داده می‌شوند و اگر آستانه کمتر باشد، لبه‌های ضعیف‌تر نیز تشخیص داده می‌شوند. در کل، الگوریتم سوبل یک روش مؤثر برای تشخیص لبه‌ها در تصاویر است و در بسیاری از برنامه‌های پردازش تصویر و بینایی ماشین مورد استفاده قرار می‌گیرد.

با استفاده از الگوریتم سوبل، می‌توانیم انواعی از عملیات پردازش تصویر را انجام دهیم، از جمله تشخیص لبه‌ها، تقویت و ضعیف‌سازی لبه‌ها، استخراج خطوط و مرزها، تشخیص شیء، تشخیص تغییرات شدت نور و بسیاری دیگر. به طور کلی، الگوریتم سوبل در مراحل پیش‌پردازش تصویر برای بهبود نتایج

الگوریتم‌های بعدی استفاده می‌شود. به عنوان مثال، در الگوریتم‌های تشخیص و شناسایی اشیاء، استفاده از الگوریتم سوبل به عنوان یک مرحله میانی معمولاً منجر به بهبود نتایج نهایی می‌شود. به طور خلاصه، الگوریتم سوبل یک روش ساده و قابل فهم است که برای تشخیص لبه‌ها در تصاویر استفاده می‌شود. با ترکیب الگوریتم سوبل با سایر روش‌ها و الگوریتم‌های پردازش تصویر، می‌توان نتایج بهتری در بسیاری از برنامه‌های بینایی ماشین و پردازش تصویر به دست آورد. با استفاده از الگوریتم سوبل، می‌توانیم همزمان لبه‌های افقی و عمودی را در تصویر تشخیص دهیم. این الگوریتم به دلیل سرعت و سادگی اجرا، به خصوص در برنامه‌های real time مورد استفاده قرار می‌گیرد.



شكل شماره ۱۶. الگوریتم sobel

الگوریتم Canny

الگوریتم کنی (Canny) یکی از الگوریتم‌های معروف در حوزه پردازش تصویر است که برای تشخیص و استخراج لبه‌ها استفاده می‌شود. الگوریتم کنی توسط John F. Canny در سال ۱۹۸۶ معرفی شد و به دلیل دقیق بالا و کارایی خوب در تشخیص لبه‌ها، یکی از روش‌های پراستفاده در پردازش تصویر محسوب می‌شود. عملکرد الگوریتم کنی از چند مرحله تشکیل شده است. در ابتدا، تصویر ورودی به سطح خاکستری تبدیل می‌شود. سپس، مرحله اصلی الگوریتم شامل گام‌های زیر است:

- کاهش نویز (Noise Reduction): با استفاده از فیلتر گوسی، نویزهای موجود در تصویر را کاهش می‌دهیم تا تأثیرات آن در تشخیص لبه‌ها کاهش یابد.
- تشخیص شدت گرادیان (Gradient Intensity Detection): با محاسبه مشتقات جزئی تصویر در جهت‌های افقی و عمودی، شدت گرادیان در هر نقطه را محاسبه می‌کنیم.
- تشخیص جهت گرادیان (Gradient Direction Detection): جهت گرادیان در هر نقطه را محاسبه می‌کنیم تا جهت لبه‌ها را تعیین کنیم.
- استفاده از آستانه (Thresholding): با استفاده از آستانه‌بندی، لبه‌های قوی و ضعیف را تفکیک می‌کنیم و لبه‌های مهم را تشخیص می‌دهیم.
- اتصال لبه‌ها (Edge Linking): لبه‌های قوی را به یکدیگر متصل می‌کنیم تا لبه‌های یکپارچه‌تری به دست آوریم.

الگوریتم کنی به دلیل قابلیت کنترل پارامترها و تولید نتایج دقیق، در بسیاری از برنامه‌ها و الگوریتم‌های پردازش تصویر مورد استفاده قرار می‌گیرد. با استفاده از الگوریتم کنی، می‌توانیم لبه‌های تصویر را به صورت دقیق و با حفظ جزئیات مهم تشخیص دهیم. الگوریتم کنی برای تشخیص لبه‌های ناهموار و با شدت‌های مختلف بسیار کارآمد است.

• یکی از ویژگی‌های برجسته الگوریتم کنی، قدرت انتخاب آستانه است. با تنظیم آستانه‌های مناسب، می‌توانیم تعداد لبه‌های تشخیص داده شده را کنترل کنیم و نتایج مطلوب را به دست آوریم. استفاده از الگوریتم کنی به طور گسترده در بسیاری از برنامه‌های پردازش تصویر مانند تشخیص اشیاء، تشخیص چهره، رایانش بینایی، تشخیص تغییرات در تصاویر و بسیاری موارد دیگر مورد استفاده قرار می‌گیرد. همچنین، الگوریتم کنی قابلیت کار با تصاویر با رزولوشن‌های مختلف را دارد و در برنامه‌های real time نیز به خوبی عمل می‌کند.

رابطه زیر، یک مثال از یک فیلتر گوسی 5×5 است که برای ایجاد تصویر زیر استفاده شده است. در ک این امر که انتخاب اندازه کرنل گوسی روی عملکرد آشکارساز اثر می‌گذارد مهم است. هر چه اندازه بیشتر باشد، آشکارساز حساسیت کمتری نسبت به نویز دارد. به علاوه خطای محلی سازی برای آشکار کردن لبه با افزایش اندازه کرنل فیلتر گوسی اندکی افزایش پیدا می‌کند. اندازه 5×5 در بیشتر موارد مناسب است اما می‌تواند بسته به شرایط خاص متفاوت باشد.

$$B = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * A$$

به طور خلاصه، الگوریتم کنی یک روش پرکاربرد و قدرتمند برای تشخیص و استخراج لبه‌ها در تصاویر است. این الگوریتم با دقت و کارایی بالا، تنوع در تنظیم پارامترها و قابلیت استفاده در بسیاری از برنامه‌های پردازش تصویر مورد توجه قرار می‌گیرد.



شكل شماره ۱۷. الگوریتم canny

الگوریتم Roberts

الگوریتم Roberts یکی از الگوریتم‌های پردازش تصویر است که برای تشخیص لبه‌ها استفاده می‌شود. این الگوریتم بر پایه تفاوت‌های شدت نوری در پیکسل‌ها عمل می‌کند. الگوریتم Roberts از دو فیلتر کوچک استفاده می‌کند. این فیلترها به شکل ماتریسی دو در دو هستند و معمولاً به صورت زیر تعریف می‌شوند:

$$G_x = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad G_y = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

در اینجا G_x فیلتر Roberts در جهت افقی و G_y فیلتر Roberts در جهت عمودی است. الگوریتم Roberts با استفاده از این دو فیلتر، تفاوت شدت نوری بین پیکسل‌های همسایه را محاسبه می‌کند. سپس مقدار مطلق این تفاوت را به عنوان شدت لبه در آن نقطه مشخص می‌کند. با استفاده از الگوریتم Roberts می‌توان لبه‌های تصویر را به طور دقیق تشخیص داد. این الگوریتم به خاطر سادگی و سرعت اجرا، در بسیاری از برنامه‌ها و سیستم‌های پردازش تصویر استفاده می‌شود.

الگوریتم Roberts با استفاده از فیلترهای Roberts به دو مرحله تقسیم می‌شود. در مرحله اول، فیلتر G_x به تصویر ورودی اعمال می‌شود و مقدار شدت لبه در جهت افقی در هر پیکسل محاسبه می‌شود. سپس در مرحله دوم، فیلتر G_y به تصویر ورودی اعمال می‌شود و مقدار شدت لبه در جهت عمودی در هر پیکسل محاسبه می‌شود. با جمع مقادیر مطلق لبه‌ها در هر پیکسل (یعنی جمع مقادیر شدت لبه‌های افقی و عمودی)، می‌توان مجموع شدت لبه کلی تصویر را محاسبه کرد. این مقدار مجموع شدت لبه به عنوان نتیجه نهایی الگوریتم Roberts استفاده می‌شود.

الگوریتم Roberts به خاطر سرعت بالا و قدرت تشخیص مناسب، به عنوان یک روش ساده و مؤثر برای تشخیص لبه‌ها در تصاویر استفاده می‌شود. این الگوریتم معمولاً در برنامه‌هایی که نیاز به تشخیص و استخراج لبه‌ها از تصاویر دارند، مانند تشخیص اشیاء، تصویربرداری پزشکی و پردازش تصویر صنعتی، به کار می‌رود. الگوریتم Roberts در عمل به صورت زیر عمل می‌کند:

- تصویر ورودی را به تصویر خاکستری تبدیل کنید اگر در حالت RGB باشد.

- برای هر پیکسل در تصویر، فیلتر G_x را به آن اعمال کنید و مقدار شدت لبه در جهت افقی را محاسبه کنید.
- برای هر پیکسل در تصویر، فیلتر G_y را به آن اعمال کنید و مقدار شدت لبه در جهت عمودی را محاسبه کنید.
- مقادیر مطلق شدت لبه‌های افقی و عمودی را در هر پیکسل با یکدیگر جمع کنید. مقدار حاصل، مجموع شدت لبه کلی تصویر است.

نتیجه نهایی الگوریتم Roberts، تصویری است که نقاط لبه در آن بیشتر روشن و سایر نقاط تیره می‌باشند. این تصویر می‌تواند به عنوان ورودی برای مراحل بعدی پردازش تصویر مورد استفاده قرار گیرد، مانند تشخیص شیء، استخراج ویژگی‌ها و غیره. با استفاده از الگوریتم Roberts، می‌توان به طور مؤثر لبه‌های تصویر را تشخیص داد و استفاده از آن در برنامه‌ها و سیستم‌هایی که به پردازش تصویر نیاز دارند، مفید است. این الگوریتم به نویز حساسیت زیادی دارد و پیکسل‌های کمتری را برای تقریب گرادیان بکار می‌برد، در ضمن نسبت به الگوریتم Canny هم قدرت کمتری دارد.



شكل شماره ۱۸. الگوریتم Roberts

الگوریتم Roberts ممکن است در برخی موارد با مشکلاتی مواجه شود. به عنوان مثال، این الگوریتم حساسیت بالایی نسبت به نویز دارد. در صورت وجود نویز در تصویر ورودی، ممکن است لبه‌های تشخیص

داده شده توسط الگوریتم ناپایدار و نامطلوب باشند. علاوه بر این، الگوریتم Roberts نسبت به تغییرات شدت نوری نسبتاً سریع و ناگهانی در تصویر حساس است. این به این معنی است که اگر شدت نور در یک منطقه‌ی تصویر به طور ناگهانی تغییر کند، الگوریتم ممکن است لبه‌های غیرمنطقی و نادرست را تشخیص دهد. بنابراین، در برخی موارد، استفاده از الگوریتم‌های پیشرفته‌تری مانند الگوریتم‌های مبتنی بر فیلتر سوبل یا لاپلاسین استوانه‌ای می‌تواند بهترین راه حل باشد. این الگوریتم‌ها اغلب دقیق‌تری در تشخیص لبه‌ها و مقاومت به نویز دارند. در نهایت، انتخاب الگوریتم مناسب برای تشخیص لبه‌ها بستگی به نیازها و محدودیت‌های برنامه و سیستم مورد استفاده دارد.

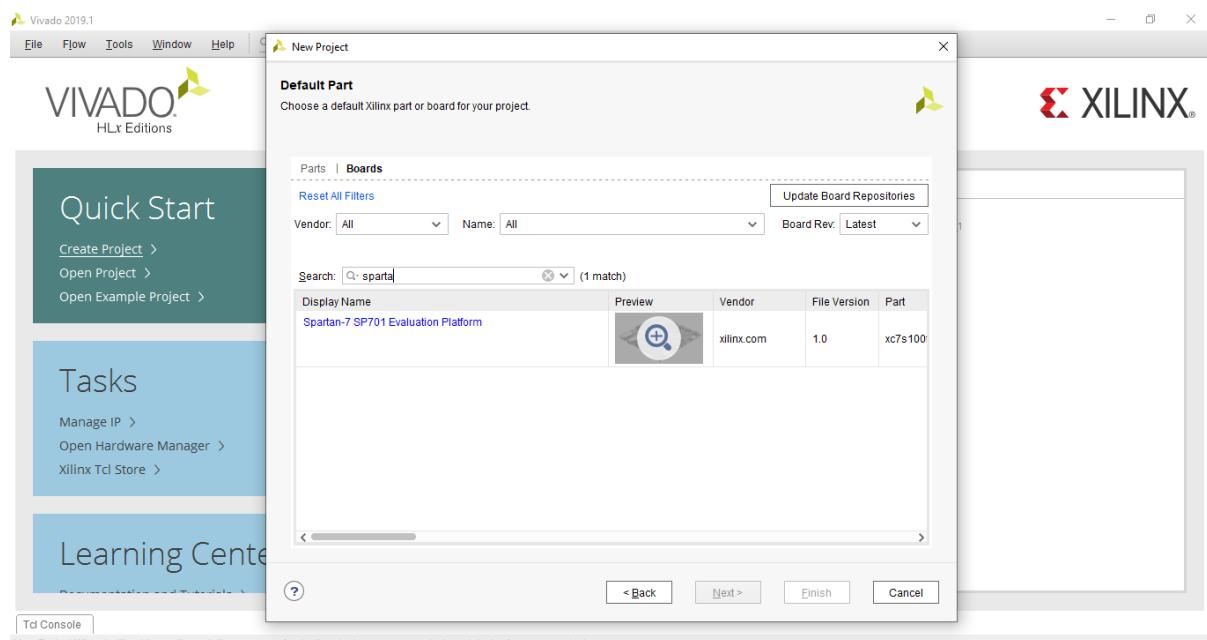
بخش چهارم: نحوه پیاده‌سازی تشخیص لبه

این مقاله از دو بخش اصلی تشکیل شده بود، بخش اول پیاده‌سازی مژول تشخیص لبه و بخش دوم پیاده‌سازی مژولی که فضای رنگی RGB را به فضای رنگی YCbCr با استفاده از روش حافظه‌سازی، تبدیل می‌کند. در واقع این مقاله در ابتدا با ارجاع به مقاله [1] ادعا می‌کند که روش حافظه‌سازی نسبت به روش‌هایی مثل UDM و کاهش عرض باس، توان دینامیکی بیشتری را کاهش می‌دهد، سپس دو نمونه عملی یا study case تشخیص لبه و تبدیل فضای رنگی را با FPGA و با استفاده از روش حافظه‌سازی پیاده می‌کند و برای اثربخشی بیشتر، با درنظر گرفتن یک حد آستانه، برای تمام ورودی‌هایی که در بازه مشخص شده قرار می‌گیرند، مقادیر حفظ شده از قبل را به عنوان خروجی بر می‌گردانند و اینگونه روش محاسبات تقریبی را نیز به کار خود اضافه می‌کنند.

ما برای پیاده‌سازی این مقاله، سعی کردیم تمام مراحل پیاده‌سازی را مطابق موارد ذکر شده در مقاله انجام بدیم و از همان ابزار و فرض‌هایی که مقاله به از آن‌ها استفاده کرده، استفاده کنیم. البته این مقاله برای ده سال گذشته بوده و خیلی از ابزارها با تغییرات زیادی طی این سال‌ها داشتند، همچنان این مقاله خروجی خود را براساس پیاده‌سازی واقعی روی برد بدست آورده بود که به دلیل محدودیت و عدم دسترسی به برد، ما کار خود را با شبیه‌سازی انجام دادیم، به همین دلیل ممکن است در بحث گزارش توان و غیره، مقادیر بدست آمده توسط ما با مقادیر آورده شده در مقاله یکسان نباشد، هر چند نتیجه‌گیری نهایی ما کاملاً ایده‌ی مطرح شده در مقاله را تایید می‌کند. البته لازم به ذکر است امکان پیاده‌سازی این روش‌ها بر روی بردۀای CortexM وجود داشت، ولی تصمیم بر این شد که برای این پروژه، در پلتفرم FPGA باقی بمانیم و به همان شبیه‌سازی اکتفا کنیم.

ابزارهایی که ما برای این پروژه استفاده کردیم، نرمافزارهای Vivado HLS 2019.1 و Vivado 2019.1 می‌باشد. از نرمافزار Vivado HLS برای سنتز برنامه‌های نوشته شده با زبان‌های سطح بالا مثل C/C++ به کد verilog و استخراج IP CORE استفاده کردیم، همچنان از نرمافزار Vivado 2019.1 برای شبیه‌سازی VHDL

لازم به ذکر است در تمام موارد بالا به چالش‌های زیادی برخود کردیم که در بخش بعدی به آن‌ها اشاره خواهیم کرد. پلتفرم استفاده شده در این پروژه، برد Spartan-7 SP701 می‌باشد که از تراشه xc7s100fgga676-2 استفاده می‌کند. دلیل انتخاب این پلتفرم، بحث لایسنس نرم‌افزار HLS بود؛ این نرم‌افزار برای خیلی از پلتفرم‌ها محدود بوده و مشکل لایسنس دارد، بنابراین لازم بود از یک پلتفرم مجاز استفاده کنیم که پیشنهاد مطرح شده در ویدیوهای آموزشی همین بود.



شکل شماره ۱۹. پلتفرم هدف استفاده شده

ما در ابتداء نحوه پیاده‌سازی ماژول تشخیص لبه با زبان سطح بالا مثل C/C++ و کدهای نوشته شده برای آن را شرح خواهیم داد، سپس نحوه استخراج کد وریلگ و IP CORE ماژول نوشته را خواهیم گفت و در ادامه آن را با استفاده از نرم‌افزار Model Sim شبیه‌سازی کرده و با استفاده از TCL console در این نرم‌افزار، فایل saif را استخراج می‌کنیم. کار کرد این فایل برای بحث تخمین توان در نرم‌افزار Vivado می‌باشد. بطور پیشفرض در این نرم‌افزار برای تخمین توان، به هر کدام از سیگنال‌های ورودی با نرخ سوئیچینگ مشخص، یک مقدار تصادفی داده و براساس آن توان تخمین زده می‌شود. این روش مناسب کار ما نیست زیرا در این پروژه ورودی‌های ماژول‌ها بسیار مهم بوده و منطق محاسبه خروجی و در نتیجه توان لازم برای محاسبه آن را تعیین

می‌کند، بنابراین ما لازم داریم تا سناریوی ورودی‌ها مطابق شبیه‌سازی‌ها باشد و توسط خود ما تعیین شود، برای این منظور می‌توان با تولید فایل saif و دادن آن به نرم‌افزار vivado، سناریوی تعیین ورودی‌ها را مشخص کرد و توان دینامیکی را به ازای ورودی‌های مشخص بدست آورد.

توضیح کد مازول تشخیص لبه با زبان C

ما برای پیاده‌سازی مازول تشخیص لبه، ابتدا الگوریتم آن را با زبان سطح بالای C نوشتیم و سپس با استفاده از نرافزار HLS، کد VHDL آن را استخراج کردیم. ما از الگوریتم Sobel که در بخش قبلی بطور کامل آن را شرح دادیم، برای تشخیص لبه استفاده کردیم. در ادامه کد نوشته شده برای این الگوریتم را بطور کامل توضیح خواهیم داد.

```

1 #include "sobel.h"
2
3 void sobel_filter(unsigned char in[512*512], unsigned char out[512*512]) {
4     int gx_sum = 0;
5     int gy_sum = 0;
6     for(int i = 1; i < 512-1; i++){
7         for(int j = 1; j < 512-1; j++){
8             gx_sum = -in[(i-1)*512+(j-1)] - 2*in[(i)*512+(j-1)] - in[(i+1)*512+(j-1)] + in[(i-1)*512+(j+1)] + 2*in[(i)*512+(j+1)] + in[(i+1)*512+(j+1)];
9             gy_sum = in[(i-1)*512+(j-1)] + 2*in[(i-1)*512+(j)] + in[(i-1)*512+(j+1)] - in[(i+1)*512+(j-1)] - 2*in[(i+1)*512+(j)] + in[(i+1)*512+(j+1)];
10            gx_sum = (gx_sum < 0) ? -gx_sum : gx_sum;
11            gy_sum = (gy_sum < 0) ? -gy_sum : gy_sum;
12            out[i*512+j] = gx_sum + gy_sum;
13            out[i*512+j] = (out[i*512+j] > 255) ? 255 : out[i*512+j];
14        }
15    }
16 }
```

شکل شماره ۲۰. تابع sobel_filter()

این کد یک فیلتر Sobel را بر روی تصویر اعمال می‌کند. این روش بر اساس مشتق‌گیری تصویر است و در دو جهت افقی و عمودی اعمال می‌شود. تابع sobel_filter دو آرایه ورودی و خروجی به نام‌های in و out دریافت می‌کند که با استفاده از آن‌ها تصویر ورودی را اعمال فیلتر Sobel کرده و در تصویر خروجی قرار می‌دهد. دو حلقه‌ی تو در تو در این تابع بر روی تمام پیکسل‌های تصویر اعمال می‌شوند. در هر مرحله، مقادیر gy_sum و gx_sum محاسبه می‌شوند که جمع مقادیر مشتق‌ها در جهت‌های افقی و عمودی هستند. سپس از این مقادیر، مقدار مطلق گرادیان لبه برای هر پیکسل محاسبه می‌شود. در نهایت، مقدار نهایی برای هر پیکسل در آرایه خروجی out ذخیره می‌شود. در تکه کد آخر، مقادیر gy_sum و gx_sum جمع می‌شوند و

مقدار مطلق آنها را محاسبه می‌کنیم. سپس مقدار نهایی برای آن پیکسل در out قرار می‌گیرد. اگر مقدار نهایی بیشتر از ۲۵۵ باشد، به جای آن مقدار ۲۵۵ قرار داده می‌شود.

```

1  #include "stdio.h"
2  #include "testBench512.h"
3  #include "sobel.c"
4
5  int main()
6  {
7      FILE *fp;
8      int retval=0;
9      unsigned char out1[512*512];
10     sobel_filter(elaine_512_input, out1);
11
12     fp = fopen("result.txt", "w");
13     for(int i = 0; i < 512; i++){
14         for(int j = 0; j < 512; j++){
15             fprintf(fp, "%d ", out1[i*512+j]);
16         }
17         fprintf(fp, "\n");
18     }
19     fclose(fp);
20     return 0;
21 }
```

شکل شماره ۲۱. برنامه Test bench

این کد یک برنامه ساده‌ی C است که فیلتر Sobel را بر روی تصویر ورودی اعمال می‌کند و نتیجه را در یک فایل متنه با نام "result.txt" ذخیره می‌کند. در ابتدا، کتابخانه‌های stdio.h و testBench512.h را وارد می‌کنیم، که شامل توابع استاندارد و تعریف‌هایی برای ورودی و خروجی است. همچنین، فایل sobel.c را وارد می‌کنیم که تابع sobel_filter را پیاده‌سازی کرده است. سپس، تابع main را تعریف می‌کنیم که از درآرایه elaine_512_input تصویر استفاده می‌کند و فیلتر Sobel را بر روی آن اعمال می‌کند. نتیجه را در آرایه out1 ذخیره می‌کنیم. سپس، فایل "result.txt" را برای نوشتن باز می‌کنیم (fopen("result.txt", "w")) و با استفاده از حلقه‌های تو در تو، مقادیر ذخیره شده در آرایه out1 را به صورت فضای سفید جداکننده جداکننده می‌کنیم و به فایل نوشته می‌شود (fprintf(fp, "%d ", out1[i*512+j])). در انتهای فایل را می‌بندیم (fclose(fp)). در نهایت، برنامه با برگشت مقدار ۰ پایان می‌باید (return 0). شکل شماره ۲۲ خروجی این برنامه را نشان می‌دهد.



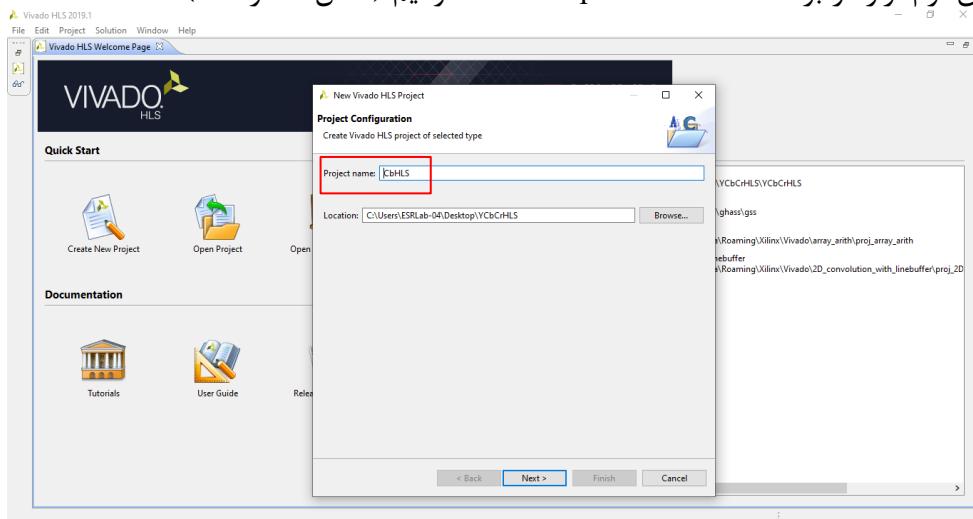
شکل شماره ۲۲. خروجی برنامه تشخیص لبه نوشته شده با الگوریتم Sobel

ساخت پروژه با ابزار HLS

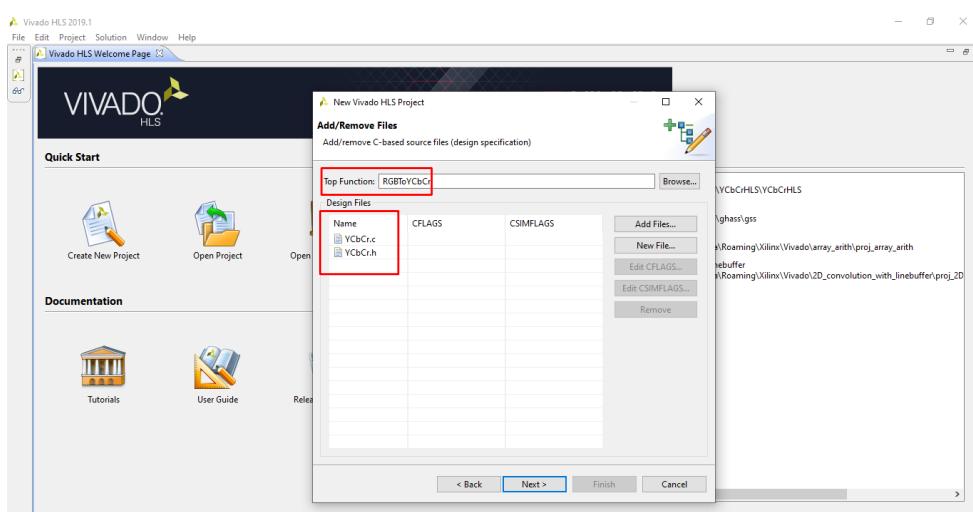
بعد از اطمینان از صحت عملکرد برنامه نوشته به زبان C، آن را به نرمافزار Vivado HLS داده تا کد آن را ساخته و IP CORE آن را جهت استفاده استخراج کنیم. در ادامه مراحل ایجاد پروژه، سنترز RTL، استخراج IP CORE و بررسی صحت و درستی خروجی ساخته شده توسط نرمافزار HLS را خواهیم گفت. لازم به ذکر است ما از نسخه 2019.1 این نرمافزار استفاده کرده‌ایم. اولین قدم، ساخت پروژه جدید می‌باشد، نکته مهم این قسمت مربوط به نام‌گذاری پروژه می‌باشد، نام پروژه نباید شامل عدد باشد و بهتر است که فقط با حروف بزرگ و کوچک نوشته شده باشد (شکل شماره ۲۳). اضافه کردن فایل‌های کد مربوط به تابعی که می‌خواهیم سنترز کنیم می‌باشد، توجه کنید در این قسمت باید تمام فایل‌هایی که برای ساخت تابع اصلی لازم هستند، مثل زیر تابع‌ها و غیره را نیز اضافه کنیم، همچنین باید از قسمت Top Function، باید تابعی که قرار است سنترز شود را مستقیم مشخص کنیم (شکل شماره ۲۴). قدم بعدی اضافه کردن فایل‌هایی است که برای تست تابع اصلی استفاده می‌شوند، توجه کنید تابع main باید در این سورس کد قرار گرفته باشد و در این مرحله اضافه شده باشد، به این فایل‌ها، فایل‌های Testbench می‌گویند (شکل

شماره ۲۵). قدم بعدی، تعیین پلتفرم هدف می‌باشد که ما همانطور که ذکر کردیم، به دلیل مسائل مربوط به

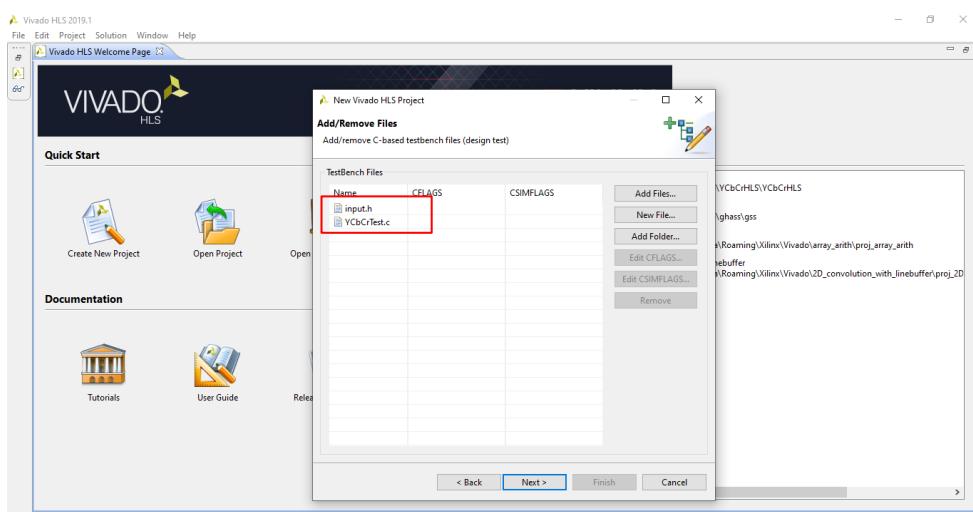
لایسنس این نرمافزار، از برد Spartan-7 SP701 استفاده کردیم (شکل شماره ۲۶).



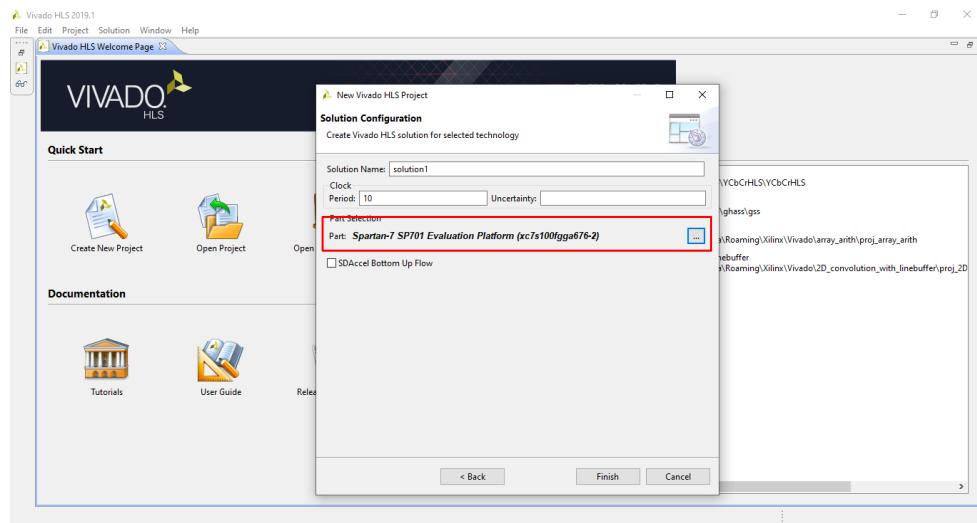
شکل شماره ۲۳. نام‌گذاری پروژه



شکل شماره ۲۴. اضافه کردن فایل‌های تابع اصلی



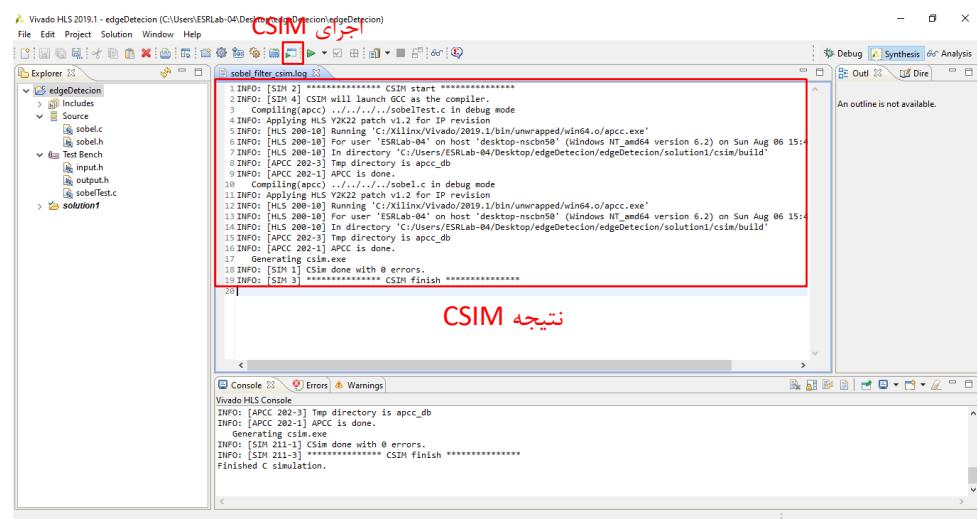
شکل شماره ۲۵. اضافه کردن فایل‌های مربوط به تست تابع اصلی



شکل شماره ۲۶. تعیین پلتفرم هدف

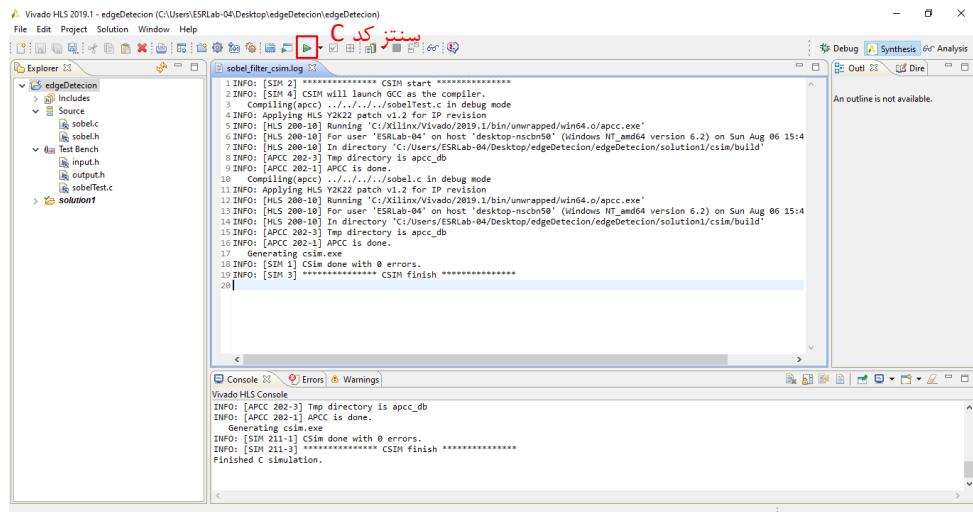
سنتز کد VHDL با استفاده از کد C با ابزار HLS

ساخت خروجی با نرم افزار HLS، همانند کامپایل کردن یک برنامه، مراحل مختلفی دارد که در ادامه به شرح این مراحل می پردازیم. مرحله اول شبیه سازی کد C پروژه می باشد که از این مرحله با نام CSIM یاد می شود. برای این مرحله لازم است تا آیکون مشخص شده در شکل شماره ۲۷ را فشار دهیم، سپس خروجی ها و option هایی که لازم داریم را مشخص کنیم و در نهایت صبر کنیم تا شبیه سازی به اتمام برسد، شکل شماره ۲۷ نتیجه شبیه سازی را نشان می دهد. یکی از نکات مثبت نرم افزار HLS، قرار دادن دیباگر در آن می باشد که باعث می شود بتوانیم در همان محیط HLS به رفع ایرادات برنامه بپردازیم.

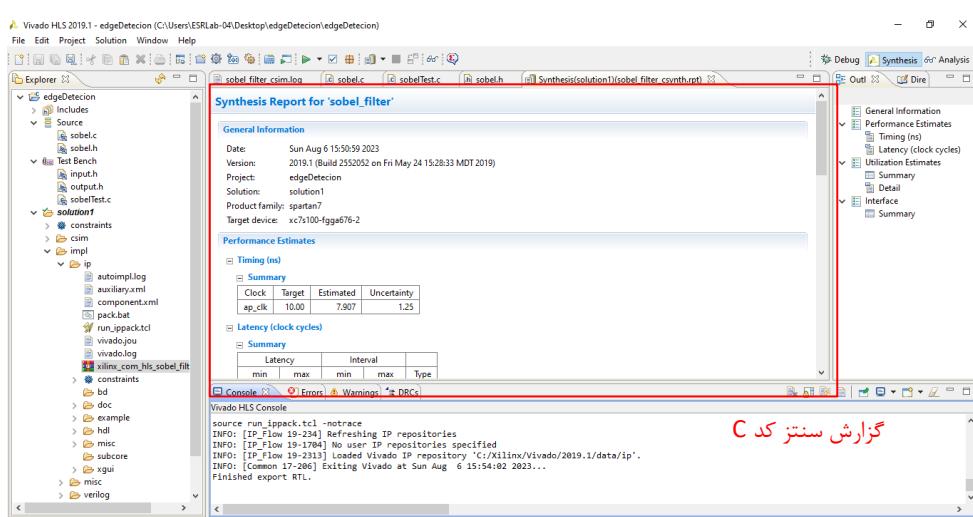


شکل شماره ۲۷. اجرای CSIM

مرحله‌ی دوم، تبدیل کد C و سنتز آن به RTL می‌باشد. تمام مراحل بدست آوردن خروجی در نرمافزار HLS دارای چالش‌های زیادی بودند اما این مرحله به نسبت چالش‌های بیشتری داشت که در بخش بعدی به آن خواهیم پرداخت. برای سنتز RTL، باید از آیکون مشخص شده در شکل شماره ۲۸ استفاده کنیم، همچنین در صورت موفقیت آمیز بودن این مرحله، در نهایت یک گزارش شامل زمان‌بندی‌ها، منابع استفاده شده، پورت‌های ورودی و خروجی و غیره تولید خواهد شد (شکل شماره ۲۹) که امکان ذخیره‌ی آن بصورت فایل XML وجود دارد و ما آن را در قسمت نتایج مربوط به ماذول تشخیص لبه قرار داده‌ایم. خروجی نهایی این مرحله، سورس کد Verilog و VHDL می‌باشد.

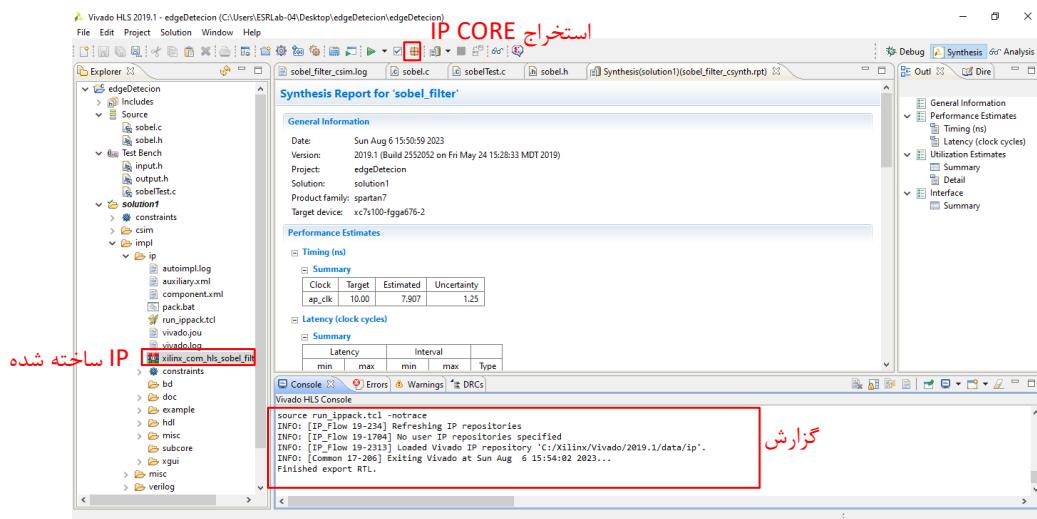


شکل شماره ۲۸. سنتز کد C



شکل شماره ۲۹. خروجی سنتز کد C

مرحله نهايی، استخراج IP CORE جهت استفاده مستقیم در نرمافزارهایي مثل Vivado و ISE میباشد. برای استخراج IP CORE، باید از آیکون مشخص شده در شکل شماره ۳۰ استفاده کنیم، همچنین درصورت موفقیتآمیز بودن اين مرحله، پیامی مطابق با شکل شماره ۳۰ نشان داده خواهد شد.



شکل شماره ۳۰. استخراج IP CORE

توضیح کد VHDL ماژول تشخیص لبه

ما برای اعمال واحد حافظه به ماژول تشخیص لبه و همچنین مدیریت پورت‌های آن مطابق نیازمان، تصمیم گرفتیم که از IP تولید شده بصورت مستقیم استفاده نکنیم و از کد VHDL سنتر شده توسط برنامه HLS استفاده کرده و تغییرات خود را مستقیم بر روی آن اعمال کنیم.

```

22 library IEEE;
23 use IEEE.STD_LOGIC_1164.ALL;
24 use IEEE.numeric_std.all;
25 -- library UNISIM;
26 -- use UNISIM.VComponents.all;
27
28 entity memoization is
29 port (
30     clk      : in std_logic;
31     rst      : in std_logic;
32     start    : in std_logic;
33     input    : in std_logic_vector(7 downto 0);
34     row      : in std_logic_vector(7 downto 0);
35     column   : in std_logic_vector(7 downto 0);
36     trigger  : out std_logic;
37     output   : inout std_logic_vector(7 downto 0)
38 );
39 end memoization;

```

شکل شماره ۳۱. پورت‌های ماژول تشخیص لبه

این کد، تعریف یک موجودیت به نام "memoization" در زبان VHDL است. این موجودیت یک مازول سخت‌افزاری را توصیف می‌کند که قابلیت ذخیره‌سازی نتایج محاسبات یک تابع را دارد تا درخواست‌های آینده برای نتایج مشابه، مقدار ذخیره شده را به جای محاسبه مجدد ارائه دهد. ویژگی‌های ورودی و خروجی موجودیت عبارتند از:

- **:clk**: ورودی کلک که برای هماهنگی مازول با سیستم استفاده می‌شود.
- **:rst**: ورودی تنظیم مجدد (Reset) که در صورت فعال شدن، مازول به حالت اولیه بازمی‌گردد.
- **:start**: ورودی شروع که با فعال شدن آن، مازول آماده دریافت و پردازش ورودی جدید می‌شود.
- **:input**: ورودی که شامل ۸ بیت است و مقدار مورد نظر برای محاسبه و ذخیره در مازول است.
- **:row**: ورودی برداری ردیف که برای نشان دادن ردیف مورد نظر در حافظه استفاده می‌شود.
- **:column**: ورودی برداری ستون که برای نشان دادن ستون مورد نظر در حافظه استفاده می‌شود.
- **:trigger**: خروجی تریگر که به عنوان نشانگری برای فعال شدن مازول در صورتی که مقدار محاسبه شده در حافظه با مقدار محاسبه شده قبلی متفاوت باشد، استفاده می‌شود.
- **:output**: خروجی برداری اطلاعات که شامل ۸ بیت است و مقدار ذخیره شده در حافظه برای ورودی مورد نظر را نشان می‌دهد. این خروجی به صورت دوطرفه تعریف شده است، بنابراین می‌توان اطلاعات را خواند و همچنین برای ذخیره اطلاعات جدید استفاده کرد.

```

41 architecture memoization_bch of memoization is
42   type ram_type is array (0 to 999) of std_logic_vector (7 downto 0); -- define 100000x8 bit RAM Type
43   type rom_type is array (0 to 255) of std_logic_vector (7 downto 0); -- define 100000x8 bit RAM Type
44   type state_type is (idle,start_rows,approximate,proces); -- State = idle , start_rows , proces
45   signal state_reg , state_next : state_type; -- state register
46   signal in_ram : ram_type; -- input Ram for save inputs(4 rows)
47   signal data_in : std_logic_vector(7 downto 0); -- Input Port RAM_IN
48   signal data_in_mem : std_logic_vector(7 downto 0);
49   signal data_out : std_logic_vector(15 downto 0); -- Output Port RAM_IN Memoization
50   signal data_out_1 : std_logic_vector(7 downto 0) := in_ram(0); -- use for compute filter output . GET pixel value form input RAM
51   signal data_out_2 : std_logic_vector(7 downto 0) := in_ram(1); -- use for compute filter output . GET pixel value form input RAM
52   signal data_out_3 : std_logic_vector(7 downto 0) := in_ram(2); -- use for compute filter output . GET pixel value form input RAM
53   signal data_out_4 : std_logic_vector(7 downto 0) := in_ram(3); -- use for compute filter output . GET pixel value form input RAM
54   signal data_out_5 : std_logic_vector(7 downto 0) := in_ram(4); -- use for compute filter output . GET pixel value form input RAM
55   signal data_out_6 : std_logic_vector(7 downto 0) := in_ram(5); -- use for compute filter output . GET pixel value form input RAM
56   signal data_out_7 : std_logic_vector(7 downto 0) := in_ram(6); -- use for compute filter output . GET pixel value form input RAM
57   signal data_out_8 : std_logic_vector(7 downto 0) := in_ram(7); -- use for compute filter output . GET pixel value form input RAM
58   signal data_out_9 : std_logic_vector(7 downto 0) := in_ram(8); -- use for compute filter output . GET pixel value form input RAM
59   signal add_inram_reg , add_inram_next : std_logic_vector(15 downto 0) := (others => '0'); -- Address RAM_IN
60   signal diff_cntr_reg , diff_cntr_next : std_logic_vector(15 downto 0) := (others => '0');
61   signal column_cntr_reg , column_cntr_next : std_logic_vector(7 downto 0) := (others => '0');
62   signal threshold : std_logic_vector(15 downto 0) := "00000000000001010"; -- make constant 1 for multiplexer
63   signal writesignal_reg, writesignal_next : std_logic;
64   signal trigger_reg, trigger_next : std_logic;
65   signal last_ram_addr : std_logic_vector(15 downto 0) := std_logic_vector(to_unsigned(255,16));
66   signal one : std_logic_vector(7 downto 0) := "00000001"; -- make constant 1 for multiplexer
67   signal four : std_logic_vector(7 downto 0) := "00000100"; -- make constant 4 for multiplexer
68   signal temp4 , temp1 : std_logic_vector(15 downto 0); -- temporary signal

```

شکل شماره ۳۲. سیگنال‌های مازول تشخیص لبه

این کد، توصیفی از بخش معماری "memoization" برای موجودیت architecture در زبان VHDL است. در این بخش، اجزای داخلی مازول و رفتار آن تعریف می‌شوند. ابتدا، دو نوع حافظه با نام "ram_type" و "rom_type" تعریف شده‌اند. یک آرایه از بردارهای "std_logic_vector" با اندازه ۸ بیت است که برای ذخیره‌سازی داده‌ها در حافظه RAM استفاده می‌شود. همچنین، "rom_type" با اندازه ۸ بیت است که برای ذخیره‌سازی داده‌ها در حافظه خوانا ROM از بردارهای "std_logic_vector" با اندازه ۸ بیت است که برای ذخیره‌سازی داده‌ها در حافظه خوانا استفاده می‌شود. سپس، یک نوع داده با نام "state_type" تعریف شده است که از نوع شناسه‌ای "process" و "approximate" است و مقادیر ممکن برای آن شامل "idle"، "start_rows" و "idle" هستند. این نوع داده برای نشان دادن وضعیت state (state) فعلی و آینده مازول استفاده می‌شود. سیگنال‌های متعددی تعریف شده‌اند که بین اجزای داخلی مازول ارتباط برقرار می‌کنند. این سیگنال‌ها شامل موارد زیر هستند.

هستند:

- سیگنال‌هایی که وضعیت state فعلی و آینده مازول را نشان می‌دهند. state_next و state_reg
- حافظه RAM که برای ذخیره‌سازی ورودی‌ها استفاده می‌شود. in_ram
- ورودی برداری داده که مقدار ورودی جدید را نشان می‌دهد. data_in_mem و data_in
- خروجی‌های برداری داده که مقادیر مختلفی را از حافظه data_out_1 تا data_out_9 خوانده و استفاده می‌کنند. data_out
- سیگنال‌های آدرس حافظه RAM ورودی. add_inram_nex و add_inram_reg
- سیگنال‌های شمارنده برای محاسبه تفاوت بین داده‌های ورودی و حفظ شده. diff_cntr_next و diff_cntr_reg
- سیگنال‌های شمارنده برای محاسبه ستون‌های حافظه column_cntr_next و column_cntr_reg
- مقدار ثابت برای استفاده در مالتی‌پلکسor (multiplexer). threshold
- سیگنال‌های کنترلی برای نوشتن داده در حافظه. writeSignal_next و writeSignal_reg
- سیگنال‌های کنترلی برای فعال کردن قسمت محاسبات trigger_nex و trigger_reg

- سیگنال last_ram_add یک آدرس ثابت برای حافظه (RAM) است که به عنوان آخرین آدرس قابل دسترس تعریف شده است.
- سیگنال‌های one و four مقادیر ثابت ۱ و ۴ هستند.
- سیگنال‌های temp1 و temp4 سیگنال‌های موقتی هستند که برای محاسبات موقتی استفاده می‌شوند.

توصیف معماری به این صورت است که با استفاده از این اجزا و سیگنال‌ها، رفتار و عملکرد ماژول "memoization" توصیف شده است. با توجه به توضیحات کد، به نظر می‌رسد که این ماژول برای محاسبه خروجی فیلتر با استفاده از حافظه (RAM) و ماشین حالت (state machine) طراحی شده است.

```

108 | begin
109 |   -- registers
110 |   process (clk,rst)
111 |   begin
112 |     if (rst = '1') then          -- determine reset values
113 |       state_reg <= idle;        --  reset state
114 |       diff_cntr_reg <= (others => '0');  -- determine reset values
115 |       add_inram_reg <= (others => '0');  -- determine reset values
116 |       writeSignal_reg <= '0';
117 |       trigger_reg <= '0';
118 |       column_cntr_reg <= (others => '0');
119 |
120 |     elsif (clk'event and clk='1') then    -- update RAM and registers in HIGH_EDGE_CLOCK
121 |       state_reg <= state_next;           -- state register
122 |       diff_cntr_reg <= diff_cntr_next;  -- row counter
123 |       add_inram_reg <= add_inram_next;  -- IN_RAM address
124 |       writeSignal_reg <= writeSignal_next;
125 |       trigger_reg <= trigger_next;
126 |       column_cntr_reg <= column_cntr_next;
127 |       data_in_mem <= in_memoized(to_integer(unsigned(add_inram_next(7 downto 0))));
128 |       if (writeSignal_reg = '1') then
129 |         in_ram(to_integer(unsigned(add_inram_reg))) <= data_in;      -- write new data on IN_RAM
130 |       end if;
131 |     end if;
132 |   end process;

```

شکل شماره ۳۳. مربوط به کلاک Process

این قطعه کد یک فرآیند (process) در زبان VHDL را نشان می‌دهد. این فرآیند بر اساس سیگنال‌های clk و rst که به عنوان ورودی دریافت می‌شوند، رجیسترها و سیگنال‌های دیگری را به روز می‌کند. در ابتدا، شرطی برای بررسی ریست (rst) وضعیت ۱ در نظر گرفته شده است. در صورتی که شرط ریست برقرار باشد، مقادیر رجیسترها و سیگنال‌ها به مقدار پیش‌فرض تنظیم می‌شوند. به عنوان مثال، مقدار column_cntr_reg به مقدار idle تنظیم می‌شود و مقادیر diff_cntr_reg و add_inram_reg به state_reg مقدار صفر تنظیم می‌شوند. در صورتی که شرط ریست برقرار نباشد و همچنین وقوع رویداد مثبت سیگنال

clk در نظر گرفته شده باشد، رجیسترها و سیگنال‌ها بر اساس مقادیر بعدی (next) خود به روزرسانی می‌شوند. به عنوان مثال، مقدار state_reg برابر با state_next قرار می‌گیرد و مقدار diff_cntr_reg برابر با diff_cntr_next قرار می‌گیرد. همچنین، در این بخش کد، مقدار data_in_mem برابر با مقدار writeSignal_reg برابر با add_inram_nex در آدرس محاسبه شده توسط in_memoized قرار می‌گیرد. اگر add_inram_reg با ۱ باشد، مقدار data_in در آدرس محاسبه شده توسط in_ram ذخیره می‌شود.

این قطعه کد یک فرآیند (process) در زبان VHDL است که بر اساس مقادیر ورودی و مقادیر رجیسترهاي diff_cntr_reg و add_inram_reg و row، column، input، start، state_reg و وضعیت فرآیند را به روز می‌کند. در ادامه کد، از یک ماشین حالت (state machine) استفاده شده است که با استفاده از مقادیر state_reg، وضعیت فعلی فرآیند را تعیین می‌کند و بر اساس آن، عملیات مورد نیاز را انجام می‌دهد. در هر وضعیت، عملیات مختلفی انجام می‌شود:

- در وضعیت idle، فرآیند در حالت آماده‌به‌شروع قرار دارد و منتظر دستور شروع (start) است. در صورتی که دستور شروع داده شود، وضعیت به start_rows تغییر می‌کند و مقادیر مرتبط با آدرس‌ها و شمارندها به روزرسانی می‌شوند.
- در وضعیت start_rows، سطرهای اولیه از ورودی (input) گرفته می‌شوند. در صورتی که تعداد سطرهای اندازه مورد نیاز برای تشخیص الگوی مورد نظر نرسیده باشد، فرآیند ادامه می‌یابد و مقادیر آدرس و شمارندها به روزرسانی می‌شوند. در غیر این صورت، وضعیت به approximate یا proces تغییر می‌کند بسته به شرایط و تعیین کننده‌های دیگر.
- در وضعیت approximate و proces، محاسبات لازم برای تولید خروجی (data_out) انجام می‌شود و مقادیر آدرس‌ها و شمارندها به روزرسانی می‌شوند. در صورتی که به پایان فرآیند رسیده باشیم، وضعیت به idle تغییر می‌کند.

```

134 process(state_reg,start,input,column,row,add_inram_reg,diff_cntr_reg)
135 begin
136   case state_reg is      -- determine state
137
138     when idle      =>          -- 1)IDLE state : wait for start
139       trigger_next <= '0';
140       if (start = '1') then      -- wait for start command
141         state_next    <= start_rows;  -- next state
142         add_inram_next <= (others => '0');  -- set address IN_RAM
143         diff_cntr_next <= (others => '0');  -- set address OUT_RAM
144         writeSignal_next <= '1';
145         column_cntr_next <= "00000001";
146       end if;
147
148

```

شکل شماره ۳۴.

این بخش از کد، بخش مربوط به وضعیت idle است که در آن منتظر دستور شروع (start) قرار دارد. در این بخش، ابتدا سیگنال trigger_next به صفر تنظیم می‌شود. سپس با استفاده از شرط ($start = '1'$) بررسی می‌شود که آیا دستور شروع داده شده است یا خیر. اگر دستور شروع داده شده باشد، مقادیر زیر به

روزرسانی می‌شوند:

- state_next به تنظیم می‌شود که وضعیت بعدی را نشان می‌دهد.
- add_inram_next به مقدار صفر تنظیم می‌شود تا آدرس IN_RAM را صفر کند.
- diff_cntr_next به مقدار صفر تنظیم می‌شود تا آدرس OUT_RAM را صفر کند.
- writeSignal_next به یک تنظیم می‌شود تا فعال کننده نوشتمن (writeSignal) را فعال کند.
- column_cntr_next به مقدار "00000001" تنظیم می‌شود تا شمارنده ستون (column_cntr) را به حالت اولیه برگرداند.

به طور خلاصه، در وضعیت idle، فرآیند در حالت آماده به شروع قرار دارد و منتظر دستور شروع است. در صورتی که دستور شروع داده شود، وضعیت بعدی به start_rows تغییر می‌کند و مقادیر مرتبط با آدرس‌ها و شمارندها به روزرسانی می‌شوند.

```

150 √ when start_rows      =>          -- 2)START_ROWS state : get first 3 rows from INPUT
151 √   | if(add_inram_reg = std_logic_vector(unsigned(last_ram_addr)+1)) then    -- Diagnostic 3 rows
152   |   | writeSignal_next <= '0';
153   |   | if(diff_cntr_reg < threshold) then
154   |   |   | state_next <= approximate;           -- next state
155   |   |   | trigger_next <= '0';
156   |   |   | add_inram_next <= (others => '0');    -- set address IN_RAM
157   |   |
158   |   | else
159   |   |   | state_next <= proces;             -- next state
160   |   |   | trigger_next <= '1';
161   |   |   | add_inram_next <= "00000000" & std_logic_vector(unsigned(column)+1);    -- set address IN_RAM
162   |   |
163   |   | end if;
164   |   |
165   |   | else
166   |   |   | data_in <= input;        -- for write new data
167   |   |   | add_inram_next <= std_logic_vector(unsigned(add_inram_reg) + 1);    -- increment address IN_RAM
168   |   |   | if(data_in_mem = input) then
169   |   |   |   | diff_cntr_next <= diff_cntr_reg;
170   |   |   |   | else
171   |   |   |   |   | diff_cntr_next <= std_logic_vector(unsigned(diff_cntr_reg) + 1);
172   |   |   |   | end if;
173   |   | end if;

```

شکل شماره .۳۵ Start_rows state

این بخش از کد، بخش مربوط به وضعیت start_rows است که در آن سطرهای اولیه از ورودی (input) گرفته می‌شوند. در این بخش، ابتدا سیگنال trigger_next به صفر تنظیم می‌شود. سپس با استفاده از شرط if(add_inram_reg = std_logic_vector(unsigned(last_ram_addr)+1)) بررسی می‌شود که آیا آدرس IN_RAM برابر با آخرین آدرس حافظه IN_RAM است یا خیر. اگر آدرس برابر با آخرین آدرس حافظه باشد، مراحل مربوط به تشخیص الگو و تولید خروجی در نظر گرفته می‌شود. در غیر این صورت، مراحل مربوط به نوشتمندانه جدید در حافظه را انجام می‌دهد.

در صورتی که آدرس برابر با آخرین آدرس حافظه باشد، مراحل زیر انجام می‌شود:

- writeSignal_next به صفر تنظیم می‌شود تا غیرفعال کننده نوشتمندانه (writeSignal_next) را فعال کند.
- با استفاده از شرط if(diff_cntr_reg < threshold)، بررسی می‌شود که آیا مقدار شمارنده تفاوت (diff_cntr) کمتر از آستانه‌ای به نام threshold است یا خیر. اگر کمتر باشد، وضعیت بعدی به trigger_next تغییر می‌کند و سیگنال approximate به صفر تنظیم می‌شود. همچنین، آدرس IN_RAM به صفر تنظیم می‌شود.

- در غیر این صورت، وضعیت بعدی به `proces trigger_next` می‌کند و سیگنال `IN_RAM` به مقدار "۰۰۰۰۰۰" در ادامه‌ی آدرس ستون (column) می‌شود. همچنین، آدرس `IN_RAM` می‌شود.

در صورتی که آدرس برابر با آخرین آدرس حافظه نباشد، مراحل زیر انجام می‌شود:

- داده‌ی جدید به متغیر `data_in` تنظیم می‌شود تا بتوانیم داده جدید را در حافظه ذخیره کنیم.
- آدرس `IN_RAM` به مقدار آدرس قبلی (`add_inram_reg`) با یک واحد افزایش می‌یابد.
- با استفاده از شرط `if(data_in_mem == input)` بررسی می‌شود که آیا داده در حافظه (data_in_mem) برابر با داده ورودی است (input) یا خیر. در صورتی که برابر باشد، مقدار شمارنده تفاوت (`diff_cntr`) بدون تغییر باقی می‌ماند. در غیر این صورت، مقدار شمارنده تفاوت یک واحد را با اضافه `1` به مقدار شمارنده فعلی (data_in_mem) برابر می‌کند.
- در این بخش، در ابتدا سیگنال "trigger_next" به ۰ تنظیم می‌شود. سپس با استفاده از شرط `if(add_inram_reg == std_logic_vector(unsigned(last_ram_addr)+1))` آیا آدرس "IN_RAM" برابر با آخرین آدرس حافظه "IN_RAM" است یا خیر. اگر برابر باشد، مراحل مربوط به نوشتنداده جدید در حافظه انجام می‌شود.

در صورتی که آدرس برابر با آخرین آدرس حافظه باشد، مراحل زیر انجام می‌شوند:

- سیگنال "writeSignal_next" به ۰ تنظیم می‌شود تا فعال کننده نوشتنداده (writeSignal) را غیرفعال کند.
- با استفاده از شرط `"if(diff_cntr < threshold)"` بررسی می‌شود که آیا مقدار شمارنده تفاوت (`diff_cntr`) کمتر از آستانه‌ای به نام "threshold" است یا خیر. اگر کمتر باشد، وضعیت بعدی به

"trigger_next" به ۰ تنظیم می‌شود. همچنین، آدرس "approximate" می‌شود.

- در غیر این صورت، وضعیت بعدی به "proces" تغییر می‌کند و سیگنال "trigger_next" به ۱ تنظیم می‌شود. همچنین، آدرس "IN_RAM" به مقدار "....." به اضافه‌ی آدرس ستون (column) می‌شود.

در صورتی که آدرس برابر با آخرین آدرس حافظه نباشد، مراحل زیر انجام می‌شوند:

- داده جدید به متغیر "data_in" تنظیم می‌شود تا بتوانیم داده جدید را در حافظه ذخیره کنیم.
- آدرس "IN_RAM" به مقدار آدرس قبلی (add_inram_reg) با یک واحد افزایش می‌یابد.
- با استفاده از شرط "if(data_in_mem = input)"، بررسی می‌شود که آیا داده در حافظه با ابر با داده ورودی است (input) یا خیر. در صورتی که برابر باشد، مقدار شمارنده تفاوت (diff_cntr) بدون تغییر باقی می‌ماند. در غیر این صورت، مقدار شمارنده تفاوت یک واحد افزایش می‌یابد.

```

174
175    when approximate      =>                               -- 3) PROCES state : compute and generate filter OUT and GET new data from input port
176        trigger_next <= '0';
177        if(add_inram_reg = last_ram_addr) then   -- for determin end of process
178            state_next <= idle;           -- go to idle state in end
179
180        else
181            add_inram_next <= std_logic_vector(unsigned(add_inram_reg) + 1);   -- increment address IN_RAM
182            data_out <= "00000001" & out_memoized(to_integer(unsigned(add_inram_reg)));
183        end if;

```

شکل شماره .۳۶

این کد یک قسمت از یک ماشین حالتی را نشان می‌دهد که در وضعیت "approximate" قرار دارد. در این وضعیت، الگوریتم فیلتراسیون را اجرا می‌کند و خروجی فیلتر را محاسبه می‌کند و داده جدید را از پورت ورودی دریافت می‌کند. در ابتدا، سیگنال "trigger_next" به صفر تنظیم می‌شود. سپس با استفاده از شرط "if(add_inram_reg = last_ram_addr)"، بررسی می‌شود که آیا آدرس "IN_RAM" برابر با آخرین آدرس حافظه "IN_RAM" است یا خیر. در صورت برابر بودن، به انتهای عملیات فیلتراسیون رسیده‌ایم و

وضعیت بعدی را به "idle" تنظیم می‌کنیم تا به وضعیت آرام بازگردیم. در غیر این صورت، آدرس "IN_RAM" به مقدار آدرس فعلی ("add_inram_reg") با یک واحد افزایش می‌یابد. همچنین، خروجی فیلتر ("data_out") با استفاده از آدرس فعلی از حافظه ("out_memoized") و یک بیت ثابت با مقدار "....." تنظیم می‌شود. این بیت ثابت به عنوان بیت کنترلی مورد استفاده قرار می‌گیرد تا نشان دهد که این خروجی مربوط به فیلتراسیون است و نه داده ورودی اصلی.

```

187 when proces      =>          -- 3)PROCES state : compute and generate filter OUT and GET new data from input port
188   trigger_next <= '0';
189   if(add_inram_reg = std_logic_vector(unsigned(last_ram_addr)-unsigned(column)+3)) then    -- for determin end of process
190     state_next <= idle;           -- go to idle state in end
191   else
192
193     if (column_cntr_reg < std_logic_vector(unsigned(column)-2)) then           -- for go to column-1 pixel
194       add_inram_next <= std_logic_vector(unsigned(add_inram_reg) + 1);    -- increment address OUT_RAM
195       column_cntr_next <= std_logic_vector(unsigned(column_cntr_reg) + 1);
196
197     else
198       add_inram_next <= std_logic_vector(unsigned(add_inram_reg) + 3);    -- increment address OUT_RAM
199       column_cntr_next <= "00000001";
200     end if;
201     data_out_1 <= in_ram(to_integer(unsigned(add_inram_reg)-unsigned(column)-1));
202     data_out_2 <= in_ram(to_integer(unsigned(add_inram_reg)-unsigned(column)));
203     data_out_3 <= in_ram(to_integer(unsigned(add_inram_reg)-unsigned(column)+1));
204
205     data_out_4 <= in_ram(to_integer(unsigned(add_inram_reg)-1));
206     data_out_5 <= in_ram(to_integer(unsigned(add_inram_reg))); --center house
207     data_out_6 <= in_ram(to_integer(unsigned(add_inram_reg)+1));
208
209     data_out_7 <= in_ram(to_integer(unsigned(add_inram_reg)+unsigned(column)-1));
210     data_out_8 <= in_ram(to_integer(unsigned(add_inram_reg)+unsigned(column)));
211     data_out_9 <= in_ram(to_integer(unsigned(add_inram_reg)+unsigned(column)+1));
212     if (temp4 > temp1) then
213       | data_out <= std_logic_vector(unsigned(temp4) - unsigned(temp1));
214     else
215       | data_out <= std_logic_vector(unsigned(temp1) - unsigned(temp4));
216     end if;
217   end if;
218 end case;
219 end process;
220

```

شکل شماره ۳۷.

این کد نیز یک قسمت از ماشین حالت را نشان می‌دهد. در این قسمت، ما در وضعیت "proces" قرار داریم و در آن عملیات محاسبه و تولید خروجی فیلتر را انجام می‌دهیم و همچنین داده جدید را از پورت ورودی دریافت می‌کنیم. در ابتدا، سیگنال "trigger_next" به صفر تنظیم می‌شود. سپس با استفاده از شرط "if(add_inram_reg = std_logic_vector(unsigned(last_ram_addr)-unsigned(column)+3))" بررسی می‌شود که آیا آدرس "IN_RAM" برابر با آدرس آخرین ردیف حافظه "IN_RAM" منهای مقدار ستون ("column") منهای ۳ است یا خیر. در صورت برابر بودن، به انتهای عملیات فیلتراسیون رسیده‌ایم و وضعیت بعدی را به "idle" تنظیم می‌کنیم تا به وضعیت آرام بازگردیم. در غیر این صورت، ابتدا با استفاده از شرط "if (column_cntr_reg < std_logic_vector(unsigned(column)-2))" بررسی می‌شود که آیا

شمارنده ستون ("column_cntr_reg") کوچکتر از مقدار ستون منهای ۲ است یا خیر. اگر شرط برقرار باشد، آدرس "OUT_RAM" به مقدار آدرس فعلی ("add_inram_reg") با یک واحد افزایش می‌یابد و شمارنده ستون نیز با یک واحد افزایش مقدار خود را به روز می‌کند. در غیر این صورت، آدرس "OUT_RAM" به مقدار آدرس فعلی با افزایش ۳ و شمارنده ستون به مقدار "۱۰۰۰۰۰۰" تنظیم می‌شود. سپس، داده‌های ورودی مربوط به فیلتر را از حافظه "IN_RAM" با استفاده از آدرس‌های مختلف محاسبه می‌کنیم و به متغیرهای "temp1" تا "data_out_9" با اختصاص می‌دهیم. سپس، با استفاده از شرط "if (temp4 > temp1)"، مقدار خروجی فیلتر را محاسبه می‌کنیم و به "data_out" اختصاص می‌دهیم. اگر شرط برقرار باشد، مقدار خروجی فیلتر برابر با تفاضل دو مقدار "temp1" و "temp4" خواهد بود و در غیر این صورت، مقدار خروجی فیلتر برابر با تفاضل دو مقدار "temp1" و "temp4" خواهد بود. در نهایت، وضعیت بعدی ماشین حالتی به وضعیت "proces" تنظیم شده است.

```

222   output <= data_out(7 downto 0);    -- assign the output
223   trigger <- trigger_next;
224   temp4 <- std_logic_vector(unsigned(data_out_5)*unsigned(four));
225   temp1 <- std_logic_vector(unsigned(data_out_2)*unsigned(one)+unsigned(data_out_6)*unsigned(one)+unsigned(data_out_8)*unsigned(one)+unsigned(data_out_4)*unsigned(one));
226
227 end memoization_beh;
228

```

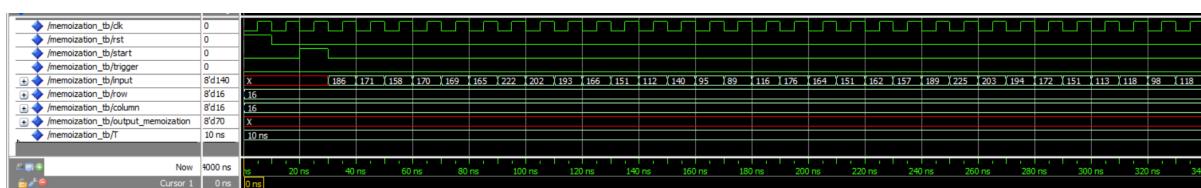
شکل شماره ۳۷.

در خط اول، خروجی "output" به اعضای ۷ تا ۰ متغیر "data_out" اختصاص داده می‌شود. در اینجا، اطلاعات ۸ بیتی موجود در متغیر "data_out" تنها تا بیت ۷ تخصیص داده می‌شود. در خط دوم، سیگنال "trigger" به مقدار سیگنال "trigger_next" تنظیم می‌شود. این عملیات به منظور به روزرسانی سیگنال "trigger" با مقدار بعدی آن استفاده می‌شود. در خط سوم و چهارم، دو متغیر جدید با نامهای سیگنال "trigger" با مقدار شده‌اند و مقادیر آن‌ها محاسبه می‌شوند. این محاسبات بر اساس مقادیر موجود در متغیرهای "temp1" و "temp4" تعریف شده‌اند و مقادیر آن‌ها محاسبه می‌شوند. در مجموع، دو مقدار "temp1" و "temp4" می‌گیرد. در محاسبه "temp4"، ضرب دو مقدار "data_out_5" و "data_out_4" صورت می‌گیرد و نتیجه به صورت یک عدد با گستره بیتی مناسب در متغیر "temp4" ذخیره می‌شود. در محاسبه "temp1"، جمع چند مقدار

موردنیاز صورت می‌گیرد و نتیجه به صورت یک عدد با گستره بیتی مناسب در متغیر "temp1" ذخیره می‌شود.

نتایج شبیه‌سازی مازول تشخیص لبه

ما در این قسمت برای نشان دادن عملکرد درست مازول نهایی، با استفاده از دو test bench مختلف یک test bench برای ورودی که نیازی به محاسبه مجدد خروجی برای آن نیست، یعنی ورودی که به ورودی ثبت شده در حافظه مازول نزدیک است و از خروجی ذخیره شده در واحد Memoization استفاده می‌کند که به آن حالت approximate می‌گوییم و یک دیگر برای حالتی که ورودی به قدری متفاوت است که نمی‌توان از واحد Memoization استفاده کرد، برای تست عملکرد مازول تشخیص لبه استفاده می‌کنیم و خروجی wave form و مقادیر محاسبه شده را نشان خواهیم داد. لازم به ذکر است ما عملکرد مازول را برای تصاویر با ابعاد و حالت‌های مختلف امتحان کردیم اما در اینجا صرفاً به نتایج تعداد محدودی از آن‌ها اکتفا کرده و سایر نتایج را در پوشه‌ی پروژه قرار دادیم تا درصورت نیاز به آن مراجعه شود. نکته‌ی مهم، سیگنال trigger می‌باشد، این سیگنال در حالتی که ورودی مشابه ورودی ذخیره شده در مازول باشد و بتوان از خروجی ذخیره شده به جای محاسبه مجدد خروجی استفاده کرد (حالت memoization) برابر صفر (شکل ۳۹) و در حالتی که نیاز به محاسبه خروجی باشد و امکان استفاده از خروجی ذخیره شده وجود نداشته باشد (حالت non memoization) برابر یک (شکل شماره ۴۲) قرار می‌گیرد، در واقع ما از این سیگنال برای راه اندازی واحد محاسبات اصلی در صورت نیاز استفاده می‌کنیم.



شکل شماره ۳۸. ابتدای شبیه‌سازی حالت memoization



شکل شماره ۳۹. قرار گرفتن خروجی حالت memoization

0	105 0 99 0 101 0 92 0 72 0 97 0 114 0 100 0
16	152 4 26 20 42 25 64 15 14 10 17 46 21 47 29 0
32	109 6 4 3 21 8 46 7 94 69 75 23 69 8 77 0
48	115 15 37 13 93 75 134 145 81 131 60 92 43 43 75 0
64	68 11 18 89 31 138 118 141 111 24 30 78 100 12 1 0
80	76 21 2 65 106 9 131 50 5 1 13 59 16 45 46 0
96	92 10 36 73 46 44 65 13 40 8 3 27 33 23 58 0
112	109 2 53 36 100 53 17 33 20 72 39 9 26 8 22 0
128	101 26 8 23 55 5 48 8 68 5 43 119 70 61 31 0
144	160 24 39 19 27 74 0 44 5 93 15 51 1 42 71 0
160	0 51 3 16 6 48 13 8 11 125 43 8 40 47 137 0
176	64 16 27 48 28 23 85 26 103 0 23 32 60 28 67 0
192	16 39 11 21 49 55 59 3 22 15 9 10 3 62 126 0
208	4 44 10 11 32 57 16 20 33 55 75 21 29 15 26 0
224	0 15 23 52 95 47 16 80 5 73 9 19 65 89 114 0
240	51 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

شکل شماره ۳۹. مقادیر قرار گرفته در خروجی حالت memoization

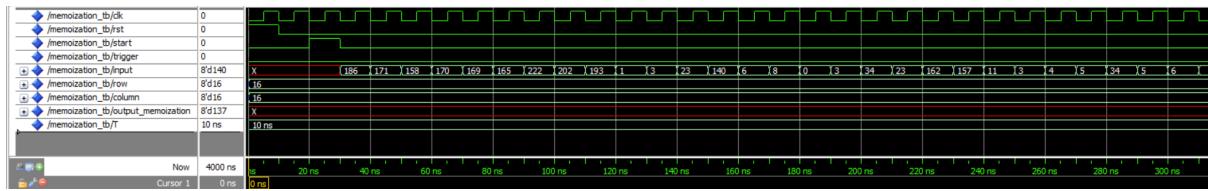
1	105 0 99 0 101 0 92 0 72 0 97 0 114 0 100 0
2	152 4 26 20 42 25 64 15 14 10 17 46 21 47 29 0
3	109 6 4 3 21 8 46 7 94 69 75 23 69 8 77 0
4	115 15 37 13 93 75 134 145 81 131 60 92 43 43 75 0
5	68 11 18 89 31 138 118 141 111 24 30 78 100 12 1 0
6	76 21 2 65 106 9 131 50 5 1 13 59 16 45 46 0
7	92 10 36 73 46 44 65 13 40 8 3 27 33 23 58 0
8	109 2 53 36 100 53 17 33 20 72 39 9 26 8 22 0
9	101 26 8 23 55 5 48 8 68 5 43 119 70 61 31 0
10	160 24 39 19 27 74 0 44 5 93 15 51 1 42 71 0
11	0 51 3 16 6 48 13 8 11 125 43 8 40 47 137 0
12	64 16 27 48 28 23 85 26 103 0 23 32 60 28 67 0
13	16 39 11 21 49 55 59 3 22 15 9 10 3 62 126 0
14	4 44 10 11 32 57 16 20 33 55 75 21 29 15 26 0
15	0 15 23 52 95 47 16 80 5 73 9 19 65 89 114 0
16	51 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

شکل شماره ۴۰. مقادیر خروجی برنامه C

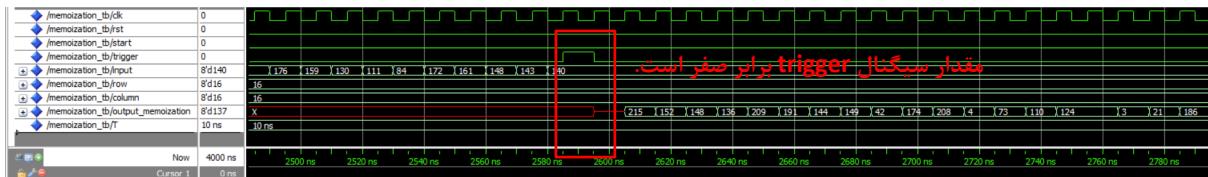
همانطور که شکل شماره ۳۹ و ۴۰ نشان می‌دهند، خروجی حالت Memoization کاملاً درست بوده

و مازول در این حالت به درستی عمل می‌کند. در قسمت بعدی، خروجی مازول را در حالت non-Memoization بررسی می‌کنیم. همانطور که شکل ۴۲ نشان می‌دهند، خروجی در این حالت نیز کاملاً درست بوده و مازول به درستی عمل می‌کند، نکته‌ی مهم این قسمت، سیگنال trigger می‌باشد که هنگام قرار گرفتن

مقدادیر در خروجی ماذول مقدار آن یک شده است و به این معناست که ورودی ماذول مشابه نبوده و قسمت محاسبات باید فعال شود و خروجی را محاسبه کند.



شکل شماره ۴۱. ابتدای شبیه‌سازی حالت non-memoization



شکل شماره ۴۲. قرار گرفتن خروجی حالت non-memoization

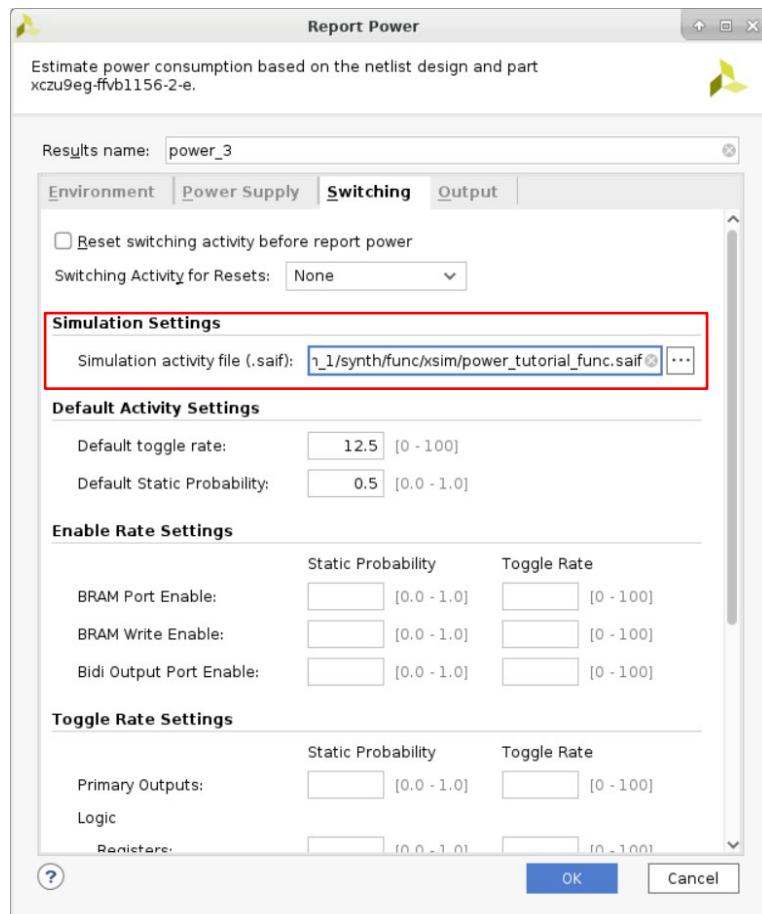
گزارش توان

بعد از اطمینان از عملکرد ماذول نهایی در هر دو حالت، از نرم‌افزار ModelSim، با استفاده از دستورات زیر فایل saif. شبیه‌سازی ها را دریافت کردیم. همانطور که قبل اشاره کردیم، این فایل حاوی مقدادیر سیگنال‌ها در واحد زمان است و با استفاده از آن می‌توانیم توان ماذول را بعد از synthesis و implementation در نرم‌افزار Vivado به ازای ورودی و سناریوی خاص حساب کرد. برای استخراج فایل saif. باید در محیط نرم‌افزار Vivado از دستورات زیر استفاده کرد:

- ❖ power add memoization_TB/memoization/*
- ❖ run 4600ns
- ❖ power report -all -bsaif routed.saif
- ❖ quit

بعد از استخراج فایل saif. و implementation کردن ماذول تشخیص لبه، با زدن گزینه‌ی Power Report، پنجره‌ی تنظیمات مربوطه به توان باز می‌شود که همانند شکل زیر از سربرگ Switching می‌توان

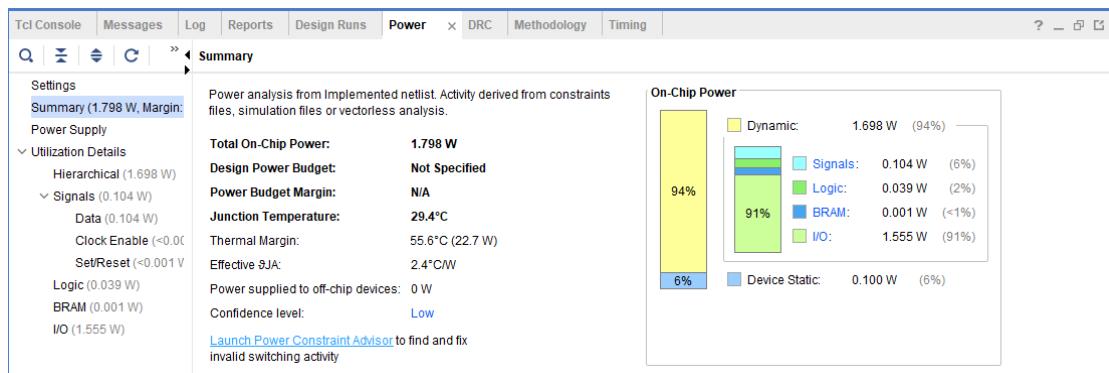
آدرس فایل saif را داد و توان را به ازای ورودی و سناریوی خاص حساب کرد تخمین زد. لازم به ذکر است که می‌توان توان را بعد از مرحله‌ی سنتز نیز تخمین زد، اما هرچه این کار را در مراحل بالاتر انجام بدهیم، دقیق تخمین بیشتر شده و خروجی به واقعیت نزدیک‌تر است.



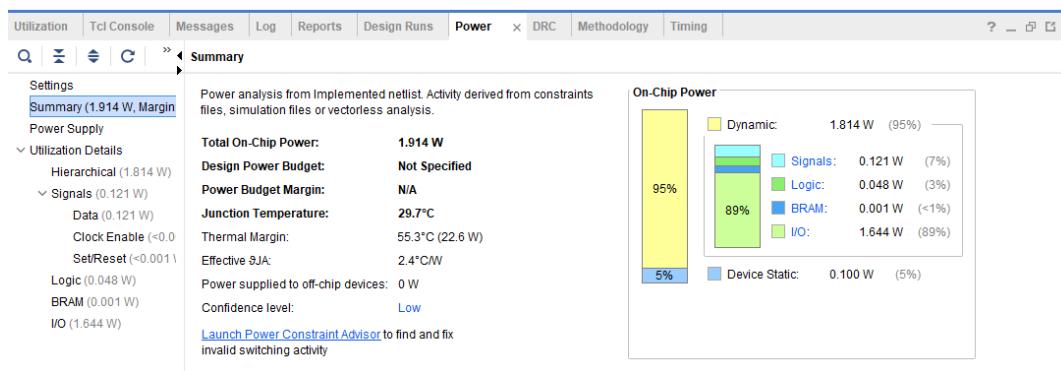
شکل شماره ۴۳. نحوه استفاده از فایل .saif

شکل‌های شماره ۴۴ و ۴۵ و ۴۶ تخمین توان را برای سه حالت با memoization، بدون memoization و دقیق نشان می‌دهند. همانطور که نتایج بالا نشان می‌دهد، در حالت memoization نسبت به حالت بدون memoization، حدود ۱۱ درصد بهبود در کل توان دینامیکی داریم، همچنین در بحث توان مربوط به logic یا منطق، حدود ۲۳ درصد بهبودی حاصل شده که نزدیک به عدد ذکر شده در مقاله یعنی ۲۰ درصد بوده و ایده مطرح شده در مقاله را کاملاً تایید می‌کند. نکته بعدی تفاوت توان signal می‌باشد که دلیل آن تفاوت اندک ورودی‌ها در حالت‌های مختلف بوده و کاملاً طبیعی است. موضوع بعدی بالا بودن میزان توان در قسمت در تمامی حالت‌هاست. شاید علت این موضوع استفاده از برد و تنظیمات تعداد زیادی از

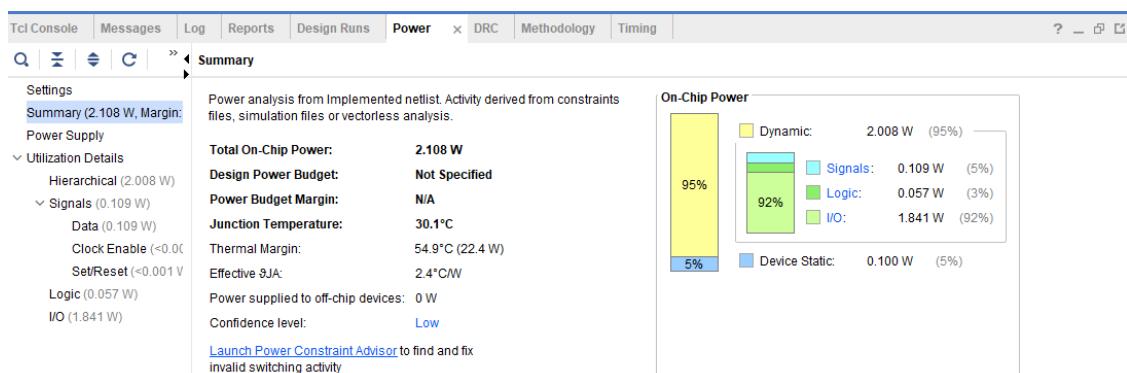
پین‌های IO بصورت پیش‌فرض می‌باشد. نکته بعدی ثابت ماندن توان استاتیک در هر سه حالت می‌باشد. لازم به ذکر است که این مقادیر برای ابعاد ۱۶ در ۱۶ می‌باشد. انتظار ما این بود که با افزایش ابعاد تصویر، میزان بهبودی افزایش یابد، به همین دلیل تمام مراحل بالا را برای تصویری با ابعاد ۶۴ در ۶۴ نیز انجام دادیم، مشاهده کردیم میزان بهبودی توان دینامیکی تا حدود ۱۱ درصد افزایش یافت، تمام نتایج مربوط به تصویر ۶۴ در ۶۴ را در پوشه result قرار دادیم تا در صورت نیاز به آن مراجعه شود.



شکل شماره ۴۴. گزارش توان در حالت Memoization



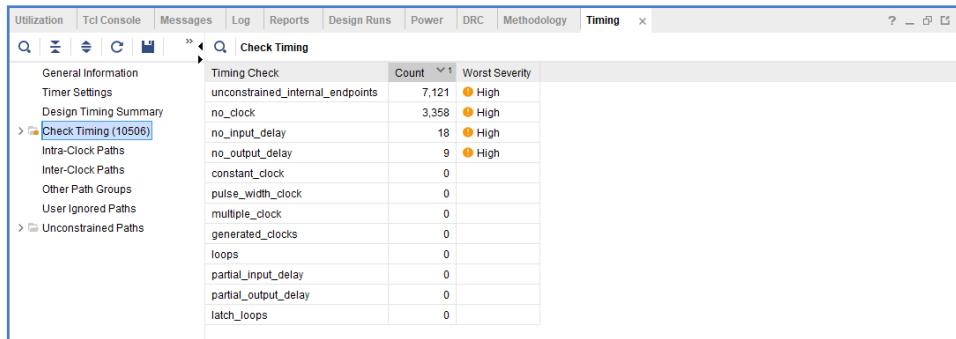
شکل شماره ۴۵. گزارش توان در حالت No-Memoization



شکل شماره ۴۶. گزارش توان در حالت دقیق

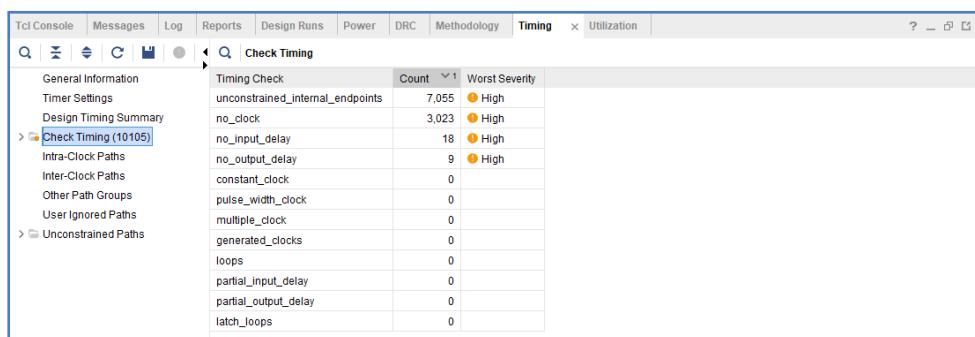
گزارش کلاک

همانطور که شکل های پایین نشان می دهند، در بحث تعداد کلاک برای تولید کامل خروجی، تفاوتی بین دو حالت memoization و non-memoization وجود ندارد، زیرا الگوریتم و نحوهی قراردادن سیگنال خروجی در هر دو حالت یکسان می باشد، در هر دو حالت، ابتدا به تعداد پیکسل های تصویر، منتظر مانده تا کل تصویر را از ورودی دریافت کند، سپس بعد از دریافت، آخرین پیکسل، با توجه به مقدار تفاوت تصویر ورودی و تصویر ذخیره شده در واحد memoization و مقدار آستانه در نظر گرفته شده، در هر دو حالت به میزان تعداد پیکسل های تصویر کلاک نیاز است تا خروجی تولید و برروی پورت مربوطه قرار داده شود، البته تفاوت جزئی بین دو حالت وجود دارد که علت آن، این است که در حالت non-memoization و زمانی که خروجی با محاسبات تولید می شود، ما دو عدد از سطر و ستون آن کم می کنیم، یعنی اگر تصویر ورودی ۱۶ در ۱۶ باشد، خروجی در حالت محاسبات ۱۴ در ۱۴ می باشد زیرا نمی توان سطر و ستون اول و آخر را با اعمال کرنل محاسبه کرد.



Timing Check	Count	Worst Severity
unconstrained_internal_endpoints	7,121	High
no_clock	3,358	High
no_input_delay	18	High
no_output_delay	9	High
constant_clock	0	
pulse_width_clock	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

شکل شماره ۴۷. گزارش کلاک در حالت memoization

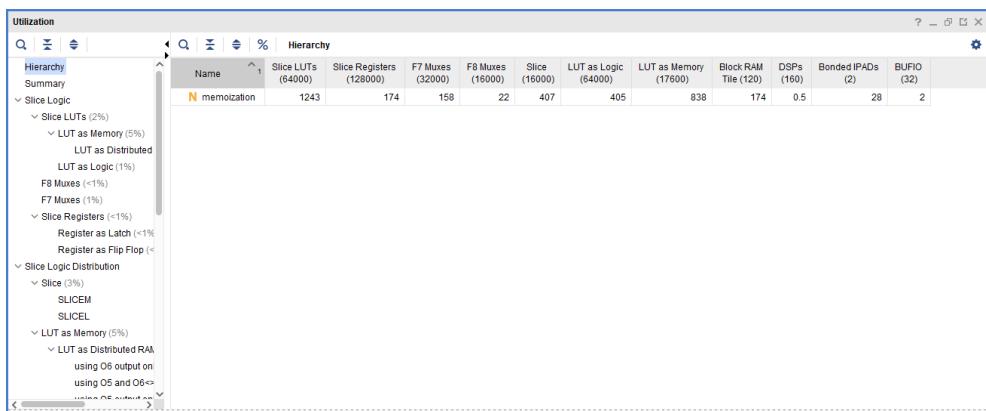


Timing Check	Count	Worst Severity
unconstrained_internal_endpoints	7,055	High
no_clock	3,023	High
no_input_delay	18	High
no_output_delay	9	High
constant_clock	0	
pulse_width_clock	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

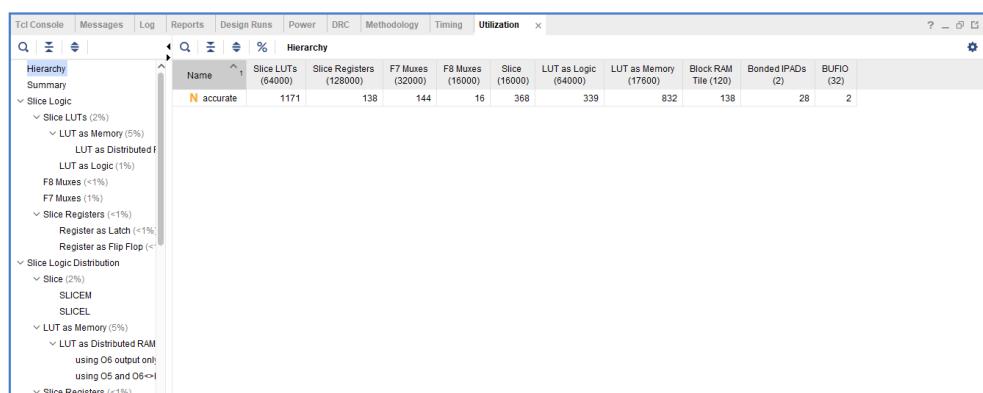
شکل شماره ۴۸. گزارش کلاک در حالت non-memoization

گزارش منابع

در این قسمت، گزارش‌هایی مثل زمان‌بندی، منابع استفاده شده و کلک را آورده‌ایم، توجه فرمایید به دلیل استفاده مستقیم از LUT برای نگهداری مقادیر ورودی و خروجی برای واحد memoization، میزان استفاده از LUT زیاد شده است. همچنین مقاله ذکر نکرده که آیا از بهینه‌سازها استفاده کرده یا نه و اگر استفاده کرده از چه بهینه‌سازی، همچنین مقاله نگفته که نحوه دادن ورودی به ماژول و گرفتن خروجی چگونه است، همچنین مقاله هیچ اشاره‌ای به نحوه محاسبه توان نکرده بود، تمام این موارد باعث می‌شود که خروجی ما دقیقاً مطابق با خروجی مقاله نباشد.



شکل شماره ۴۹. گزارش منابع در حالت memoization



شکل شماره ۵۰. گزارش منابع در حالت non-memoization

نکات تکمیلی

در این قسمت، به چندین نکته مهم اشاره خواهیم کرد، نکته‌ی اول این است که ما نیاز به تبدیل عکس‌ها و تصاویر به مقادیر عددی، ساخت آرایه ورودی برای کد C، ساخت Test Bench برای تست کد VHDL، ساخت واحدهای ورودی و خروجی memoization و غیره داشتیم، برای راحتی و افزایش سرعت، اسکریپت‌ها و برنامه‌هایی نوشتم که تمامی این کارها را خودکار انجام داده و تمام فایل‌های و موارد مورد نیاز را بصورت خودکار برای ما تولید می‌کند (ما تمام فایل‌های تولیدی و مقادیر مورد نیاز را در پوشه‌ی مربوط به هر تصویر قرار دادیم).

نکته دوم، ابعاد تصویر می‌باشد، خود مقاله از ابعاد ۳۲۰ در ۲۴۰ استفاده کرده، ما هم در ابتدا خواستیم از ابعاد ۲۵۶ در ۲۵۶ استفاده کنیم، اما به دلیل اینکه ما برخلاف مقاله که بصورت واقعی پیاده‌سازی کرده بودند، از شبیه‌سازی استفاده می‌کردیم، زمان بسیار زیادی برای کامپایل و شبیه‌سازی test bench مورد نیاز بود، به همین دلیل تصمیم گرفتیم از ابعاد ۶۴ در ۶۴ و ۱۶ در ۱۶ استفاده کنیم.

نکته سوم این است که تمام مقادیر بدست آمده در قسمت گزارش‌ها، برای ابعاد ۱۶ در ۱۶ می‌باشد و ما در این گزارش، صرفاً به ذکر آن اکتفا کردیم، اگرچه ما برای سایر تصاویر و ابعاد ۶۴ در ۶۴، مراحل بالا را تکرار کرده و در پوشه مربوط به هر کدام، تصاویر خروجی را قرار دادیم اما از آوردن آن‌ها در این گزارش برای طولانی نشدن آن، خودداری کردیم.

نکته چهارم این است که مقاله ذکر نکرده که آیا از بهینه‌سازها استفاده کرده یا نه و اگر استفاده کرده از چه بهینه‌سازی، همچنین مقاله نگفته که نحوه دادن ورودی به ماژول و گرفتن خروجی چگونه است، همچنین مقاله هیچ اشاره‌ای به نحوه محاسبه توان نکرده بود، تمام این موارد باعث می‌شود که خروجی ما دقیقاً مطابق با خروجی مقاله نباشد، اگرچه میزان بهبودی و نتیجه‌ی اصلی پیاده‌سازی ما کاملاً نزدیک به نتیجه نهایی مقاله بود (بهبودی ۲۳ درصدی در توان لاجیک در مقایسه با بهبودی ۲۰ درصدی ذکر شده در مقاله).

بخش پنجم: نحوه‌ی پیاده‌سازی تبدیل فضای رنگی RGB به YCbCr

بسایر از نکات این بخش، مشابه بخش قبلی بوده و تکراری است ما برای طولانی نشدن گزارش، از گفتن مجدد آن‌ها خودداری کرده و به ذکر مطالب مهم اکتفا خواهیم کرد، همچنین لازم به ذکر است که پیاده‌سازی تبدیل فضای رنگی به نسبت تشخیص لبه آسان‌تر بوده و مشابه قسمت قبل، این مژوول را با استفاده از ابزار HLS تولید کردیم.

توضیح کد مژوول تبدیل فضای رنگی RGB به YCbCr با زبان C

بطور خلاصه، این کد یک تابع است که مقادیر رنگ سه بایتی (قرمز، سبز و آبی) را به فضای رنگ YCbCr تبدیل می‌کند. فضای رنگ YCbCr یک فضای رنگ مختص تصاویر و ویدئوهای رنگی است که برای نمایش رنگ‌ها و اطلاعات سیاه و سفید در کنار هم استفاده می‌شود.

در قسمت اول تابع، سه مقدار ثابت Crmem، Ymem و Cbmem به ترتیب به عنوان مقادیر پیش‌فرض برای Y، Cr و Cb تعریف می‌شوند. سپس، با استفاده از یک شرط، بررسی می‌شود که آیا مقادیر رنگ ورودی (R، G و B) در محدوده‌های مشخصی قرار دارند یا خیر. اگر مقادیر رنگ ورودی در این محدوده‌ها قرار داشته باشند، یعنی رنگ ورودی به مقادیر ثابتی شبیه است. در این صورت، مقادیر Y، Cr و Cb با استفاده از مقادیر ثابت Crmem، Ymem و Cbmem مقداردهی می‌شوند و متغیر Δ برابر ۱ قرار می‌گیرد. این یعنی رنگ ورودی مشابه یک رنگ ثابت است و نیازی به محاسبه تبدیل آن به فضای رنگ YCbCr نیست.

در غیر این صورت، اگر مقادیر رنگ ورودی در محدوده‌های مشخصی قرار نداشته باشند، مقادیر Y، Cr و Cb با استفاده از فرمول‌های مشخص شده محاسبه می‌شوند. این فرمول‌ها بر اساس روابط ریاضی بین مقادیر رنگ و فضای رنگ YCbCr است. همچنین، مقادیر ۱۶ و ۱۲۸ به صورت ثابت به فرمول‌ها اضافه می‌شوند. متغیر f در این حالت برابر 0 قرار می‌گیرد که نشان دهنده این است که رنگ ورودی تبدیل شده است ولی با رنگ ثابت مشابه نیست.

در نهایت، ساختار YCbCr حاصل با مقادیر محاسبه شده برای Y، Cb و Cr و مقداردهی شده برای f به عنوان خروجی تابع برگردانده می‌شود. این ساختار شامل مقادیر Y، Cb، Cr و f است که نمایانگر مقادیر فضای رنگ YCbCr برای رنگ ورودی می‌باشد.

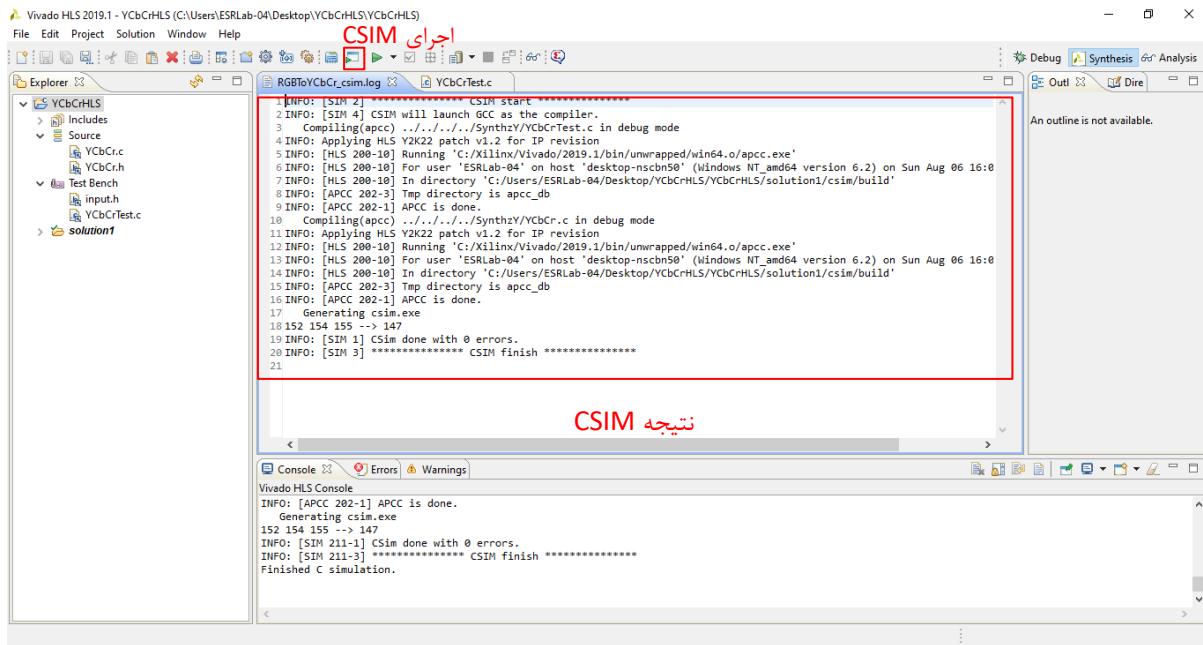
```

1 #include "YCbCr.h"
2
3
4
5 struct YCbCr RGBToYCbCr(unsigned char R, unsigned char G, unsigned char B) {
6     struct YCbCr ycbcr;
7     unsigned char Ymem = 62;
8     unsigned char Cbmem = 133;
9     unsigned char Crmem = 123;
10
11
12     if (R<R_MEMORY && R>R_MEMORY - TH && G<G_MEMORY + TH && G>G_MEMORY - TH && B<B_MEMORY + TH && B>B_MEMORY - TH)
13     {
14         ycbcr.Y = Ymem;
15         ycbcr.Cb = Cbmem;
16         ycbcr.Cr = Crmem;
17         ycbcr.f = 1;
18     }
19     else
20     {
21         ycbcr.Y = (0.257 * R) + (0.504 * G) + (0.098 * B) + 16;
22         ycbcr.Cb = ((-0.148) * R) - (0.291 * G) + (0.439 * B) + 128;
23         ycbcr.Cr = (0.439 * R) - (0.368 * G) - (0.071 * B) + 128;
24         ycbcr.f = 0;
25     }
26 }
27 }
```

شکل شماره ۵۱. برنامه نوشته به زبان C برای تبدیل فضای رنگی

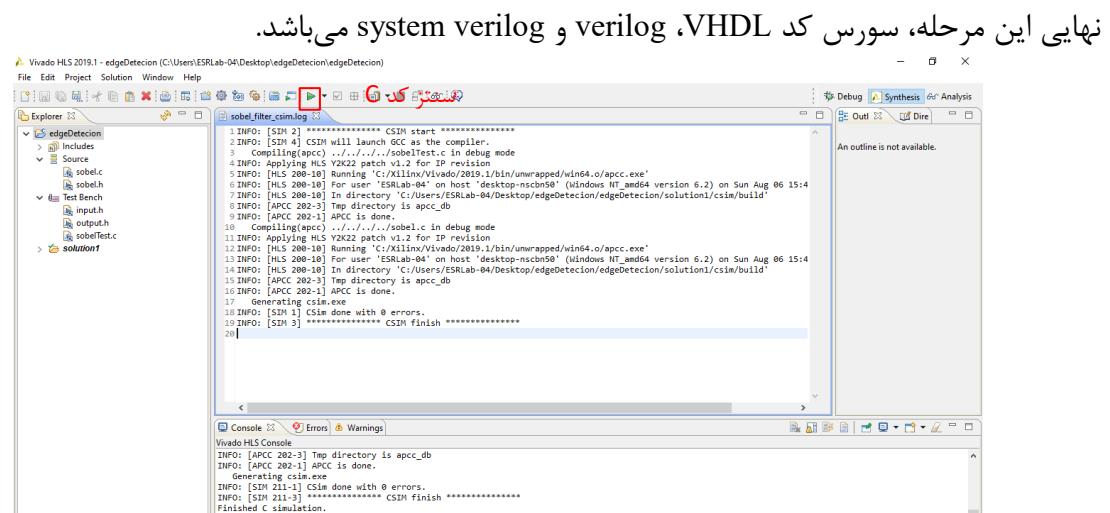
استخراج IP از کد C با ابزار HLS

ساخت خروجی با نرمافزار HLS، همانند کامپایل کردن یک برنامه، مراحل مختلفی دارد که در ادامه به شرح این مراحل می‌پردازیم. مرحله اول شبیه‌سازی کد C پروژه می‌باشد که از این مرحله با نام CSIM یاد می‌شود. برای این مرحله لازم است تا آیکون مشخص شده در شکل شماره ۵۲ را فشار دهیم، سپس خروجی‌ها و option‌هایی که لازم داریم را مشخص کنیم و در نهایت صبر کنیم تا شبیه‌سازی به اتمام برسد، شکل شماره ۵۲ نتیجه شبیه‌سازی را نشان می‌دهد. یکی از نکات مثبت نرمافزار HLS، قراردادن دیباگر در آن می‌باشد که باعث می‌شود بتوانیم در همان محیط HLS به رفع ایرادات برنامه بپردازیم.



شکل شماره ۵۲. اجرای CSIM

مرحله‌ی دوم، تبدیل کد C و سنتز آن به RTL می‌باشد. تمام مراحل بدست آوردن خروجی در نرمافزار HLS دارای چالش‌های زیادی بودند اما این مرحله به نسبت چالش‌های بیشتری داشت که در بخش بعدی به آن خواهیم پرداخت. برای سنتز RTL، باید از آیکون مشخص شده در شکل شماره ۵۳ استفاده کنیم، همچنین در صورت موفقیت آمیز بودن این مرحله، در نهایت یک گزارش شامل زمان‌بندی‌ها، منابع استفاده شده، پورت‌های ورودی و خروجی و غیره تولید خواهد شد (شکل شماره ۵۴) که امکان ذخیره‌ی آن بصورت فایل XML وجود دارد و ما آن را در قسمت نتایج مربوط به مژول تبدیل فضای رنگی قرار داده‌ایم. خروجی نهایی این مرحله، سورس کد VHDL و Verilog می‌باشد.



شکل شماره ۵۳. سنتز کد C

Synthesis Report for 'RGBToYCbCr'

General Information

Date:	Sun Aug 6 16:04:11 2023
Version:	2019.1 (Build 2552052 on Fri May 24 15:28:33 MDT 2019)
Project:	YCbCrHLS
Solution:	solution1
Product family:	spartan7
Target device:	xc7s100-fgg467-2

Performance Estimates

- Timing (ns)**
- Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.567	1.25

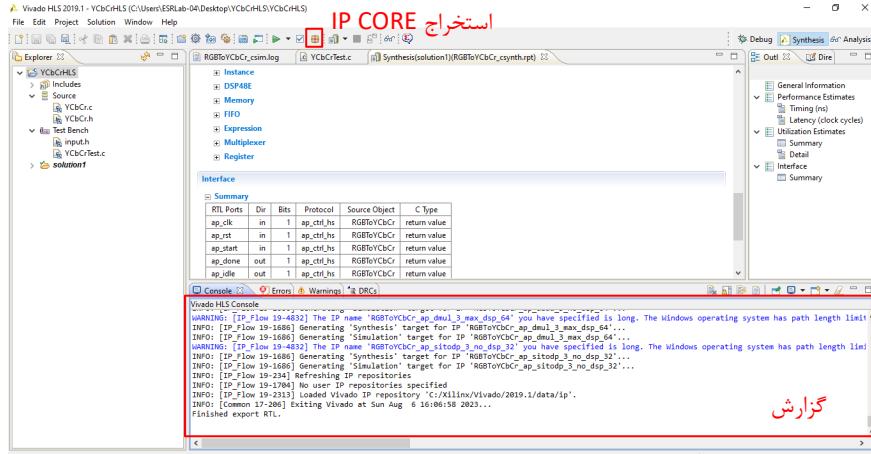
- Latency (clock cycles)**
- Summary**

Latency	Interval			
min	max	min	max	Type
26	26	26	26	none

- Detail**
- Instance**
- Loop**

شكل شماره ۵۴. خروجی سنتز کد C

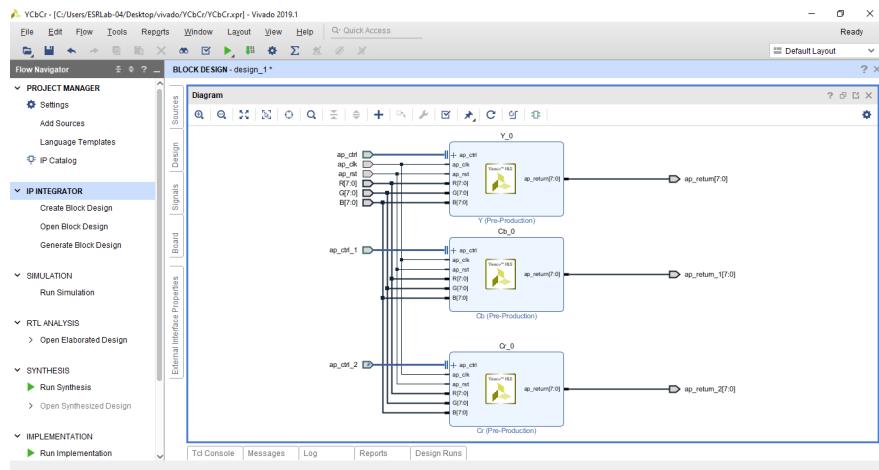
مرحله نهایی، استخراج IP جهت استفاده مستقیم در نرم افزارهایی مثل Vivado و ISE می باشد. برای استخراج IP، باید از آیکون مشخص شده در شکل شماره ۵۵ استفاده کنیم، همچنین در صورت موفقیت آمیز بودن این مرحله، پیامی مطابق با شکل شماره ۵۵ نشان داده خواهد شد.



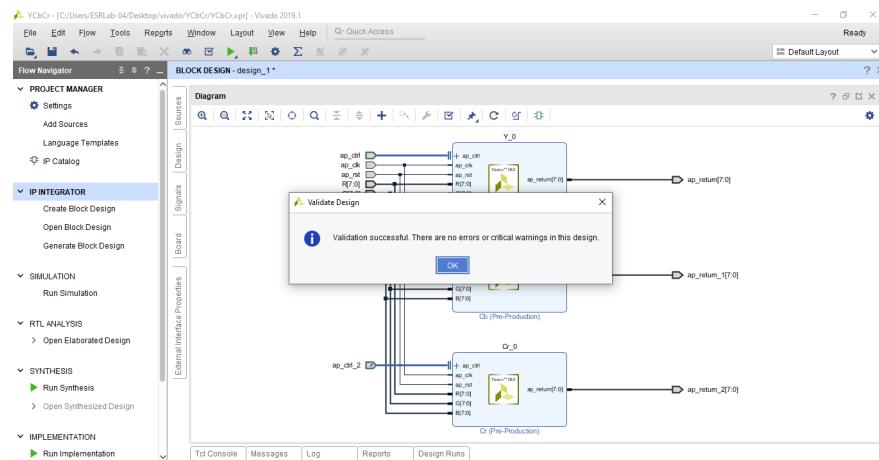
شكل شماره ۵۵. استخراج IP CORE

بعد از استخراج IP، یک پروژه جدید در نرم افزار Vivado ایجاد کرده و یک block design در آن ایجاد کردیم، سپس آدرس استخراج شده را دستی به لیست IP های نرم افزار اضافه کرده و در قدم بعدی، آنها را به block design اضافه می کنیم، بعد اضافه کردن IP های مورد نیاز، اتصالات آنها را ایجاد کرده و یک

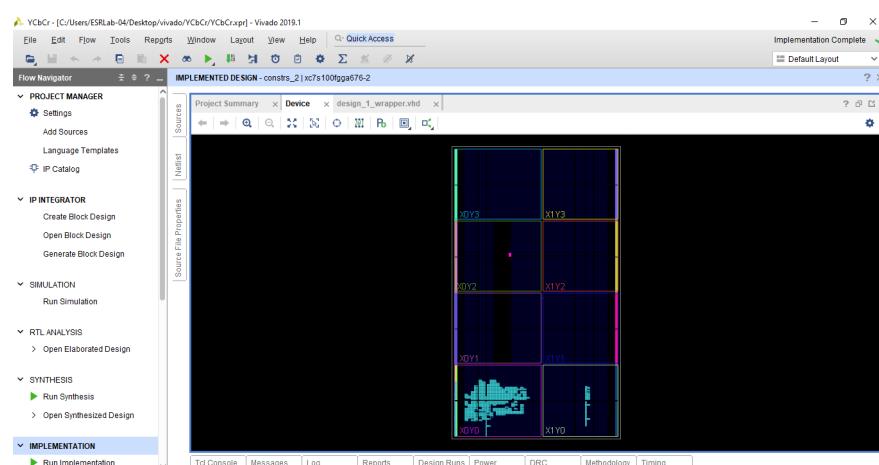
از نرم افزار گرفته در نهایت یک خروجی hdl wrapper از آن تولید می کنیم. سپس برای گرفتن گزارش توان و غیره، آن را implementation می کنیم.



شکل شماره ۵۶. ساخت Block design



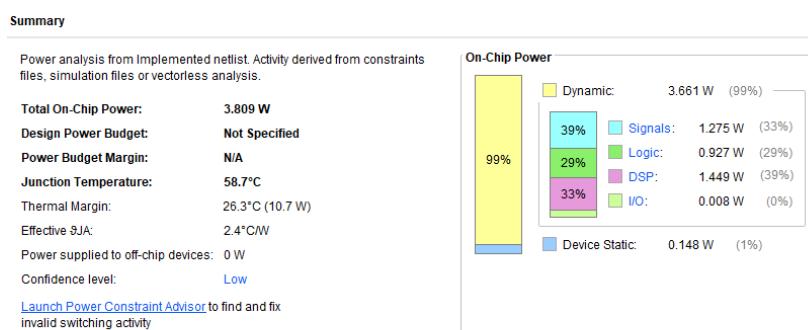
شکل شماره ۵۷. گرفتن تاییدیه طراحی



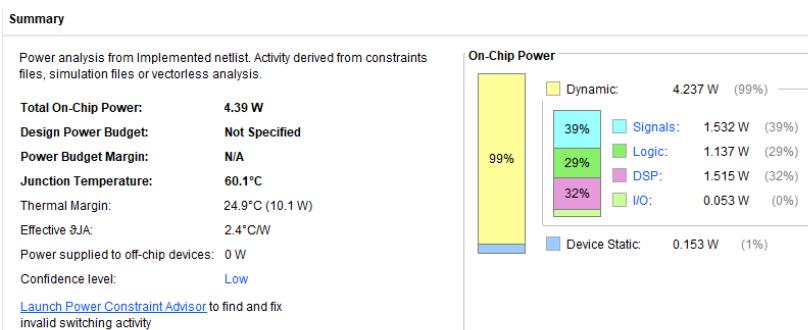
شکل شماره ۵۸. خروجی implementation طراحی

گزارش توان

بعد از اطمینان از عملکرد مازول نهایی در هر دو حالت، از نرمافزار ModelSim، فایل saif. شبیه-سازی‌ها را دریافت کرده و با استفاده از نرمافزار Vivado، توان را براساس آن‌ها تخمین زدیم. همان‌طور که شکل‌های زیر نشان می‌دهند، کل توان دینامیکی در حالت memoization نسبت به حالت no-memoization ۱۶ درصد بهبود داشته. همچنین توان مربوط به بخش logic، ۲۲ درصد و توان مربوط به بخش signals نیز حدود ۲۰ درصد بهبود داشته. میزان بهبودی توان دینامیکی در مثال تبدیل فضای رنگی نسبت به تشخیص لبه بیشتر بود، دلیل اصلی آن هم نوع محاسبات در مازول تبدیل فضای رنگی می‌باشد، زیرا همانطور که در قسمت قبل اشاره کردیم، برای تبدیل فضای رنگی نیاز به ضرب و جمع اعشاری است، همچنین تعداد ضرب‌ها و جمع‌ها نسبت به مازول تشخیص لبه نیز بیشتر می‌باشد، در واقع ما در مازول تشخیص لبه ضرب در اعداد ثابت یک و دو داشتیم و نیاز بود که جمع سه عدد، از سه عدد دیگر کم شده و قدر مطلق آن در خروجی قرار بگیرد، بنابراین انتظار داریم با استفاده از تکنیک‌های محاسبات تقریب و حافظه سازی بهبودی بیشتری حاصل شود.



شکل شماره ۵۹. توان مازول تبدیل فضای رنگی در حالت memoization



شکل شماره ۶۰. توان مازول تبدیل فضای رنگی در حالت non-memoization

گزارش کلاک و منابع

شکل‌های زیر مقدار کلاک برای تکمیل خروجی و منابع لازم برای پیاده‌سازی این مأذول را در دو نرم‌افزار Vivado و HLS نشان می‌دهد.

General Information

Date:	Sun Aug 6 16:04:11 2023
Version:	2019.1 (Build 2552052 on Fri May 24 15:28:33 MDT 2019)
Project:	YCbCrHLS
Solution:	solution1
Product family:	spartan7
Target device:	xc7s100-fgg4676-2

Performance Estimates

Timing (ns)

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.567	1.25

Latency (clock cycles)

Latency	Interval
min	max
26	26
26	26
none	

Summary

Detail

Instance

شکل شماره ۶۱. خروجی Timing در نرم‌افزار HLS

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	681	-
FIFO	-	-	-	-	-
Instance	-	25	1729	2086	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	195	-
Register	-	-	411	-	-
Total	0	25	2140	2962	0
Available	240	160	128000	64000	0
Utilization (%)	0	15	1	4	0

Detail

- Instance
- DSP48E
- Memory
- FIFO
- Expression
- Multiplexer
- Register

شکل شماره ۶۲. خروجی منابع در نرم‌افزار HLS

IMPLEMENTED DESIGN * - constns_21 xc7s100fgg4676-2

IMPLEMENTATION

- Run Implementation
- Open Implemented Design
- Constraints Wizard
- Edit Timing Constraints
- Report Timing Summary
- Report Clock Networks
- Report Clock Interaction
- Report Methodology
- Report DRC
- Report Noise
- Report Utilization
- Report Power
- Schematic

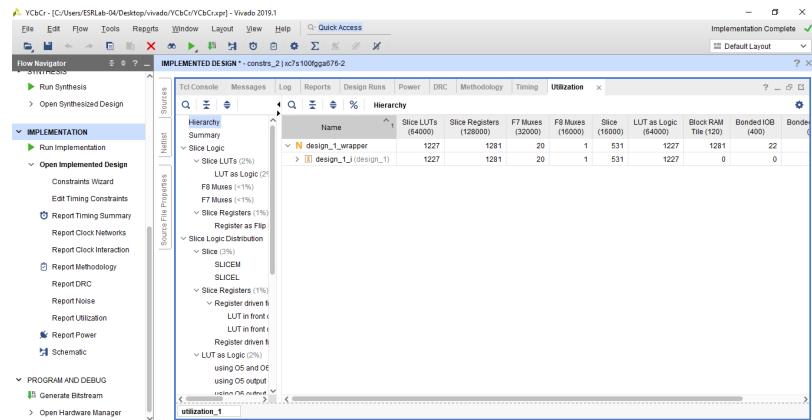
PROGRAM AND DEBUG

- Generate Bitstream
- Open Hardware Manager

Timing Check

Timing Check	Count	Worst Severity
unconstrained_internal_endpoints	2,547	High
no_clock	1,295	High
no_input_delay	26	High
no_output_delay	11	High
constant_clock	0	
pulse_width_clock	0	
multiple_clock	0	
generated_clocks	0	
loops	0	
partial_input_delay	0	
partial_output_delay	0	
latch_loops	0	

شکل شماره ۶۳. خروجی Timing در نرم‌افزار Vivado



شکل شماره ۶۴. خروجی منابع در نرم افزار Vivado

نحوه پیدا کردن مقادیری که باید در واحد memoization ذخیره شوند

بازده و عملکرد کلی روش پیاده شده بستگی به مقداری دارد که ما در واحد memoization ذخیره کرده‌ایم، هر چه این مقدار در الگوهای ورودی بیشتر تکرار شده باشد، نیاز کمتری به محاسبات برای تولید خروجی خواهیم داشت و در نتیجه توان کمتری مصرف خواهد شد، بنابراین چالش اصلی تعیین مقدار حفظ شده در حافظه می‌باشد. ما برای این منظور یک برنامه با زبان C نوشتیم. این برنامه مقداری که بیشترین ورودی‌ها (البته با در نظر گرفتن یک حد آستانه و خطاب خاطر بحث approximation) در بازه‌ی آن قرار می‌گیرند را مشخص کرده و در خروجی به ما برمی‌گرداند. همانطور که شکل زید نشان می‌دهد، بهینه‌ترین ورودی، (46, 56, 65) می‌باشد که در بازه بیست تا بیست و آن قرار گرفته است.

169 176 186 -> 5888
167 176 181 -> 5887
168 177 182 -> 5975
169 178 183 -> 6052
170 179 184 -> 6138
171 178 185 -> 6151
174 178 187 -> 6787
174 182 185 -> 6868
175 182 188 -> 6978
178 181 186 -> 7549
178 182 187 -> 7758
180 183 188 -> 8359
183 186 193 -> 8338
184 189 195 -> 8495
185 190 196 -> 8664
185 190 194 -> 8772
186 190 195 -> 8771
189 192 199 -> 9372
192 193 198 -> 9954
193 196 201 -> 10118
194 197 202 -> 10223
195 197 202 -> 10479
195 198 203 -> 10500
198 201 206 -> 10623
198 202 205 -> 10661
199 203 206 -> 10764
200 204 207 -> 10857
200 205 208 -> 10922
201 206 209 -> 11073
205 206 210 -> 11325
206 207 211 -> 11375
206 206 208 -> 11406
206 207 209 -> 11467
206 207 210 -> 11555
208 209 211 -> 11536
209 209 211 -> 11683
210 210 212 -> 11699
209 209 209 -> 11658
46 47 57 -> 14777
54 57 57 -> 16688
58 48 53 -> 17395
49 50 54 -> 18131
48 52 55 -> 18697
50 58 60 -> 18905
51 68 65 -> 19053
49 58 60 -> 19070
48 57 64 -> 19970
45 54 63 -> 19995
47 56 65 -> 20832
46 56 65 -> 20948

شکل شماره ۶۵. پیدا کردن بهینه‌ترین ورودی برای واحد memoization

نتایج شبیه‌سازی مازول تبدیل فضای رنگی

در این قسمت خروجی به دست آمده از مازول تبدیل فضای رنگی RGB به فضای رنگی YcbCr را آورده و آن را بررسی کرده‌ایم. شکل شماره ۶۶ تصویر اصلی و کاملی است که از database معرفی شده در مقاله استخراج شده و تصویر ۶۷ نیز قسمتی از تصویر اصلی است که به عنوان ورودی به مازول داده شد و ابعاد آن همانند مقاله ۳۲۰ در ۲۴۰ می‌باشد، علت انتخاب این تصویر و برش آن خود مقاله بوده که دقیقاً از همین تصویر و قسمت جدا شده استفاده کرده بود. شکل شماره ۶۸ خروجی تابع (rgb2ycbcr(RGB) می‌باشد که از آن برای مقایسه با خروجی بدست آمده از مازول استفاده کردیم. شکل ۶۹ خروجی پارامتر ۷ تصویر، شکل ۷۰ خروجی پارامتر Cb، شکل شماره ۷۱ خروجی پارامتر Cr و شکل شماره ۷۲ خروجی نهایی را نشان می‌دهد.



شکل شماره ۶۶ تصویر اصلی در فضای رنگی RGB



شکل شماره ۶۸. خروجی تابع rgb2ycbcr متب



شکل شماره ۶۷. ورودی مازول در فضای رنگی RGB



شکل شماره ۷۰. خروجی پارامتر Cb



شکل شماره ۶۹. خروجی پارامتر ۷



شکل شماره ۷۲. خروجی نهایی مازول تبدیل فضای
رنگی با استفاده از واحد memoization



شکل شماره ۷۱. خروجی پارامتر Cr

جهت مقایسه‌ی بهتر و بررسی راحت‌تر، ما تصمیم گرفتیم تمام پیکسل‌هایی که مقدارشان از واحد memoization بدست می‌آید و توسط واحد پردازش محاسبه نمی‌شوند را به رنگ مشکی نشان دهیم، بنابراین تمام پیکسل‌هایی که بازه بیست‌تایی از (46, 56, 65) قرار دارند را به (16, 128, 128) نگاشت کنیم. شکل ۷۳ خروجی پارامتر ۷ تصویر، شکل ۷۴ خروجی پارامتر Cb، شکل شماره ۷۵ خروجی پارامتر Cr و شکل شماره ۷۶ خروجی نهایی پارامتر ۷ خروجی نهایی را نشان می‌دهد.



شکل شماره ۷۴. خروجی پارامتر Cb



شکل شماره ۷۳. خروجی پارامتر ۷



شکل شماره ۷۶. خروجی نهایی مازول تبدیل فضای رنگی با استفاده از واحد memoization



شکل شماره ۷۵. خروجی پارامتر Cr

نکات تكميلی

نکته‌ی اول اين است که ما نياز به تبدیل عکس‌ها و تصاویر به مقادير عددی، ساخت آرایه ورودی برای کد C، پيدا کردن بازه‌ای که بيشترین تكرار را دارد، ساخت واحدهای ورودی و خروجی memoization داشتيم، برای راحتی و افزایش سرعت، اسکريپت‌ها و برنامه‌هایي نوشتيم که تمامی اين کارها را خودکار انجام داده و تمام فایل‌های و موارد نياز را بصورت خودکار برای ما توليد می‌کند (ما تمام فایل‌های تولیدی و مقادير مورد نياز را در پوشه‌ی مربوط به YcbCr قرار داده‌ایم).

نکته دوم، ابعاد تصوير می‌باشد، ما در اين قسمت همانند مقاله از ابعاد ۳۲۰ در ۲۴۰ استفاده کردیم، حتی برخلاف مازول تشخيص لبه که نتوانستیم data set ورودی آن را پیدا کنیم، در این بخش موفق شدیم از همان ورودی‌های مقاله با تفاوت اندک استفاده کنیم. به دلیل استفاده از ابعاد ۳۲۰ در ۲۴۰ که به زمان بسیار زیادی برای کامپایل و شبیه‌سازی test bench نیاز دارد، ما برای این مازول به شبیه‌سازی و استخراج wave form نپرداختیم.

نکته سوم این است که مقاله ذکر نکرده که آیا از بهینه‌سازها استفاده کرده یا نه و اگر استفاده کرده از چه بهینه‌سازی، همچنین مقاله نگفته که نحوه دادن ورودی به مازول و گرفتن خروجی چگونه است، همچنین مقاله هیچ اشاره‌ای به نحوه محاسبه توان نکرده بود، تمام این موارد باعث می‌شود که خروجی ما دقیقاً مطابق با خروجی مقاله نباشد.

بخش ششم : چالش‌های پیاده‌سازی

ما در این بخش به مهم‌ترین چالش‌هایی که حین پیاده‌سازی پروژه به آن برخورد کردیم را معرفی و راه حل آن‌ها را نیز شرح خواهیم داد. این پروژه به دلیل استفاده از ابزارهای بسیار جدید که آموزش‌های کمی برای آن موجود دارد، دارای چالش‌های زیاد بود که حل آن‌ها نیاز به صرف زمان بسیار زیادی داشت. بیشتر چالش‌هایی که ما در این پروژه به آن‌ها برخورد کردیم، مربوط به نرم‌افزار HLS بود. در ظاهر نرم‌افزار HLS بسیار ساده به نظر می‌رسد اما به شدت محتوای آموزش و همچنین پشتیبانی شرکت‌ها و جوامع توسعه دهنده‌گان از آن کم می‌باشد که این موضوع به شدت کار را با مشکل روبرو می‌کرد. همچنین تنها مشکل ما در نرم‌افزار Vivado، مربوط به گرفتن خروجی توان بود که در ادامه بطور کامل آن‌ها را شرح خواهیم داد.

وقوع خطا هنگام شبیه‌سازی کد C در نرم‌افزار HLS

اولین مشکلی که به آن برخورد کردیم، وقوع خطا هنگان شبیه‌سازی کد C در نرم‌افزار HLS می‌بود که یک خطای نامفهوم در خط فرمان برنامه برای تولید می‌کرد و دقیقاً مشکل را بیان نمی‌کرد، این در حالی بود که کد ما درست بود و کاملاً توسط کامپایلر gcc، کامپایل و اجرا می‌شد. بعد از جست و جوی فراوان به این نتیجه رسیدیم که باید یک فایل h. همانم با فایلی که حاوی تابع اصلی است (تابعی که می‌خواهیم درنهایت سنتز کنیم) بسازیم که محتویات آن باید حتماً بصورت زیر باشد:

```
1 #ifndef _SOBEL_H_
2 #define _SOBEL_H_
3
4 #include "stdio.h"
5
6 void sobel_filter(unsigned char in[512*512], unsigned char out1[512*512]);
7
8#endif
```

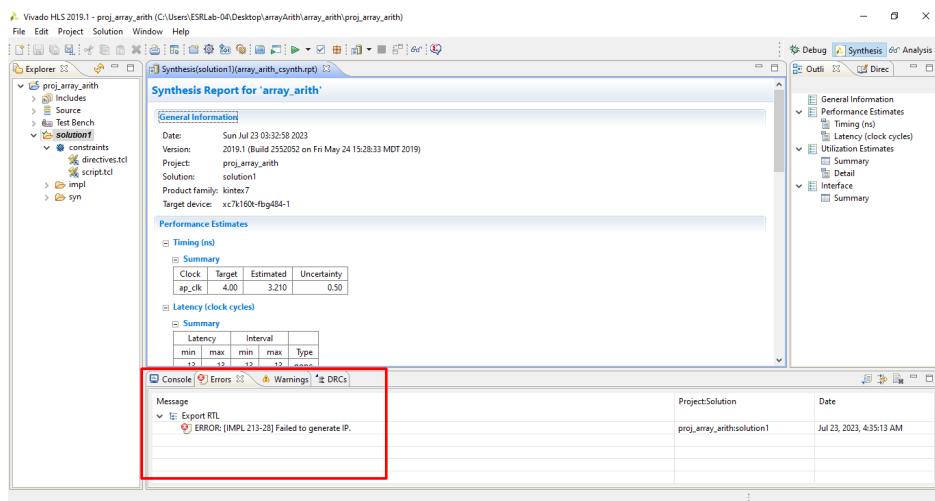
شکل شماره ۷۷. محتویات فایل header کد تابع اصلی

نکته مهمی که باید به آن اشاره کنیم، این هست که در فایل تست کد C، در صورت کامپایل کردن با کامپایلرهای رایج مثل gcc، حتماً باید فایل تابع اصلی را فراخوانی کنیم، یعنی مثلاً در تشخیص لبه، اگر نام فایل تابع اصلی "sobel.c" باشد، در فایل تست "sobelTest.c" که تابع main در آن قرار دارد، درصورت

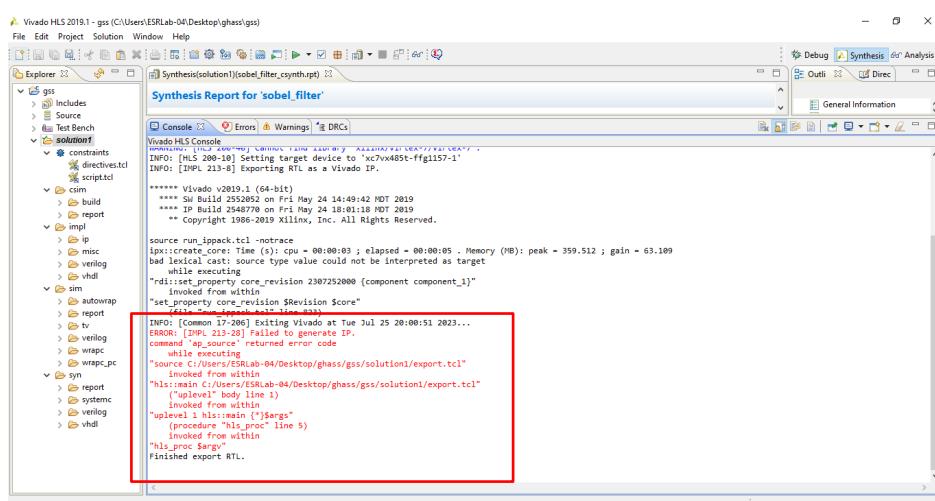
کامپایل و تست کردن برنامه بصورت عادی، باید فایل "sobel.c" را فراخوانی کنیم، اما در صورت شبیه‌سازی کد C در نرم‌افزار HLS، باید حتماً فراخوانی فایل "sobel.c" را پاک کنیم تا شبیه‌سازی موفقیت‌آمیز باشد.

وقوع خطا هنگام استخراج IP در نرم‌افزار HLS

متاسفانه یک مشکل رایج در نرم‌افزار HLS Vivado، وقوع خطا هنگام استخراج IP می‌باشد که یک پیام نامفهوم همانند شکل زیر در محیط خط فرمان این نرم‌افزار تولید می‌کند، این مشکل در نسخه‌های جدید این نرم‌افزار حل شده است اما همچنان در نسخه‌های قدیمی باقی مانده و هیچ بروزرسانی از طرف شرکت سازنده برای آن ارائه نشده است.

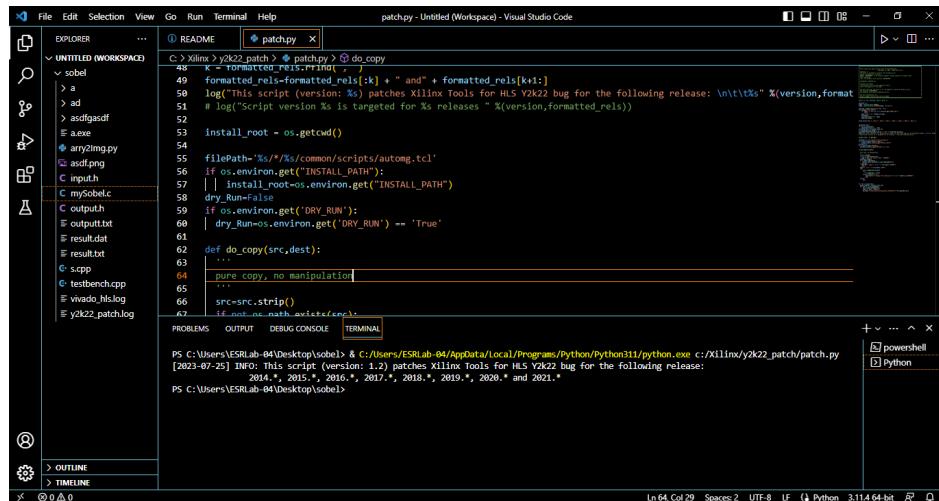


شکل شماره ۷۸. خطا هنگام استخراج IP



شکل شماره ۷۹. خطا هنگام استخراج RTL

برای حل این مشکل، یک patch از طرف جامعه توسعه‌دهندگان در گیت‌هاب منتشر شده بود، اما متأسفانه این patch برای نسخه‌ای که ما از آن استفاده می‌کریم قابل استفاده نبود، به همین دلیل ما این patch را پورت کرد و استفاده کردیم. برای استفاده از این patch کافی است آن را به همراه سایر محتویات پوشه در محلی که نرم‌افزار HLS در آن قرار گرفته، کپی کرده و با توجه به نسخه نرم‌افزار HLS، دستور و پارامترهای اجرای آن را در محیط CMD ویندوز وارد کنیم. شکل‌های زیر محتویات این patch به همراه نحوه اجرای آن را نشان می‌دهند.

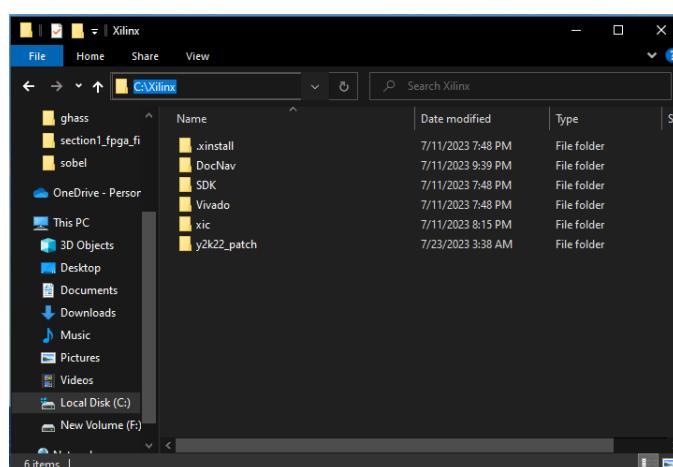


```

File Edits Selection View Go Run Terminal Help
patchpy - Untitled (Workspace) - Visual Studio Code
EXPLORER README patchpy do_copy
UNTITLED (WORKSPACE)
sobel
> a
> ad
> asdfgadf
> a.exe
> any2imgpy
> asdf.png
> C inputh
> C mySobel.c
> C outputh
> C outputtxt
> C resultdat
> C resulttxt
> C s.cpp
> C testbench.cpp
> E vivado.hsl.log
> E y2k22.patch.log
C:\Xilinx\y2k22_patch> patchpy > do_copy
48     K = formatted_rels.print()
49     formatted_rels.formatted_rels[sk] + " and " + formatted_rels[k+i]
50     log("This script (version: %s) patches Xilinx tools for HLS Y2K22 bug for the following release: \n\t\t%s" % (version,format
51     # log("Script version %s is targeted for Xs releases %s" % (version,formatted_rels))
52
53     install_root = os.getcwd()
54
55     filePath= '%s/common/scripts/automg.tcl'
56     if os.environ.get("INSTALL_PATH"):
57         [ | install_root=os.environ.get("INSTALL_PATH")
58     dry_Run=False
59     if os.environ.get('DRY_RUN'):
60         [ | dry_Run=os.environ.get('DRY_RUN') == 'True'
61
62     def do_copy(src,dest):
63         ...
64         pure_copy_no_manipulation()
65
66     src=src.strip()
67     if not os.path.exists(dest):
68
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\ESRLab-04\Desktop\sobel> & C:/Users/ESRLab-04/AppData/Local/Programs/Python/Python311/python.exe c:/Xilinx/y2k22_patch/patch.py
[2023-07-25] INFO: This script (version: 1.2) patches Xilinx Tools for HLS Y2K22 bug for the following release:
2014.*., 2015.*., 2016.*., 2017.*., 2018.*., 2019.*., 2020.* and 2021.*.
PS C:\Users\ESRLab-04\Desktop\sobel>

```

شکل شماره ۸۰. استفاده شده برای حل خطای استخراج IP



شکل شماره ۸۱. محل کپی کردن فایل patch

```

C:\ Command Prompt
Microsoft Windows [Version 10.0.19045.3208]
(c) Microsoft Corporation. All rights reserved.

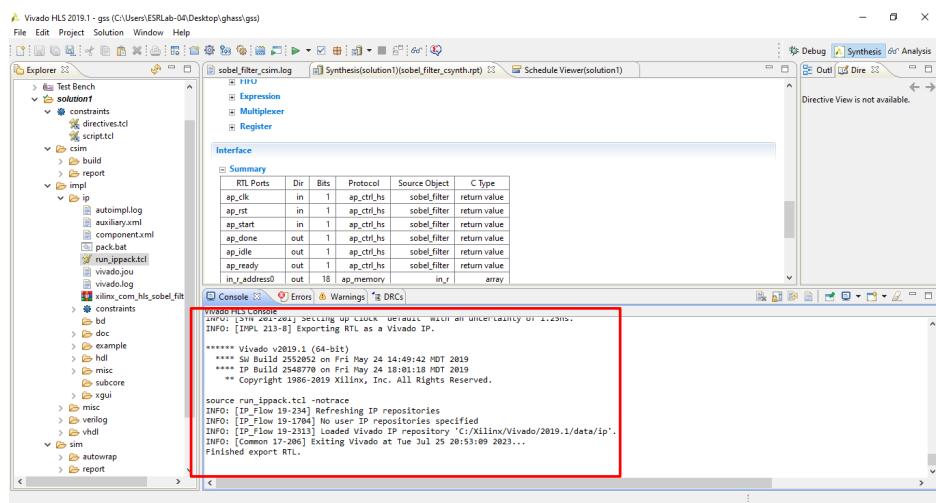
C:\Users\ESRLab-04>cd ..
C:\Users>cd ..
C:\>cd Xilinx

C:\Xilinx>python .\patch.py
[2023-07-25] INFO: This script (version: 1.2) patches Xilinx Tools for HLS Y2k22 bug for the following release:
  2014.* , 2015.* , 2016.* , 2017.* , 2018.* , 2019.* , 2020.* and 2021.*
[2023-07-25] UPDATE: C:\Xilinx\Vivado\2019.1\common\scripts
[2023-07-25] COPY: C:\Xilinx\y2k22_patch\automg_patch_20220104.tcl to C:\Xilinx\Vivado\2019.1\common\scripts\automg_pat
ch_20220104.tcl

C:\Xilinx>

```

شکل شماره ۸۲. نحوه اجرای patch



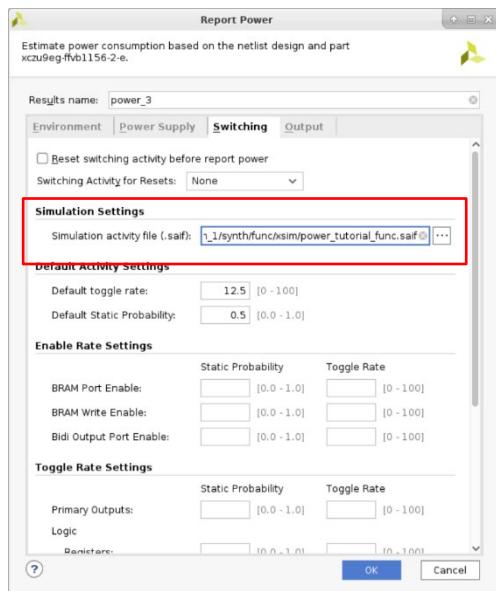
شکل شماره ۸۳. رفع خطای استخراج IP

استخراج توان مازول براساس ورودی‌های خاص

در این پژوهه، برای اثبات ایده و روش پیشنهادی، لازم بود تا مازول‌ها را بر اساس ورودی‌های خاص و از پیش تعیین شده اجرا و توان مصرف شده در این اجرا و سناریوی خاص را استخراج و گزارش کنیم، زیرا بر اساس ورودی‌ها و مقادیر ثبت شده در واحد memoization تصمیم گرفته می‌شود تا کدام واحدها فعال و اجرا شوند و توان مصرف کنند. در اولین تلاش، مشاهده کردیم که توان حالت memoization نسبت به حالت non-memoization بیشتر شده که این کاملاً برخلاف انتظار ما و نتایج مقاله بود، دلیل این موضوع، عدم آشنایی بندۀ با مکانیزم محاسبه توان در نرمافزار Vivado بود، بعد از کمی جست و جو متوجه شدیم که نرمافزار Vivado، بدون در نظر گرفتن ورودی‌های داده شده در فایل Test bench، یک نرخ سوئیچینگ بدون

در نظر گرفتن حتی منطق کلی مازول به تمام سیگنال‌ها می‌دهد و توان کل را براساس آن تخمین می‌زند. برای این منظور، در ابتدا باید فایل saif را توسط نرم‌افزار ModelSim تولید کرده و سپس آن را به نرم‌افزار vivado داده تا سناریوی ورودی‌ها و تمام سیگنال‌ها مطابق سناریوی Test bench شود و توان دینامیکی به ازای ورودی‌های مشخص محاسبه شود. مشکل اصلی نحوه تولید فایل saif بود، این فایل و فایل vcd را براحتی می‌توان با اضافه کردن دستوراتی به Test bench که با وریلاگ نوشته شده است، بدست آورد، اما ما با VHDL نوشته شده بود و امکان استفاده از آن دستورات برای ما وجود نداشت، یک را بازنویسی Test bench با زبان وریلاگ بود ولی راه ساده‌تر، استفاده از محیط Transcript نرم‌افزار ModelSim بازنویسی Test bench با استفاده از دستورات زیر، فایل saif تولید شده و می‌توان از آن برای محاسبه وارد کردن دستورات زیر بود. با استفاده از دستورات زیر، فایل saif تولید شده و می‌توان از آن برای محاسبه توان استفاده کرد.

- ❖ power add memoization_TB/memoization/*
- ❖ run 4600ns
- ❖ power report -all -bsaif routed.saif
- ❖ quit



شکل شماره ۸۴. نحوه استفاده از فایل .saif.

بخش هفتم : مقایسه نتایج پیاده‌سازی با مقاله

در این بخش به مقایسه و جمع‌بندی نتایج به دست آمده از شبیه‌سازی با نتایج مطرح شده در مقاله خواهیم پرداخت و با ذکر دلیل، این دو را بررسی خواهیم کرد. برای طولانی نشدن گزارش، ما از آوردن تصاویر خروجی‌ها خودداری کرده ولی آن‌ها را پوششی Result قرار دادیم تا در صورت نیاز به آن‌جا مراجعه صورت گیرد.

جدول‌های ۲، ۳، ۴، ۵ و ۶ مقاله : دلیل مطرح کردن ایده

این جدول‌ها در واقع نتایجی هستند که از مقالات [2] و [3] استخراج شده‌اند و نویسنده‌گان با استناد به نتایج این مقالات، این ایده را مطرح کردند که استفاده از محاسبات تقریبی همراه با حافظه‌سازی، می‌تواند منجر به بهبود توان گردد؛ همچنین با استفاده از نتایج همین مقالات، بررسی کردند که میزان بهبودی حاصل شده از روش محاسبات تقریبی همراه با حافظه‌سازی، از روش‌های مرسوم مثل UDM و Truncated و غیره benchmark شده در این جداول و برای کرنل‌ها و کاربردهای مرسوم مثل ضرب نقطه‌ای، Norm و غیره و غیره، بیشتر است. از طرفی این مقاله عنوان کرده که تبدیل فضای رنگی و تشخیص لبه، دو تا از پرکاربردترین کارهایی هستند که در حوزه پردازش تصویر انجام می‌شوند، بنابراین این مقاله می‌خواهد با معرفی دو کار تشخیص لبه و تبدیل فضای رنگی با استفاده از روش محاسبات تقریبی به همراه حافظه‌سازی به عنوان case study، کارایی این دو روش را اثبات کند و نتایج بدست آمده از مقالات [2] و [3] را تایید کند.

Benchmark	Slice Registers	LUTs	Memory Resource
Dot Product (a)	3333	5703	—
M-Dot Product (b)	3346	5838	Block RAM
Dot Product-Truncated (c)	3049	5082	—
%Overhead: (b) over (a)	0.39	2.36	—
PSPA	0.011		
L2-Norm (a)	626	947	—
M-L2-Norm (b)	630	958	Slice FF
L2-Norm-Truncated (c)	609	918	—
%Overhead: (b) over (a)	0.63	2.01	—
PSPA	0.084		
SAD (a)	677	728	—
M-SAD (b)	685	757	Slice FF
SAD-Truncated (c)	652	713	—
%Overhead: (b) over (a)	1.18	3.9	—
PSPA	0.017		

جدول شماره ۴ مقاله. میزان سربار مساحت در کرنل‌های رایج با استفاده از محاسبات تقریبی

Benchmark	Dynamic Power (mW)	Clock Period (ns)	MSE Achieved (Specified)	Execution Cycles
Dot Product (a)	382.43	6.07	-	3605
M-Dot Product (b)	287.37	6.045	9.8% (10%)	3223
Dot Product-Truncated (c)	344.47	6.099	9.9% (10%)	3605
%Saving: (b) over (a)	24.85	-	-	-
L2-Norm (a)	35.77	4.867	-	3604
M-L2-Norm (b)	27.83	4.886	7.7% (8%)	3102
L2-Norm-Truncated (c)	31.7	4.802	8.1% (8%)	-
% Saving: (b) over (a)	22.19	-	-	-
SAD (a)	33.67	3.241	-	3603
M-SAD (b)	29.74	3.545	8.9% (10%)	3293
SAD-Truncated (c)	32.01	3.034	9.7% (10%)	3603
%Saving: (b) over (a)	11.67	-	-	-

جدول شماره ۲ مقاله. میزان بهبود توان دینامیکی در کرنل‌های رایج با استفاده از محاسبات تقریبی

UCI Database	Dynamic Power(mW)	Clock Period(ns)	No. of incorrect classification	Power Saving(%)
Skin	800.75	8.952	-	-
M-Skin	650.18	9.233	50/10000	19.28
Skin_7b	730.23	8.882	125/10000	9.31
Skin_6b	700.31	8.241	171/10000	12.92
Iris	100.12	7.892	-	-
M-Iris	85.13	8.112	5/100	14.97
Iris_7b	97.33	7.532	9/100	2.78
Iris_6b	93.29	7.114	17/100	6.82

جدول شماره ۳ مقاله. میزان بهبود توان دینامیکی در کاربردهای رایج با استفاده از حافظه‌سازی SVM

Benchmark	Slice Registers	LUTs	Memory Resource
Dot Product (a)	3333	5703	-
M-Dot Product (b)	3367	5880	Block RAM
Dot Product-Truncated (c)	3049	5082	-
% Overhead: (b) over (a)	1.02	3.1	-
PSPA	0.066		
L2-Norm (a)	626	947	-
M-L2-Norm (b)	644	986	Block RAM
L2-Norm-Truncated (c)	609	918	-
%Overhead: (b) over (a)	2.87	4.11	-
PSPA	0.022		
SAD (a)	677	728	-
M-SAD (b)	698	766	Slice FF
SAD-Truncated (c)	652	713	-
%Overhead: (b) over (a)	3.1	5.21	-
PSPA	0.01		

جدول شماره ۶ مقاله. میزان سربار مساحت در کرنل‌های رایج با استفاده از حافظه‌سازی

Benchmark	Dynamic Power (mW @ 100 MHz)	Clock Period (ns)	MSE Achieved (Specified)
Dot Product (a)	392.41	6.07	-
M-Dot Product(b1)	310.12	5.712	9.2% (10%)
Dot Product Truncated (c)	350.72	6.099	9.6% (10%)
%Saving: (b1) over (a)	20.97	-	-
L2-Norm (a)	40.67	4.867	-
M-L2-Norm(b1)	32.65	5.21	7.1% (8%)
L2-Norm-Truncated (c)	36.63	4.802	7.6% (8%)
% Saving (b1) over (a)	19.71	-	-
SAD (a)	35.72	3.241	-
M-SAD(b1)	31.83	3.892	8.6% (10%)
SAD-Truncated (c)	34.07	3.034	9.2% (10%)
%Saving: (b) over (a)	10.89	-	-

جدول شماره ۵ مقاله. میزان بهبود توان دینامیکی در کرنل‌های رایج با استفاده از حافظه‌سازی

جدول‌های ۷ و ۸ مقاله : توان مصرفی، معیار PSPA و سرباز سخت‌افزاری تشخیص لبه

جدول‌های ۷ و ۸ مقاله، نتایج بدست آمده در پیاده‌سازی تشخیص لبه با استفاده از محاسبات تقریبی و حافظه‌سازی توسط نویسنده‌گان مقاله را نشان می‌دهند، همچنین جدول‌های شماره ۲ و ۳، نتایج بدست آمده از مژول تشخیص لبه توسط ما را نشان می‌دهد، قبل از مقایسه نتایج، لازم است ذکر کنیم این مقاله از تصاویری با عنوان MT. WILSON IMAGE از مقاله استفاده کرده که لینک آن را در قسمت منابع آورده بود، اما لینک معرفی شده منقضی شده بود و ما نتوانستیم از همان ورودی مقاله در پیاده‌سازی خود استفاده کنیم. موضوع بعدی بحث ابعاد تصاویر است، مقاله از ابعاد ۳۲۰ در ۲۴۰ استفاده کرده، ما هم در ابتدا خواستیم از ابعاد ۲۵۶ در ۲۵۶ استفاده کنیم، اما به دلیل اینکه ما برخلاف مقاله که بصورت واقعی پیاده‌سازی کرده بودند، شبیه‌سازی می‌کنیم، زمان بسیار زیادی برای کامپایل و شبیه‌سازی latest bench مورد نیاز بود، به همین

دلیل تصمیم گرفتیم از ابعاد ۶۴ در ۶۴ و ۱۶ در ۱۶ استفاده کنیم. همچنین کار خوبی که ما نسبت به مقاله انجام دادیم، پیاده‌سازی مازول تشخیص لبه برای دو ابعاد مختلف یعنی ۱۶ در ۱۶ و ۶۴ در ۶۴ بود، ما در واقع خواستیم میزان بهبودی توان دینامکی و میزان سربار مساحت را بر اساس تغییر ابعاد مشخص کنیم، در جدول شماره ۲ که نتایج پیاده‌سازی ما را نشان می‌دهد، تصاویر ۱ و ۳، عکسی با عنوان elaine و تصاویر ۲ و ۴، عکسی با عنوان lena هستند، همچنین ابعاد تصاویر ۱ و ۲، ۱۶ در ۱۶ و ابعاد تصاویر ۳ و ۴، ۶۴ در ۶۴ می‌باشد. نکته‌ی بعدی این است که مقاله هیچ اشاره‌ای نکرده که از کدام الگوریتم تشخیص لبه استفاده کرده است. الگوریتم تشخیص لبه‌ای که ما استفاده کردیم، الگوریتم Sobel می‌باشد که از دقت خوبی برخودار است. همچنین این مقاله هیچ اشاره‌ای نکرده که آیا از بهینه‌سازهای FPGA استفاده کرده یا خیر. تمام موارد بالا باعث می‌شوند تا خروجی ما و مقاله یکسان نباشد و با هم اختلاف داشته باشند.

همانطور که جدول‌های زیر نشان می‌دهند، استفاده از روش محاسبات تقریبی همراه با حافظه‌سازی، باعث بهبود توان دینامیکی می‌شود، اما از طرفی استفاده از این دو روش، باعث سربار سختافزاری شده و به منابع بیشتری مثل حافظه و غیره نیاز دارد. بهترین معیار برای مقایسه، معیار PSPA می‌باشد، این معیار از رابطه زیر بدست می‌آید:

$$PSPA = \frac{\text{Dynamic Power Saving}}{\text{Area Overhead}}$$

Image	Dynamic Power (mW): No memoization	Dynamic Power (mW): Memoization	Dynamic Power Saving
1	15.76	13.08	17
2	15.58	12.91	17.13
3	16.67	14.06	15.65
4	15.43	12.75	17.36
Geo Mean	15.85	13.19	16.77

جدول شماره ۷ مقاله. میزان بهبود توان دینامیکی بدست آمده در تشخیص لبه با استفاده از محاسبات تقریبی و حافظه‌سازی

Image	Dynamic Power (mW): No memoization	Dynamic Power(mW): No memoization	Dynamic Power Saving
1	48	39	9
2	53	48	5
3	173	71	102
4	179	69	110

جدول شماره ۲. میزان بهبود توان دینامیکی بدست آمده در تشخیص لبه با استفاده از محاسبات تقریبی و حافظه‌سازی

Design	Slice FF	LUT
No memoization (HLS Synthesized Block Only)	10	195
Static Memoized Architecture	25	215
Overhead	15	20
PSPA	0.058	

جدول شماره ۸ مقاله. میزان سربار مساحت و PSPA با استفاده از محاسبات تقریبی و حافظه‌سازی

Design	Slice FF	LUT
No memoization (HLS Synthesized Block Only)	76	586
Static Memoized Architecture	188	651
Overhead	112	65
PSPA		0.0621

جدول شماره ۳. میزان سربار مساحت و PSPA با استفاده از محاسبات تقریبی و حافظه‌سازی

همانطور که نتایج بالا نشان می‌دهد، در حالت memoization نسبت به حالت بدون memoization حدود ۲۳ درصد بهبودی حاصل شده که نزدیک به عدد ذکر شده در مقاله یعنی ۲۰ درصد بوده و ایده مطرح شده در مقاله را کاملاً تایید می‌کند. همچنین از نظر معیار PSPA، پروژه پیاده شده توسط ما حتی از مقاله بهتر عمل کرده و به ازای مساحت مشخص، میزان بهبودی بیشتری به همراه دارد.

شکل‌های ۹ و ۱۰ مقاله : خروجی ماژول تشخیص لبه

این مقاله جهت بررسی خروجی ماژول پیاده‌سازی شده با روش محاسبات تقریبی و روش حافظه‌سازی و مقایسه آن با خروجی ماژول‌های تشخیص لبه که توسط روش‌های دیگر مثل Data path truncation UDM پیاده‌سازی شدند، شکل‌های شماره ۹ و ۱۰ را آورده و در جدول‌های ۹ و ۱۰ آن‌ها را از نظر توان و مساحت مقایسه کرده است. ما برای جلوگیری از طولانی شدن گزارش، از آوردن این جداول خودداری کردیم، اما همانطور که انتظار داشتیم و در قسمت قبل هم به طور کامل دلیل آن را شرح دادیم، روش محاسبات تقریبی و حافظه سازی، به مرتب بهتر از دو روش UDM و Data path truncation عمل کرده و چون خروجی نهایی پروژه ما، هم از نظر میزان بهبودی در توان دینامیکی و هم از نظر معیار PSPA مشابه خروجی مقاله بوده، می‌توانیم بگوییم پروژه پیاده‌سازی شده توسط ما نیز نسبت به دو روش UDM و Data path truncation بهینه‌تر عمل خواهد کرد.



شکل شماره ۹ مقاله. خروجی ماژول تشخیص لبه پیاده شده توسط مقاله برای تصاویر Mt. Wilson



شکل شماره ۸۵. خروجی مازول تشخیص لبه برای تصویر elaine با ابعاد ۲۵۶ در ۲۵۶



شکل شماره ۸۶. خروجی مازول تشخیص لبه برای تصویر lena با ابعاد ۲۵۶ در ۲۵۶

جدول های ۱۱ و ۱۲ مقاله : توان مصرفی، معیار PSPA و سرباز سخت افزاری تبدیل فضای رنگی

جدول شماره ۴، تلفیقی از جدول شماره ۱۱ مقاله اصلی (که شامل سرباز سخت افزاری، توان دینامیکی مصرف شده، میزان بهبود توان دینامیکی، PSNR و تعداد سیکل لازم برای اجرا می باشد) و خروجی های بدست آمده توسط پروژه پیاده سازی شده توسط ما می باشد. رنگ آبی خروجی مقاله اصلی، رنگ مشکی خروجی مقالات ابتدایی و مرجع، رنگ قرمز خروجی بدست آمده توسط پروژه پیاده سازی شده توسط ما می باشد. برخلاف تشخیص لبه، ما در این قسمت توانستیم تصویری که این مقاله از آن به عنوان ورودی

استفاده کرده بود را پیدا کنیم، بنابراین ورودی ما تقریباً شبیه ورودی مقاله اصلی می‌باشد. اندازه تصویر ورودی، ۳۲۰ در ۲۴۰ می‌باشد. به دلیل تفاوت در مقدار حد آستانه و همچین نحوه دریافت ورودی توسط مژول و پیاده‌سازی عملی توسط مقاله (در مقابل شبیه‌سازی توسط ما)، مقادیری که ما برای توان و منابع سخت‌افزاری بدست آورده‌یم، تا حدودی با مقادیر بدست آمده توسط مقاله اصلی تفاوت دارد. دلیل اصلی دیگر این تفاوت، مشخص نبودن پیکسل ذخیره شده در واحد Memoization مژول مقاله می‌باشد، همانطور که قبل اگفته‌یم، این پیکسل و مقدار آستانه اثر مهمی در توان دینامیکی دارد و انتخاب آن بسیار مهم می‌باشد و ما برای رسیدن به بهترین حالت، یک برنامه نوشته‌ی ترین پیکسل را انتخاب کنیم که آن را بطور کامل در بخش پنجم شرح دادیم. اما با این وجود برای مقایسه‌ی بهتر و درست بین طراحی‌های مختلف، از معیار کنترل کیفیت PSNR استفاده شده تا کارایی هر طرح مجزا از فرض‌های اولیه آن مشخص شود.

Design	Area(LUT, DSP)	Dynamic Power (mW)	Dynamic Power Saving(%)	PSNR (dB)	Execution Cycles
Original (LUT)	495, 0	21.75	-	-	255628
Memoization (LUT)	519, 0	19.32	11.17	65.12	255345
UDM	487, 0	21.02	5.4	70.39	255628
Original (LUT, DSP)	54, 10	17.02	-	-	255628
Memoization (LUT, DSP)	78, 10	15.74	7.5	62.01	255345
Trunc-Case1	46, 9	16.06	5.6	55.96	255628
Trunc-Case2	50, 9	16.6	2.4	57.75	255628
Trunc-Case3	39, 9	15.76	7.4	55.96	255628
My design	(681,0)	18.9	12.87	60.25	153616

جدول شماره ۴. میزان مصرف سریار سخت‌افزاری، توان دینامیکی، PSNR و تعداد سیکل در تبدیل فضای رنگی به RGB در طراحی‌های مختلف (رنگ آبی خروجی مقاله اصلی، رنگ قرمز خروجی طرح ما می‌باشد).

همانطور که جدول شماره ۴ نشان می‌دهد، طرح ما نسبت به سایر طرح‌ها، با وجود هزینه‌ی سخت-افزاری بیشتر، اما در بحث میزان کاهش توان دینامیکی و معیار PSNR، بهتر از روش‌های اولیه عمل می‌کند، اما با وجود اینکه در کل توان دینامیکی کمتری نسبت به روش memoization مقاله اصلی مصرف می‌کند، اما از نظر معیار PSNR نسبت به مقاله اصلی، بدتر عمل کرده و PSNR پروژه ما کمتر شده است. در جدول بعدی

بررسی می کنیم آیا این هزینه‌ی سختافزاری به صرفه است یا خیر. این موضوع با استفاده از پارامتر PSPA مشخص می‌شود، همانطور که در قسمت قبل اشاره کردیم، PSPA از رابطه زیر بدست می‌آید:

$$PSPA = \frac{\text{Dynamic Power Saving}}{\text{Area Overhead}}$$

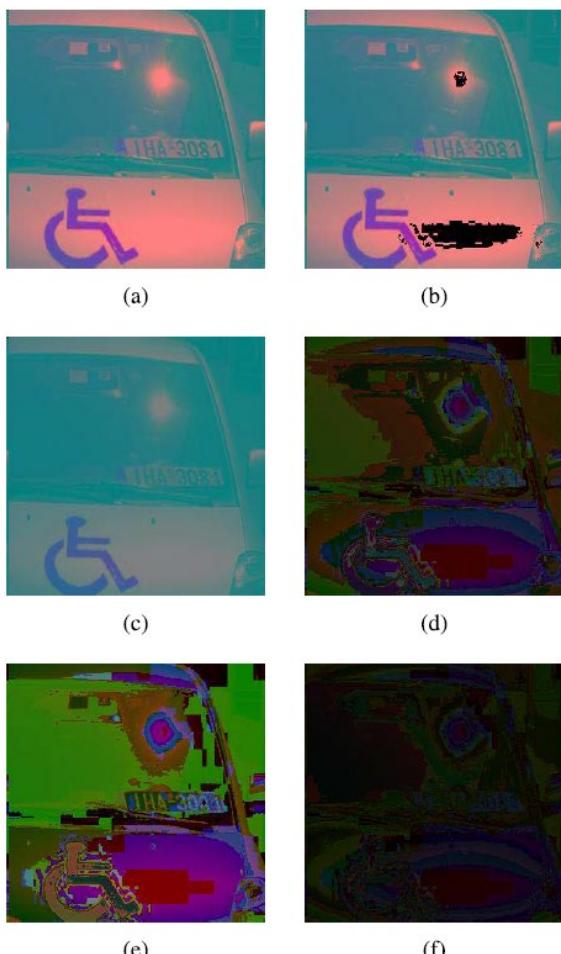
با توجه به رابطه بالا و مقادیر بدست آمده از شبیه‌سازی و نرم‌افزار Vivado، مقدار محاسبه شده برای PSPA برابر با 0.0083 شد که نسبت به سایر طرح‌ها بهتر بود. بنابراین می‌توانیم نتیجه بگیریم که هزینه‌ی سختافزاری پروژه ما، نسبت به کاهش توانی که برای ما به همراه دارد، به صرفه می‌باشد. شاید این سوال مطرح شود که علت این موضوع پیاده‌سازی عملی توسط سایر طرح‌ها و شبیه‌سازی توسط ما باشد، لازم به ذکر است که خود مقاله مقادیر جدول بالا از شبیه‌سازی بدست آورده و به این موضوع در متن مقاله اشاره کرده است.

Design	Slice FF	LUT	DSP Block
No memoization (LUT)	256	495	0
Dynamic Memoized Architecture (LUT)	312	519	0
Overhead	56	24	0
PSPA	0.005		-
No memoization (DSP block based)	61	54	10
Dynamic Memoized Architecture (DSP block based)	69	78	10
Overhead	8	24	0
PSPA	0.006		-
My Desing	411	681	0
Overhead	155	186	0
PSPA	0.0083		-

جدول شماره ۵. میزان سربار مساحت و PSPA با استفاده از محاسبات تقریبی و حافظه‌سازی در تبدیل فضای رنگی RGB به YcbCr در طراحی‌های مختلف (رنگ قرمز مقادیر مربوط به طرح ما می‌باشد)

شکل ۱۳ مقاله : خروجی مازول تبدیل فضای رنگی

این مقاله جهت بررسی خروجی مازول پیاده‌سازی شده با روش محاسبات تقریبی و حافظه‌سازی و UDM و Data path truncation مقایسه آن با خروجی مازول‌های تشخیص لبه که توسط روش‌های دیگر مثل پیکسل‌هایی پیاده‌سازی شدند، شکل شماره ۱۳ را آورده است، همچنین برای دادن درک مناسب به خواننده، پیکسل‌هایی که مقادیر آن‌ها از واحد memoization بدست آمده را با رنگ مشکی مشخص کرده است. ما هم جهت مقایسه خروجی خود با خروجی مقاله، شکل‌های ۸۷ و ۸۸ را در کنار شکل شماره ۱۳ مقاله آورده‌ایم. شکل شماره ۸۷ خروجی اصلی مازول پیاده‌سازی شده و در شکل شماره ۸۸ برای دادن درک مناسب، پیکسل‌هایی که مقادیر آن‌ها از واحد memoization بدست آمده را با رنگ مشکی مشخص کرده‌ایم.



شکل شماره ۱۳ مقاله. خروجی مازول تبدیل فضای رنگی RGB به Truncation- (d).UDM (c).Memoization (b) اصلی. (a).YCbCr .Truncation-case3 (f).Truncation-case2 (e).case1



شکل شماره ۸۷. خروجی مازول تبدیل فضای رنگی RGB به YCbCr با خروجی memoization معمولی



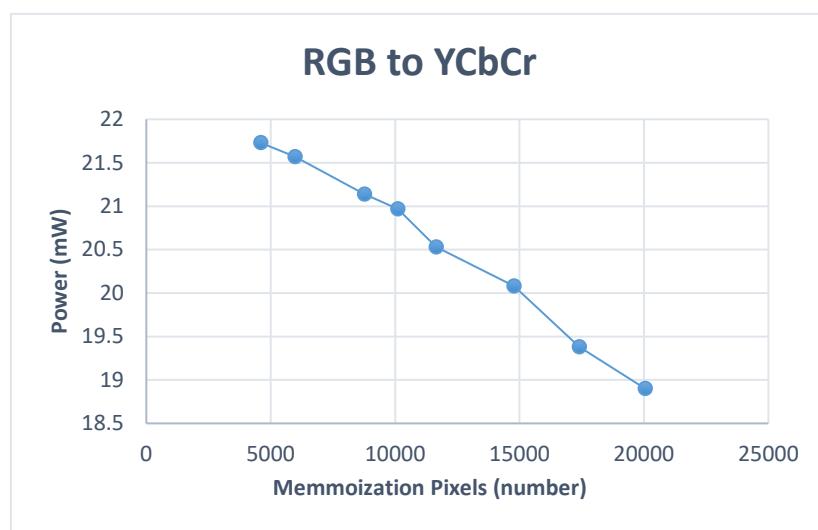
شکل شماره ۸۸. خروجی مازول تبدیل فضای رنگی RGB به YCbCr با خروجی memoization مشکی



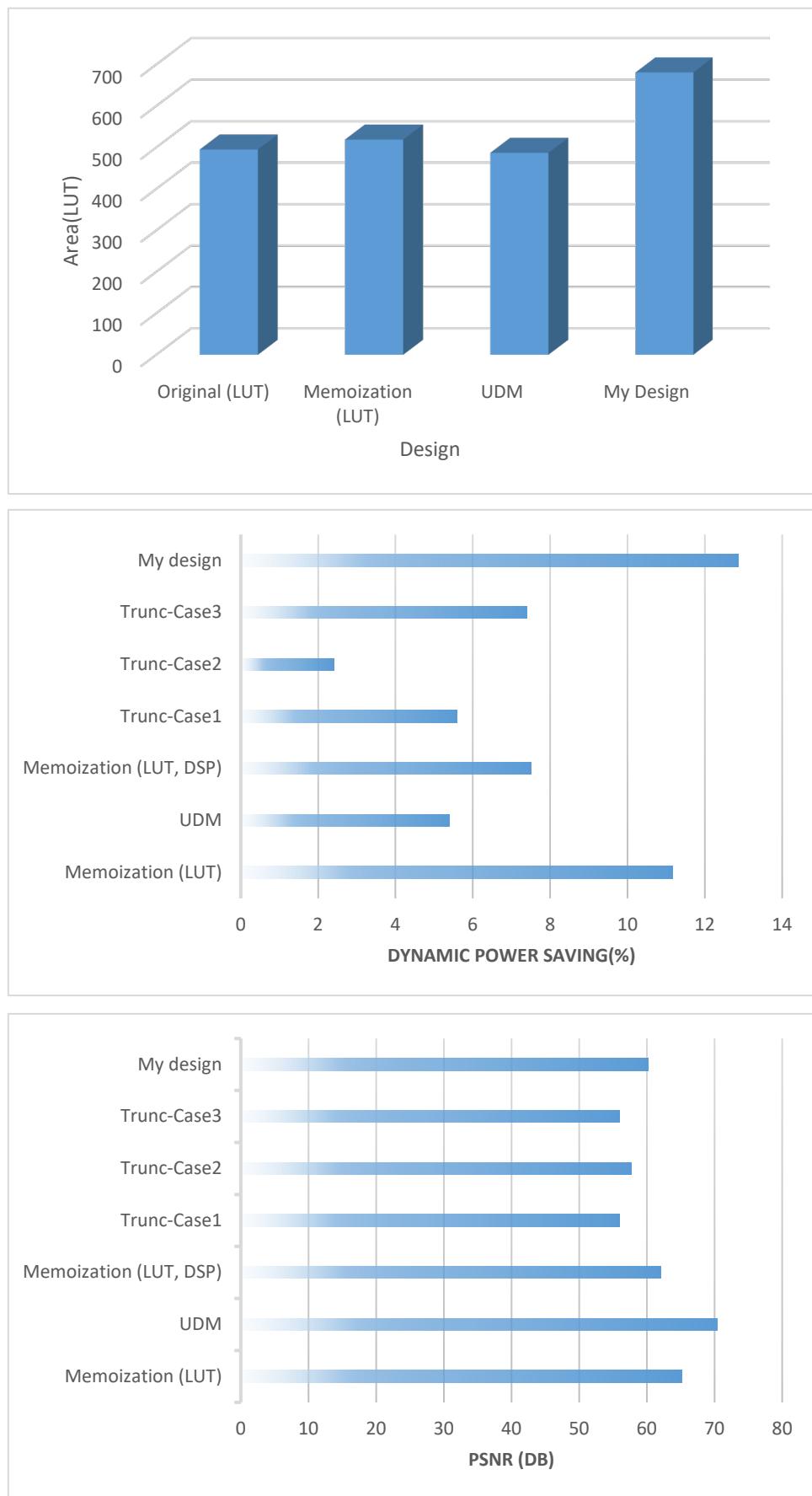
شکل شماره ۸۹. تصویر ورودی در فضای RGB

نمودار توان بر حسب مقدار پیکسل‌های تقریبی

همانطور که اشاره کردیم، میزان توان مصرفی به تعداد پیکسل‌هایی که مقدار آن‌ها از واحد memoization بدست آمده، وابسته است و هر چه تعداد این پیکسل‌ها بیشتر باشد، میزان توان مصرفی کاهش خواهد یافت. تعداد پیکسل‌هایی که مقدار آن‌ها بدون محاسبه و از واحد memoization مستقیم بدست می‌آید، به مقدار آستانه و مقدار پیکسل پایه بستگی دارد. ما در نمودار زیر، میزان توان مصرفی بر حسب تعداد پیکسل‌های تقریبی را نشان داده‌ایم. همانطور که شکل زیر نشان می‌دهد و ما نیز انتظار داشتیم، توان با افزایش تعداد پیکسل‌های تقریبی و کاهش محاسبات، کم می‌شود. لازم به ذکر است که خود مقاله به این موضوع نپرداخته بود و ما برای اثبات این موضوع، تصمیم به رسم این نمودار گرفتیم.



مقایسه‌ی طراحی‌های مختلف



بخش هشتم : نتیجه‌گیری و جمع‌بندی

هدف ما از نگارش این گزارش و انجام این پژوهه، ارائه راهکاری برای کاهش توان مصرفی در کاربردهایی مثل تشخیص لبه و تغییر فضای رنگی بود. مقاله‌ای که ما انتخاب کردیم، با استناد به مقالات [۱]، [۲] و [۳]، ادعا کرده بود استفاده از روش محاسبات تقریبی و حافظه‌سازی، موجب کاهش توان شده و نسبت به روش‌های مرسوم مثل UDM و Data path truncation موثرتر عمل خواهد کرد. از این رو مقاله با پیاده‌سازی دو نمونه کاربرد تشخیص لبه و تغییر فضای رنگی و مقایسه و بررسی توان مصرفی روش‌های مختلف، ایده خود را اثبات کرد. ما هم همانند مقاله این دو کاربرد را با FPGA پیاده‌سازی کرده و به همان نتایج رسیدیم، اگرچه مقادیری که برای توان و سربار سخت‌افزار بدست آوردیم، به دلیل فرضیات مختلف که در بخش‌های قبلی کاملاً به آن‌ها اشاره کردیم، با مقادیر بدست آمده در این مقاله یکسان نبوده و تفاوت داشت.

مسئله‌ی اصلی و بزرگ‌ترین ایرادی که به این مقاله می‌توان گرفت، فرض شباهت بالای تصاویر ورودی است، این مقاله فرض کرده تصاویر ورودی تا ۹۰ درصد یکسان هستند و اکثر تصاویر مشابه بوده، مثلاً تصاویر از یک مکان ثابت در ساعات مختلف روز هستند، یا در کاربرد تبدیل فضای رنگی، معمولاً یک رنگ (مثلاً رنگ سفید در بدنه‌ی خودروها) دائمًاً تکرار می‌شود. با این فرض‌ها استفاده از این دو تکنیک کاملاً به صرفه است، زیرا همانطور که در بخش قبل مطرح کردیم، روش‌های محاسبات تقریبی و حافظه‌سازی، هزینه‌ی سخت‌افزاری بالایی دارند و استفاده از آن‌ها زمانی قابل توجیه است که ورودی‌ها مشابه هم باشند تا نیازی به محاسبه Data path و UDM نباشد. اما در صورتی که ورودی‌ها شباهت زیادی نداشته باشند، روش‌هایی مثل truncation بهتر عمل کرده و استفاده از Memoization دیگر صرفه ندارد.

پیشنهاد می‌شود برای بدست آوردن میزان بهبودی، یک ابزار مثل MEET برای FPGA‌ها توسعه داده شود تا تعداد سوئیچینگ را برای تمام سیگنال‌ها حساب کند و طرح‌های مختلف نیز براساس تعداد سوئیچینگ با هم مقایسه شوند. همچنین پیشنهاد می‌شود ایده‌ی مطرح شده در این مقاله را با firmware مختلف، به خصوص در میکروکنترلرهای CORTEX-M پیاده‌سازی شود و کارایی این روش‌ها در آنجا نیز

مورد بررسی قرار گیرد. همچنین می‌توان با تکنیک‌های یادگیری ماشین در کاربردهای پیچیده‌تر، محتویات حافظه memoization را انتخاب کرد تا میزان بهبودی این روش‌ها افزایش یابد. یکی از کاربردهایی که این روش‌ها می‌توانند در آن خوب عمل کنند، خودروهای خودران و دوربین‌های نظارتی می‌باشد.

مراجع

- [1] P. Kulkarni, P. Gupta, and M. Ercegovac, “Trading accuracy for power with an underdesigned multiplier architecture,” in Proc. 24th Int. Conf.VLSI Design, Jan. 2011, pp. 346–351.
- [2] R. Hegde and N. R. Shanbhag, “Energy-efficient signal processing via algorithmic noise-tolerance,” in Proc. Int. Symp. Low Power Electron .Design, 1999, pp. 30–35.
- [3] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, “Low-power digital signal processing using approximate adders,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 32, no. 1 ,pp. 124–137, Jan. 2013.
- [4] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, “ABACUS: A technique for automated behavioral synthesis of approximate computing circuits,” in Proc. DATE, Dresden, Germany, 2014, Art. ID 361.
- [5] F. Khalvati and M. D. Aagaard, “Window memoization: An efficient hardware architecture for high-performance image processing”, J. Real-Time Image Process., vol. 5, no. 3, pp. 195 - 212, Sep. 2010.
- [6] Static Power Consumption in FPGA. [Online]. Available: <http://forums.xilinx.com/t5/Design-Tools-Others/How-to-calculate-estimate-staticpower-on-FPGAs-just-for/td-p/226289>, accessed Dec. 22, 2015.
- [7] Yadav, S., Raj, R. “Power efficient network selector placement in control plane of multiple networks-on-chip”, J Supercomput 78, 6664–6695, 2022
- [8] M. Wilson. Dataset. [Online]. Available: http://www.vision.caltech.edu/Image_Datasets/MtWilson/index.html, accessed Dec. 24, 2015.
- [9] Car License Plate Dataset. [Online]. Available: <http://www.medialab.ntua.gr/research/LPRdatabase.html>, accessed Dec. 24, 2015.
- [10] Xilinx Power Tools Tutorial, UG 733 v14.3, Xilinx Inc., San Jose, CA, USA, Oct. 2012.