筑波大学技術報告
36 : 18 - 26 , 2015

ソフトウェアとハードウェア性能比較実験装置の開発

小野　雅晃
筑波大学システム情報工学等技術室
〒305-8573 茨城県つくば市天王台1-1-1

概要

近年、高性能なプロセッサに FPGA(Field Programmable Gate Array)が付属している Xilinx 社の Zynq や Altera 社の SoC FPGA などの SoC(System On a Chip)が作られるようになってきた。これらの SoC では、ARM プロセッサと FPGA が内部の AXI バスで密に結合され、プロセッサで行われる処理の一部を比較的簡単にハードウェア(FPGA)にオフロードすることができる。

本研究では、この特徴を利用して、Xilinx 社の Zynq を使用した学生実験用のソフトウェアとハードウェア性能比較実験装置（以下、実験装置とする）を開発した。

キーワード： Zynq、FPGA

1．はじめに

本研究では、ソフトウェアとハードウェアの協調設計に関する学生の理解を深めるための実験装置を製作した。実験装置は、Xilinx 社の Zynq チップを搭載した ZYBO ボード 2 台と、FPGA のツールを搭載する超小型のパーソナルコンピュータを 1 つの箱に入れてコンパクトな形状とした。

Zynq チップは、Xilinx 社の ARM プロセッサと FPGA が 1 つの SoC として 1 チップに集積されている。この特徴から種々の機能を ARM プロセッサで動作するソフトウェアとして実装することも、FPGA で動作するハードウェアとして実装することも容易に行うことができる。

ZYBO ボードには、CMOS カメラを搭載して画像処理のための画像を取得できるようにした。その 2 台の ZYBO ボードの間を高速な HDMI ポートで接続した。そのため、実験装置 2 台で並列処理を行うことができる。なお、超小型のパーソナルコンピュータを全体制御用として設置し、実験の制御を行わせる。

実験装置で、画像のフィルタ処理を 1.～5.の条件で実行した時の実行時間を測定し、比較検討を行うことができた。

1. 最適化オプション無し、または複数のレベルの最適化オプションを付加した状態でコンパイルした時に、複数の異なる実装のラプラシアンフィルタ処理ソフトウェアを ARM プロセッサのみで実行した場合
2. 最適化オプション無し、または複数のレベルの最適化オプションを付加した状態で、ARM プロセッサ＋ SIMD(Single Instruction Multiple Data)エンジン NEON を使用するようにコンパイルした複数の実装のラプラシアンフィルタ処理ソフトウェアを実行した場合
3. OpenMP を使用して、1.と 2.を実行した場合
4. ソースコードを Vivado HLS でハードウェア(FPGA)にオフロードした場合（AXI バスのマスタアクセスを使用した）
5. HDL(Hardware Description Language)でハードウェア(FPGA)に画像フィルタを実装した場合

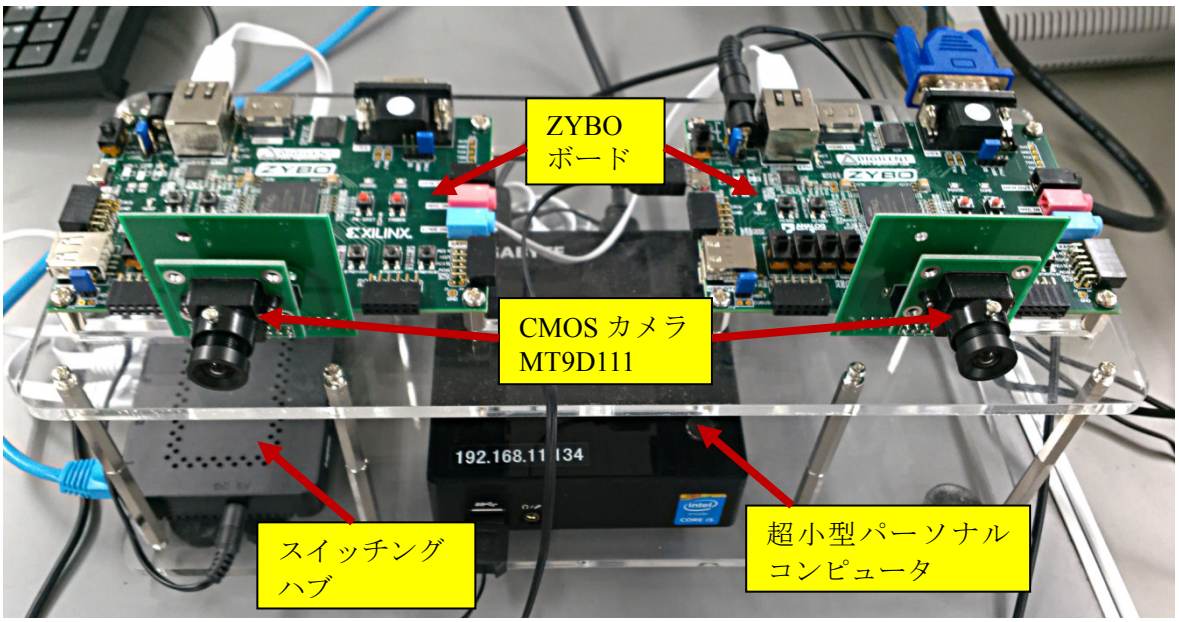

図 1.　ソフトウェアとハードウェアの性能比較実験装置

２．実験装置

実験装置を図 1 に示す。実験装置は CMOS カメラの MT9D111 を Pmod コネクタに搭載した ZYBO ボードが 2 台、超小型パーソナルコンピュータを 1 台、スイッチングハブを 1 台搭載している。2 台の ZYBO ボードのカメラを使用して、ステレオカメラの機能を持つように考慮されている。実験装置のベースプレートは 5mm のアクリル板で、レーザー加工機を使用して加工を行った。レーザー加工機を使用しているので、自由な加工が可能である。その利点を活かして、ZYBO ボードの取り付け穴は長円形に造作したので、ZYBO ボードをスライドさせて CMOS カメラのステレオカメラの間隔を変更することができる。MT9D111 の最大解像度は 1600 ピクセル× 1200 ラインで、その場合のフレームレートは 15fps(frame per second)である。MT9D111 は、現状では 800 ピクセル× 600 ラインの 15fps で動作している。MT9D111 はインターフェース基板に搭載されて、ZYBO の Pmod コネクタの内の 2 個に取り付けられている。その様子を図 2 に示す。

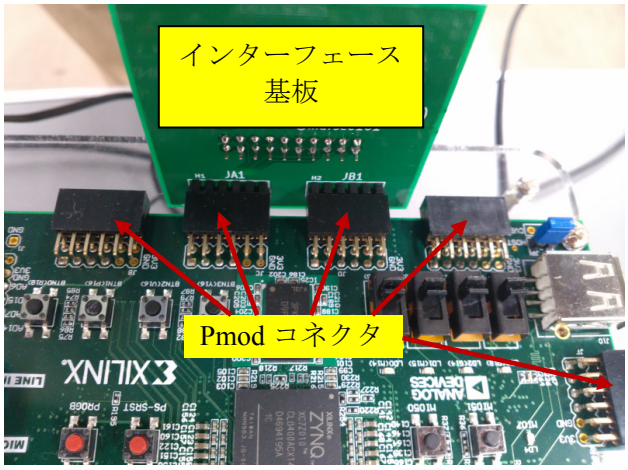

図 2. ZYBO ボードの Pmod コネクタとインターフェース基板を介した CMOS カメラの取り付け

超小型パーソナルコンピュータは、intel Core i5 プロセッサが搭載され、8GB のメモリ、256GB の SSD (Solid State Drive)が搭載されている。超小型パーソナルコンピュータには、Ubuntu14.04 がインストールされている。その上に HDL を記述することで FPGA 部分のハードウェアを生成する Xilinx 社の Vivado ツールや C、C++、SystemC 言語から HDL を生成することができる Vivado HLS、Vivado が生成したハードウェアに応じたソフトウェアを作製することができる SDK(Software Development Kit)などをインストールしてある。これらによって、USB ケーブルで接続された ZYBO ボードの FPGA のコンフィギュレーションを実行することや、ARM プロセッサのソフトウェアのデバックを行うことができる。

更に、実験装置内にスイッチングハブを装備して、ZYBO 2 台の LAN と超小型パーソナルコンピュータの LAN を外部ネットワークに接続した。

３．ラプラシアンフィルタ処理

実験装置の ZYBO ボードで実行する画像処理として、ラプラシアンフィルタ処理を選択した。ラプラシアンフィルタ処理は、画像のエッジを検出する画像フィルタの 1 種で、画像のピクセルの縦横のピクセルを 3x3、5x5 のブロック単位で演算を行い、真ん中の画像の値とする。3x3、5x5 のラプラシアンフィルタの画像に掛ける係数を図 3 に示す。

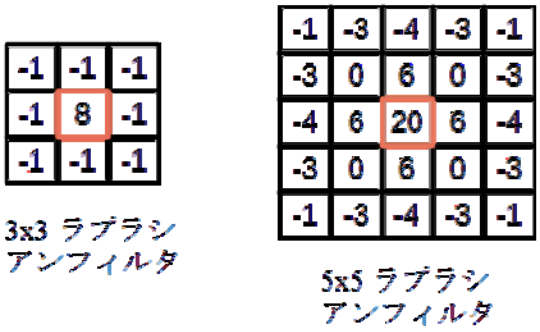

図 3. 各ラプラシアンフィルタの係数

図 3 の 3x3、5x5 のラプラシアンフィルタの係数は画像の輝度の値に掛ける係数を示し、中央の赤い四角はラプラシアンフィルタ処理後に演算結果が入るピクセルを示している。例えば、3x3 ラプラシアンフィルタで演算を行うすべてのピクセル値が 10 だったとする。その場合は式 1 に示すようにラプラシアンフィルタ処理後の値は 0 となる。

$$10 \times(-1)+10 \times(-1)+10 \times(-1) +10 \times(-1)+10 \times 8+10 \times(-1) +10 \times(-1)+10 \times(-1)+10 \times(-1)=0 \cdots (式\ 1)$$

今回は 3x3 のラプラシアンフィルタを採用した。画面全体に対するラプラシアンフィルタ処理を説明する。最初に左上の画面のピクセルを 3x3 の係数で 1 回演算し、中央のピクセルの値とする。1 ピクセル右にずらしてからもう一度 3x3 の係数で演算すると最初のピクセルの左隣のピクセルの値が計算できる。それを左端まで演算を行う。次は、左上から 1 ライン下にずらし、3x3 の係数で演算を行う。後は順次演算を行い、右下端まで処理を行うと、1 画面分のラプラシアンフィルタ処理が終了する。その様子を図 4 に示す。

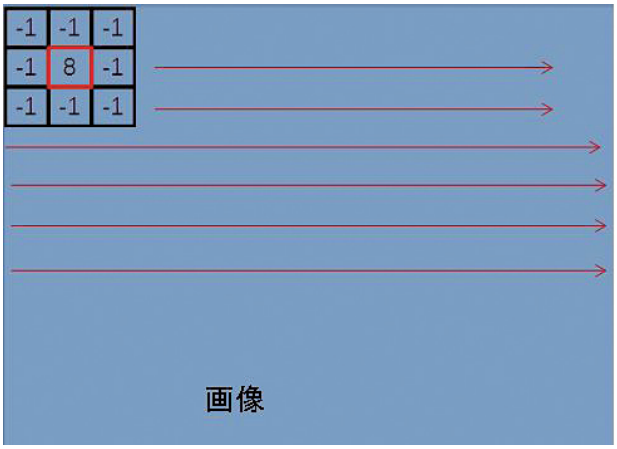

図 4. 1 画面分のラプラシアンフィルタ処理

図 5.　ラプラシアンフィルタ処理前の画像

図 6.　ラプラシアンフィルタ処理後の画像

ラプラシアンフィルタ処理は、Zynq チップの ARM プロセッサのソフトウェアで処理することもできるが、Zynq チップの FPGA 部分を使用したハードウェアとして実行することもできる。ラプラシアンフィルタ処理前の画像を図 5 に示し、ラプラシアンフィルタ処理後の画像を図 6 に示す [1]。

4．ZYNQ チップの構造

Zynq チップ [2] は ARM プロセッサと FPGA 部分が 1 チップに統合された IC(Integrated Circuit)である。ARM プロセッサが搭載されている領域は PS(Processing System)と呼ばれていて、2 個の ARM プロセッサと IO 周辺デバイスが搭載されている。PS に搭載されているデバイスを示す。

- 2 個の ARM プロセッサ（最大 1GHz 動作）
- 2 個の NEON SIMD(Single Instruction Multiple Data)エンジン（ARM プロセッサに付属）
- 2 個の 32KB Instruction L1 Cache、32KB Data L1 Cache（ARM プロセッサに付属）
- 512KB L2 Cache
- DDR(Double Data Rate)2, DDR3, DDR3L, LPDDR(Low Power Double Data Rate)2 コントローラ
- 2 個の USB(Universal Serial Bus)
- 2 個の Gigabit Ethernet
- 2 個の SD(Secure Digital), SDIO(Secure Digital Input Output)
- GPIO(General Purpose Input Output)
- 2 個の CAN(Controller Area Network)
- 2 個の I2C(Inter-Integrated Circuit)
- 2 個の SPI（Serial Peripheral Interface）

上記に示す様に、多くのデバイスが PS に搭載されて、ARM プロセッサから使用することができる。Zynq チップのブロック図を図 7 に示す。

PS と PL(Programmable Logic, FPGA 搭載部)は AXI バスで接続されている。PS と PL 間の接続ポートを示す。

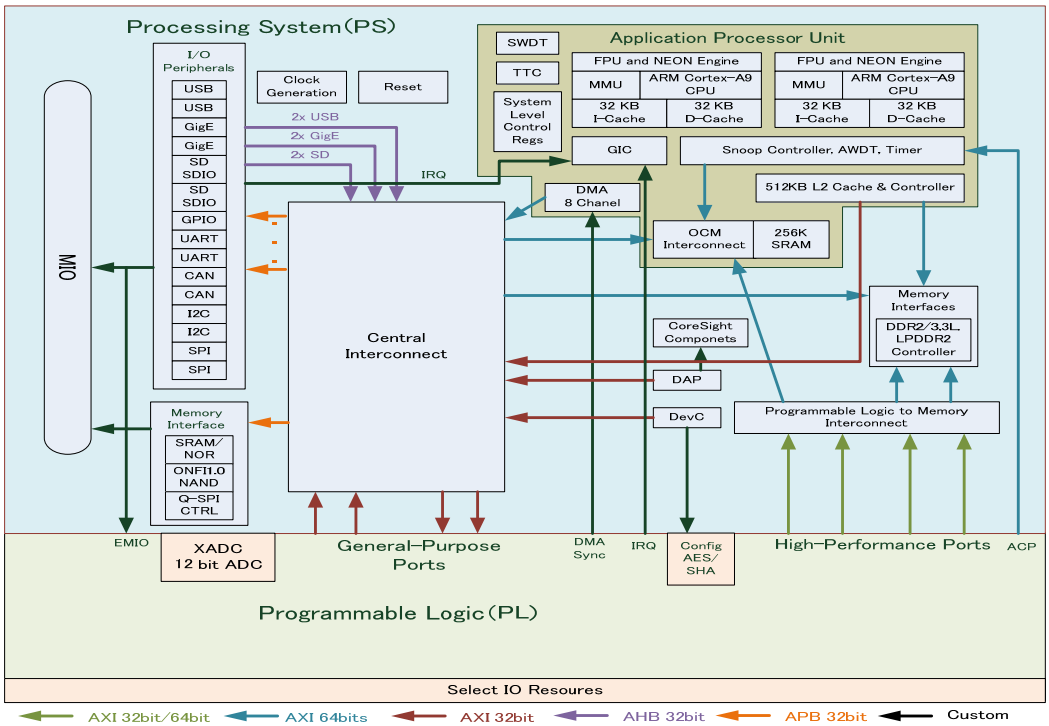

図 7.　Zynq チップのブロック図

- 4 ポートの AXI_HP ポート（図 7 では High-Performance Ports）
- 各 2 ポートの AXI_GP マスタ&スレーブ・ポート（図 7 では General-Purpose Ports）
- 1 ポートの AXI_ACP ポート（図 7 では ACP）

AXI_HP ポートは高速のスレーブ・ポートで、動作周波数が 200MHz、64 ビット幅のデータバスの場合の性能は最大 1.6GB/s となる。AXI_GP ポートは PL に作った IP のレジスタの設定などの性能が比較的要求されない用途に使用する事ができる。AXI_ACP ポートは ARM プロセッサのキャッシュを対象に Read/Write することができるので、ARM プロセッサと PL 間のデータの Read/Write 時のレイテンシが短くなる。

５．ラプラシアンフィルタの実装方法

CMOS カメラで撮影された画像（撮影画像）は、Zynq チップの PL 部分に実装されたカメラ・コントローラの DMA(Direct Memory Access)で、DDR3 SDRAM の所定の領域に転送される。

ビットマップ・ディスプレイ・コントローラはカメラ画像領域の画像データか、またはラプラシアンフィルタ処理領域の画像データを読み出して、ディスプレイに表示する。

ラプラシアンフィルタ処理を行う際には、ソフトウェアまたはハードウェアが、カメラ画像領域から画像データを読み出して、ラプラシアンフィルタ処理を行い、結果をラプラシアンフィルタ処理領域に書き込む。ソフトウェアがラプラシアンフィルタ処理を行う際には、ARM プロセッサがラプラシアンフィルタ処理を行い、ハードウェアが行う場合は PL 部に実装されたラプラシアンフィルタ IP がラプラシアンフィルタ処理を行う。ラプラシアンフィルタ IP は、DDR3 SDRAM のカメラ画像領域から画像データを AXI4 Master Read を使用して DMA アクセスにより、読み出してラプラシアンフィルタ処理を行う。そのラプラシアンフィルタ処理結果は AXI4 Master Write を使用して DMA アクセスにより、DDR3 SDRAM のラプラシアンフィルタ処理領域に書き込まれる。その様子を図 8 に示す。

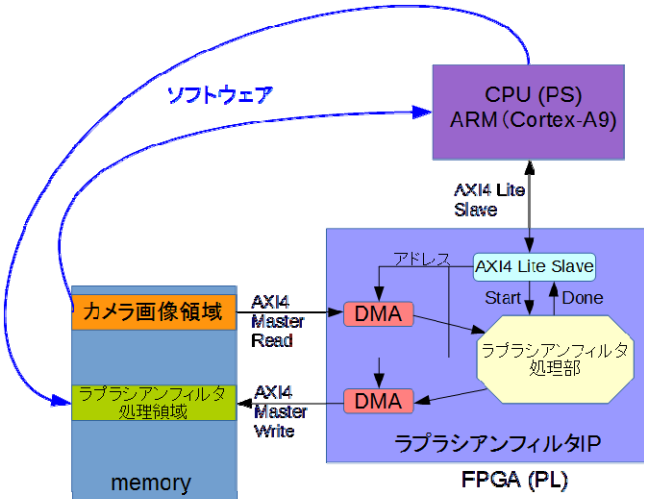

図 8. ソフトウェア、ハードウェアによるラプラシアンフィルタ処理の実装方法

ハードウェアの性能比較対象としては、C や C++、SystemC 言語から HDL(Hardware Description Language)に変換する高位合成ツールの Vivado HLS を用いた場合と HDL を用いて直接ラプラシアンフィルタ処理を書いた場合についての性能比較を行った。Vivado HLS を使用した場合には、C や C++言語によるソースコードの書き方やディレクトリと呼ばれる指示子によって性能が変化するので、4 種類の異なる実装による性能比較を行った。

ソフトウェアの性能比較対象としては、Vivado HLS で使用したソースコードとほぼ同一の 4 種類のソフトウェア（laplacian_filter1~laplacian_filter4[3]）を用意してその性能を比較した。なお、Vivado HLS ではディレクティブを付加して性能を向上させているが、ソフトウェアには無関係であるためディレクティブは削除した。また、ARM プロセッサだけで無く、図 7 に示した SIMD(Single Instruction Multiple Data)の NEON エンジンを使用する条件でも 4 種類のソフトウェアについて実験を行った。通常の ARM プロセッサは 1 個のみを使用してラプラシアンフィルタ処理を実行するが、搭載されている 2 個の ARM プロセッサを同時に使用する OpenMP を使った時の性能も確認した。OpenMP を使用した場合には、4 種類のソフトウェアでは処理画像が不完全だったので、今まで使用していた 4 種類の実装を小変更した 4 種類(laplacian_filter5 ~ laplacian_filter8[3])も使用した。その変更内容は、従来のラプラシアンフィルタの実装では画像の周辺 1 ピクセルの値を 0 にしていたが、この実装のソースコードでは条件分岐が増えてしまう。そこで画像の左上と左端の 2 ピクセルを 0 にすることにした。これにより、ソースコードの条件分岐が減って OpenMP が 3 種類の実装では、正しく動作するようになった。しかし、最後の実装(laplacian_filter8)では処理画像が不完全だった。ソースコードの実装に違いによるラプラシアンフィルタ出力の違いを図 9 に示す。

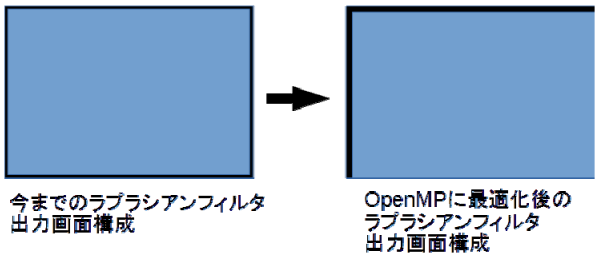

図 9. OpenMP への最適化によるラプラシアンフィルタの実装の違い

図 9 で、黒く示した部分が 0 に固定した部分で、青く示した部分がラプラシアンフィルタ処理出力の部分である。

ラプラシアンフィルタ処理のハードウェア実装とソフトウェア実装では、ほぼ同一のソースコードを使用しているため、実行時間の差を容易に比較することができる。

表 1. gcc 4.6 でコンパイルしたラプラシアンフィルタの各実装の最適化オプションの違いによる処理時間

	−O無し (ms)	-O1 (ms)	-O2 (ms)	-O3 (ms)	-Os (ms)	-Ofast (ms)
laplacian_filter1	448.0	130.0	130.0	127.0	136.0	127.0
laplacian_filter2	328.0	99.5	86.4	85.1	96.3	82.1
laplacian_filter3	279.0	94.2	83.6	82.7	91.6	82.8
laplacian_filter4	228.0	123.0	92.8	93.3	114.0	92.7

6. 実験項目

「5. ラプラシアンフィルタの実装方法」に記した方法でソフトウェア、ハードウェアの性能比較を行った。4 種類のソースコードはそれぞれ laplacian_filter1、laplacian_filter2、laplacian_filter3、laplacian_filter4 とする。それぞれのソースコードについての説明を次に述べる。

- laplacian_filter1
  - ソフトウェアとして作製した C ソースコードを使用した
  - 画像を 1 ピクセル読んで、ラプラシアンフィルタ処理し、1 ピクセルごとに書く
- laplacian_filter2
  - memcpy()を使用して画像データをまとめて読み書きした（memcpy()の使用により Vivado HLS ではバースト転送が行われる）
- laplacian_filter3
  - laplacian_filter2 の割り算をステートマシンに変更した（Vivado HLS では PIPELINE ディレクティブを挿入した）
- laplacian_filter4
  - memcpy()の使用やソースコードを Vivado HLS に最適化した（Vivado HLS では最適なディレクティブを追加して高速化を図った）

laplacian_filter1 ~ laplacian_filter4 を図 9 に示す通り、画像の左上と左端の 2 ピクセルを 0 にする様に改造したソフトウェアがそれぞれ laplacian_fiter5 ~ laplacian_filter8 に対応する。

なお、ZYBO 基板の ARM プロセッサ Cortex-A9、動作周波数 650MHz で動作する Ubuntu 14.04 LTS 上でソフトウェア及びハードウェアの実行を行った。時間計測は Linux の gettimeofday システムコールで行った。

6.1 コンパイラに gcc 4.6 を使用した場合

最初に gcc 4.6 でコンパイルした場合の最適化オプションによるラプラシアンフィルタ処理時間を表 1 に示す [4]。

表 1 を最適化オプションごとに棒グラフにした図を図 10 に示す。図 10 の縦軸はラプラシアンフィルタの処理時間 ms 単位、横軸は最適化オプションごとのラプラシアンフィルタの実装の違いによる性能差を表す。図 10 を見ると gcc の最適化オプション無しでは、laplacian_filter1 ~ laplacian_filter4 までで直線的に処理時間が減っているのがわかる。また表 1 からも最適化オプション無しと最適化オプション有りとでは、処理時間に差があるのがわかる。

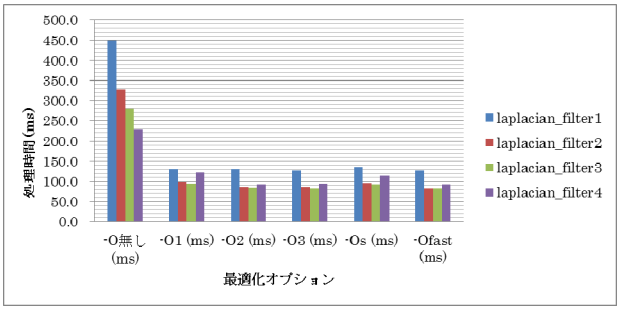

図 10. gcc 4.6 でコンパイルしたラプラシアンフィルタの各実装の最適化オプションの違いによる処理時間のグラフ

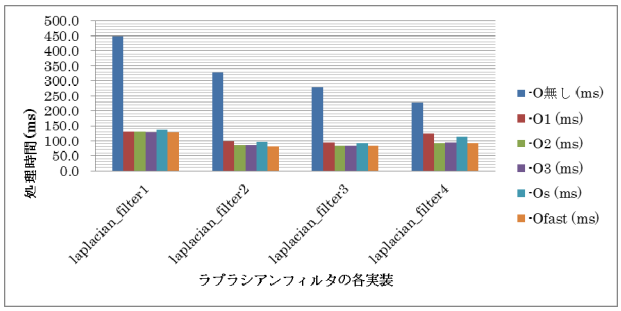

図 11. gcc 4.6 でコンパイルしたラプラシアンフィルタの各実装の違いによる処理時間のグラフ

図 11 はラプラシアンフィルタの実装ごとの最適化オプションによる性能の差をラプラシアンフィルタの実装ごとに並べたグラフである。各実装共に、最適化オプション無しの処理時間が長いことがわかった。最速の処理時間としては、laplacian_filter2 の最適化オプション-O2 の場合の 82.1 ms が最速だった。

次に、同じ gcc 4.6 を使用して、ARM プロセッサの SIMD エンジンの NEON を使用してラプラシアンフィルタ処理を演算した結果を表 2 に示す [4]。

gcc のオートベクタライズ機能を使用してコンパイルし NEON 用のコードを出力した。gcc のオートベクタライズのオプションは-mcpu=cortex-a9

表 2. gcc 4.6 のオートベクタライズ機能によって NEON エンジンを使用した場合のラプラシアンフィルタの処理時間

	NEON −O無し (ms)	NEON -O1 (ms)	NEON -O2 (ms)	NEON -O3 (ms)	NEON -Os (ms)	NEON -Ofast (ms)
laplacian_filter1	447.0	131.0	132.0	137.0	138.0	137.0
laplacian_filter2	313.0	95.0	78.3	83.4	92.2	83.0
laplacian_filter3	278.0	94.4	77.7	83.4	92.0	84.2
laplacian_filter4	229.0	123.0	96.4	96.3	118.0	96.2

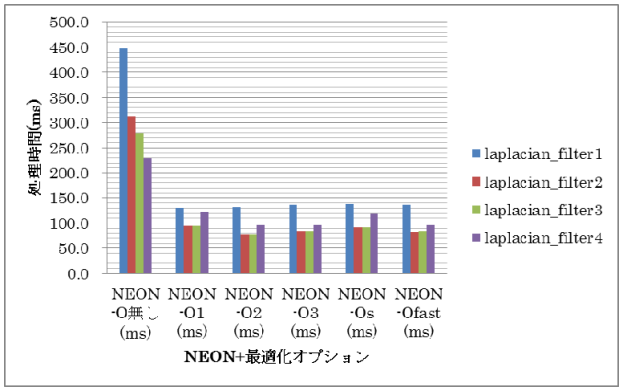

図 12. gcc 4.6 オートベクタライズ機能によって NEON エンジンを使用した場合の各実装の最適化オプションの違いによる処理時間のグラフ

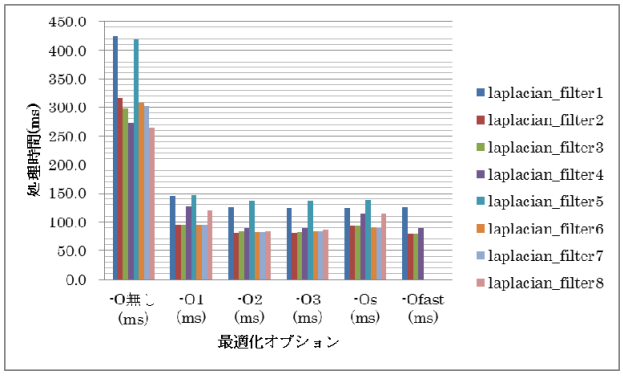

図 13. gcc 4.8 でコンパイルしたラプラシアンフィルタの各実装の最適化オプションの違いによる処理時間のグラフ

-mfpu=neon -ftree-vectorize -ffast-math -mvectorize-with-neon-quad を使用した。但し、コンパイル時の最適化オプションやラプラシアンフィルタの実装によっては、NEON 用のコードが出力されない時がある。表 2 で赤い字によって示された処理時間が、NEON のコードが出力された条件の場合を示す。NEON を使用した場合も gcc の最適化オプションを入れたほうが速いことがわかる。最適化オプションによる処理時間の違いをラプラシアンフィルタの実装ごとにグラフにした図を図 12 に示す。図 10 と図 12 は同じ傾向を示した。gcc 4.6 のオートベクタライズ機能を使用してラプラシアンフィルタ処理をコンパイルした時の最速の値は、laplacian_fiter3 の最適化オプション-O3 の時の 77.7 ms だった。NEON を使用しない場合に比べて NEON を使用した場合のほうが最速の処理時間がより短くなった。

6.2 コンパイラに gcc 4.8 を使用した場合

gcc 4.8 の場合のラプラシアンフィルタの gcc 最適化オプションと NEON 使用の場合のラプラシアンフィルタ処理時間を表 3、表 4 に示す [5]。gcc 4.8 を使用した場合は laplacian_filter5 ~ laplacian_filter8 も処理時間を測定して表に追加した。ラプラシアンフィルタの各実装における最適化オプションの違いによる処理時間のグラフを図 13 に示す。

表 3. gcc 4.8 でコンパイルしたラプラシアンフィルタの各実装の最適化オプションの違いによる処理時間

	-O無し (ms)	-O1 (ms)	-O2 (ms)	-O3 (ms)	-Os (ms)	-Ofast (ms)
laplacian_filter1	425.0	145.0	126.0	125.0	125.0	126.0
laplacian_filter2	316.0	95.0	80.9	80.8	94.0	80.2
laplacian_filter3	299.0	95.3	84.4	82.2	94.4	80.2
laplacian_filter4	273.0	127.0	88.9	88.6	114.0	88.9
laplacian_filter5	420.0	148.0	137.0	137.0	139.0	
laplacian_filter6	308.0	95.2	82.7	83.5	89.9	
laplacian_filter7	303.0	95.7	83.2	83.5	89.3	
laplacian_filter8	265.0	119.0	84.2	87.1	114.0	

表 4. gcc 4.8 のオートベクタライズ機能によって NEON エンジンを使用した場合のラプラシアンフィルタの処理時間

	NEON -O無し (ms)	NEON -O1 (ms)	NEON -O2 (ms)	NEON -O3 (ms)	NEON -Os (ms)	NEON -Ofast (ms)
laplacian_filter1	425.0	145.0	133.0	143.0	128.0	141.0
laplacian_filter2	315.0	94.2	75.1	75.3	94.4	76.3
laplacian_filter3	299.0	95.2	74.9	74.4	94.1	74.6
laplacian_filter4	273.0	127.0	88.9	88.9	122.0	89.4
laplacian_filter5	420.0	147.0	131.0	132.0	143.0	
laplacian_filter6	309.0	95.3	79.8	81.1	88.9	
laplacian_filter7	303.0	94.8	80.5	81.9	90.7	
laplacian_filter8	265.0	119.0	87.7	86.9	116.0	

表 5. llvm 3.4、clang 3.4 でコンパイルしたラプラシアンフィルタの各実装の最適化オプションの違いによる処理時間

	-O無し (ms)	-O1 (ms)	-O2 (ms)	-O3 (ms)	-Os (ms)	-Ofast (ms)
laplacian_filter1	304.0	148.0	137.0	133.0	134.0	133.0
laplacian_filter2	238.0	109.0	99.0	101.0	103.0	101.0
laplacian_filter3	221.0	106.0	95.5	97.7	105.0	97.6
laplacian_filter4	241.0	121.0	94.7	95.0	94.9	94.9

表 6. llvm 3.4、clang 3.4 のオートベクタライズ機能によって NEON エンジンを使用した場合のラプラシアンフィルタの処理時間

	NEON −O無し (ms)	NEON -O1 (ms)	NEON -O2 (ms)	NEON -O3 (ms)	NEON -Os (ms)	NEON -Ofast (ms)
laplacian_filter1	300.0	143.0	134.0	134.0	133.0	133.0
laplacian_filter2	232.0	102.0	86.8	88.3	87.9	88.6
laplacian_filter3	219.0	102.0	89.7	90.8	88.4	91.0
laplacian_filter4	230.0	115.0	92.4	92.4	93.4	93.2

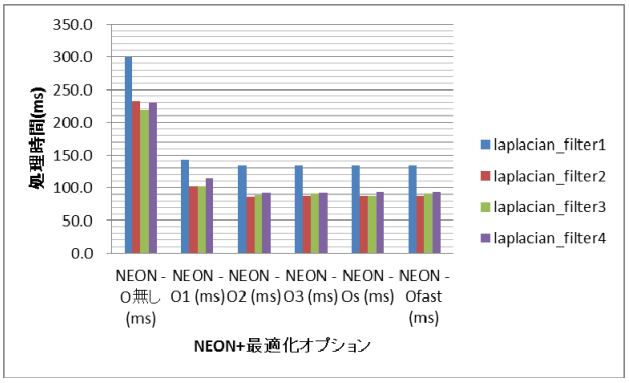

図 14. llvm 3.4、clang 3.4 オートベクタライズ機能によって NEON エンジンを使用した場合の最適化オプションによる処理時間のグラフ

laplacian_filter1 と laplacian_filter5 は多少実装が異なるだけなので、ほとんど同じ傾向を示す。laplacian_filter2 と laplacian_filter6 もその他も同様である。gcc 4.6 と gcc 4.8 のラプラシアンフィルタの処理時間を比較すると、一長一短であることがわかった。最短の処理時間は、gcc 4.8 の場合の laplacian_filter3 の NEON を使用した-O3 の場合で、74.4 ms だった。

6.3 コンパイラに llvm 3.4, clang 3.4 を使用した場合

llvm 3.4, clan 3.4 を使用してコンパイルした場合の最適化オプションと NEON 使用の場合のラプラシアンフィルタ処理時間を表 5、表 6 に示す [4]。NEON を使用する場合のオートベクタライズのオプションは、-mcpu=cortex-a9 -mfpu=neon -mfloat-abi=hard を使用し、そこに最適化オプションを追加している。また、gcc でコンパイルした場合と同様に、オートベクタライズのオプションを使用してコンパイルしても、NEON を使用するコードを出力するとは限らない。表 6 の NEON を使用するコードが含まれている実行コードの処理時間を赤い字で表示した。llvm 3.4、clang 3.4 オートベクタライズ機能によって NEON エンジンを使用した場合の最適化オプションによる処理時間のグラフを図 14 に示す。図 14 を見ると、最適化オプション無しの場合は、gcc でコンパイルしたよりも laplacian_filter2 ~ laplacian_fiter4 の処理時間の差が少ないことがわかった。gcc 4.6、gcc 4.8 に比べて llvm 3.4、clang 3.4 ではラプラシアンフィルタの処理時間が長くかかる傾向があることがわかった。

6.4 OpenMP を用いて 2 つの CPU コアを使用した場合

gcc 4.8 において OpenMP を使用するためのオプション- fopenmp をコンパイル・オプションとして追加した場合の最適化オプションによるラプラシアンフィルタ処理時間を表 7 に示す [5]。gcc 4.8 において OpenMP を使いながら、更に、NEON エンジンを使用した時の最適化オプションによるラプラシアンフィルタ処理時間を表 8 に示す [5]。なお、処理時間の赤字は NEON エンジンを使用するコードが出力されていたことを示す。ラプラシアンフィルタ処理結果は、laplacian_fiter1 ~ laplacian_filter4、laplacian_filter8 で表示結果にバグがあった。最短の処理時間は laplacian_filter8 の最適化オプション -O3 の場合の 72.5 ms だった。これは、ソフトウェア実装最速のラプラシアンフィルタ処理時間である。

表 7. gcc 4.8 において OpenMP を使用した場合の最適化オプションによる処理時間

	−O無し (ms)	-O1 (ms)	-O2 (ms)	-O3 (ms)	-Os (ms)
laplacian_filter1	209.0	117.0	105.0	101.0	211.0
laplacian_filter2	236.0	107.0	94.4	93.9	144.0
laplacian_filter3	224.0	100.0	87.0	87.0	94.1
laplacian_filter4	167.0	90.0	74.4	76.8	90.0
laplacian_filter5	231.0	111.0	95.2	95.3	203.0
laplacian_filter6	204.0	95.4	80.9	81.6	111.0
laplacian_filter7	204.0	94.2	80.9	81.9	111.0
laplacian_filter8	182.0	93.9	74.3	72.5	88.6

表 8. gcc 4.8 において OpenMP を使いながら、NEON エンジンを使用した時の最適化オプションによる処理時間

	NEON −O無し (ms)	NEON -O1 (ms)	NEON -O2 (ms)	NEON -O3 (ms)	NEON -Os (ms)
laplacian_filter1	210.0	117.0	99.9	103.0	210.0
laplacian_filter2	234.0	107.0	94.6	95.7	144.0
laplacian_filter3	228.0	99.4	85.6	90.4	95.0
laplacian_filter4	173.0	89.4	76.8	73.6	90.0
laplacian_filter5	213.0	109.0	96.1	96.5	209.0
laplacian_filter6	190.0	92.3	80.2	79.6	110.0
laplacian_filter7	190.0	92.5	80.5	79.8	111.0
laplacian_filter8	172.0	89.0	73.0	72.8	89.5

表 9. Vivado HLS を用いてラプラシアンフィルタ処理をハードウェア化した時の処理時間

実装	最速のソフトウェアの処理時間(ms)		ハードウェアの処理時間(ms)		SW/HW
(ソフトウェア実装)	SW	条件	HW	条件	
laplacian_filter1 (laplacian_fiter5)	95.2	OpenMP -O3	509	PIPELINE無し	0.19
laplacian_filter2	74.9	NEON -O3	80	PIPELINE無し、memcpy()	0.94
laplacian_filter3	74.4	NEON -O3	60	PIPELINE有り、memcpy()、割り算無し	1.24
laplacian_filter4 (laplacian_filter8)	72.5	OpenMP -O3	15.3	PIPELINE有り、memcpy()、割り算無し、高速化技法多数	4.74

6.5 Vivado HLS を用いてハードウェア化した場合

ソフトウェアでのラプラシアンフィルタの実装と殆ど同一の C ソースコードを Vivado HLS を使用して高位合成によりハードウェアにした時のラプラシアンフィルタ処理時間を表 9 に示す [1][6][7]。ソフトウェア処理時間はそれぞれの実装で最速の処理時間を表示したが、ソフトウェアで OpenMP を使用した場合は laplacian_filter1 ~ laplacian_filter4 の代替として laplacian_filter5 ~ laplacian_filter8 を使用した。Vivado HLS でハードウェア化した実装はバス・インターフェースに AXI4 Master を使用し、条件に書いてある通りにディレクティブを追加して最適化してある。最速は、Vivado HLS でハードウェア化した laplacian_filter4 の 15.3 ms だった。このハードウェア実装とソフトウェア実装のラプラシアンフィルタ処理時間の倍率は 4.74 倍で、高位合成によりハードウェア化した方がソフトウェアよりも性能が向上した。但し、laplacian_fitler1 では、ハードウェア実装とソフトウェア実装との処理時間の倍率は 0.19 倍で、ソフトウェアの方が高速だった。よって、ソフトウェアとハードウェアの性能は C ソースコードの実装に依存することが分かった。具体的に性能を向上させるためのノウハウを下記に示す。

- memcpy()を使用して、まとまったデータ単位を一度に転送すると AXI4 Master インターフェースのバースト転送を使用することができて性能が向上する。
- 適当な for()ループに PIPELINE ディレクティブを付加するとパイプラインが実装されてスループットが向上する。但し大きな範囲に適用するとツールの計算も膨大となり、待ち時間も大きくなる。またリソースも消費してしまうので、注意が必要である。
- 割り算はレイテンシが大きいのでなるべく使用しないこと。
- 高い性能を必要とする場合は、合成後のハードウェアを推測しながら、最適な C ソースコードの構成を考慮し、適切なディレクティブを追加することが必要である。

6.6 HDL を用いてハードウェア化した場合

最後に、Verilog HDL (Verilog Hardware Description Language) を用いてハードウェア化した場合のラプラシアンフィルタ処理時間について述べる。通常、ハードウェアでラプラシアンフィルタを構成する場合は、カメラからのストリーム・データを逐次的にラプラシアンフィルタ処理をすることが一般的である。その実装では、DDR3 SDRAM をフレームバッファとして使用しないので、メモリ帯域に余裕ができるのがメリットである。ハードウェアでラプラシアンフィルタを構成する場合の一般的なブロック図を図 15 に示す。

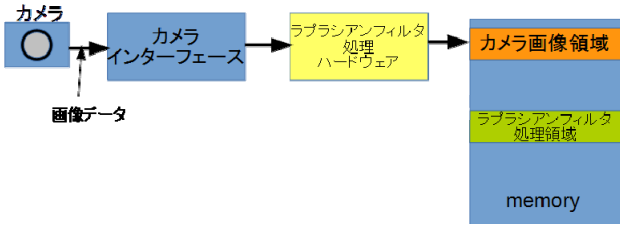

図 15. ハードウェアでラプラシアンフィルタを構成する場合の一般的なブロック図

今回の Verilog HDL による実装では、ソフトウェアの実装に合わせるためにわざと DDR3 SDRAM のカメラ画像領域から読み込んで、ラプラシアンフィルタ処理後にラプラシアンフィルタ処理領域に書き込んでいる（図 8 参照）。Verilog HDL を使用したラプラシアンフィルタ処理時間は 10.1 ms[8] で、全体を通して最速だった。

これは、C 言語に比べて低レベルな言語である Verilog HDL で完全なパイプラインを書くことができたためである。図 8 において、memory のカメラ画像領域から読み込んだカメラ・データが揃った時点で、すぐにラプラシアンフィルタ処理を開始している。ラプラシアンフィルタ処理は、1 クロックごとに 1 ピクセルの処理を完了できる。よって、カメラ画像領域からの読み込みが 1 ライン分終了した時点で、ラプラシアンフィルタ処理は終了している。ラプラシアンフィルタ処理されたデータは、直ちにラプラシアンフィルタ処理領域に書き出すことができる。

メモリ帯域に余裕があれば、カメラ・データの読み込みとラプラシアンフィルタ処理後のメモリへの書き出しが同時にできるので、更に性能向上が可能である。

7. まとめ

Xilinx 社の Zynq を使用した学生実験用のソフトウェアとハードウェア性能比較実験装置を作製した。ラプラシアンフィルタ処理について、最適化レベルの異なるソフトウェア実装、NEON エンジンを使用したソフトウェア実装、OpenMP を使用したソフトウェア実装、Vivado HLS で C ソースコードをハードウェア化した実装、HDL でのハードウェア

実装で性能を比較することができた。その結果として次に示す事がわかった。

- Vivado HLS でハードウェア化して性能が良い C ソースコードは、一般的にソフトウェアでも性能が良い。
- gcc のオートベクタライズ機能を使用して NEON を使用すると性能が向上することが多い。
- gcc の OpenMP 自動並列化オプションを使用すると性能が向上する場合が多いが、バグが出ないように C ソースコードの構造を考慮する必要がある。

8. 謝辞

Vivado HLS の最適化された C ソースコードを提供して下さった株式会社イメージ･テック 植田友幸様に深く感謝致します。様々な面で、ご支援を頂いた筑波大学システム情報系の和田耕一教授に深く感謝いたします。今回の実験装置の製作について科研費奨励研究 (15H00371)のご支援を頂きました。感謝致します。

参考文献

[1] 小野 雅晃, C-6「Xilinx 社の FPGA における高位合成ツール Vivado HLS の効果と性能」発表資料, Design Solution Forum 2015, 2015/10/02, http://dsforum.jp/2015/timetable_soft.html#c6
[2] Zynq-7000 All Programmable SoC テクニカル リファレンス マニュアル(UG585), Ver.1.10, 2015/02/23, http://japan.xilinx.com/support/documentation/user_guides/j_ug585-Zynq-7000-TRM.pdf
[3] GitHub, marsee101/laplacian_filters, https://github.com/marsee101/laplacian_filters
[4] 小野 雅晃, gcc と clang で -Ofast 最適化オプションを付けた場合の実行速度, FPGA の部屋, 2015/07/14, http://marsee101.blog19.fc2.com/blog-entry-3199.html
[5] 小野 雅晃, 新しいラプラシアンフィルタのソフトウェア実装, FPGA の部屋, 2015/07/20, http://marsee101.blog19.fc2.com/blog-entry-3203.html
[6] 小野 雅晃, ラプラシアンフィルタのソフトウェアとハードウェアの速度の比較 2, FPGA の部屋, 2015/07/01, http://marsee101.blog19.fc2.com/blog-entry-3189.html
[7] 小野 雅晃, Vivado HLS 2014.4 で合成したラプラシアンフィルタ IP の高速化 14（性能が最大になる設定を探る 7、まとめ）, FPGA の部屋, 2051/04/15, http://marsee101.blog19.fc2.com/blog-entry-3120.html
[8] 小野 雅晃, ZYBO に HDL で書いたラプラシアンフィルタを実装する 9（制御ソフトウェアを作製して実機確認）, FPGA の部屋, 2015/11/09, http://marsee101.blog19.fc2.com/blog-entry-3304.html

Development of experiment device for comparing software and hardware performance

Masaaki Ono

Technical Service Office for Systems and Information Engineering, University of Tsukuba,
1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8573 Japan

Keywords: neural network; Zynq, FPGA;