알테라 SoC로 실행하는 DPD 프로파일링 및 최적화

글/ 리처드 메이든(Richard Maiden), 알테라(Altera) 무선 시스템 솔루션 엔지니어링 수석 매니저

이 글은 알테라(Altera®) SoC 상에서 실행하는 디지털 전치 왜곡(DPD) 알고리즘에 대한 알고리즘 효율성을 분석하는 방법을 설명한다. 최적화를 안내하기 위해 ARM® 개발 스튜디오 DS-5™(Development Studio 5) 알테라 에디션 툴킷에 포함된 스트림라인 툴을 사용하여 설계 공간을 분석하고, 코드를 프로파일링한다. 메모리 다항식에 기반한 DPD 계수 계산 알고리즘은 다수의 행렬 조작을 포함하며, 알테라 SoC에서 이러한 계산은 소프트웨어와 하드웨어에서 구현할 수 있다. 엔지니어는 스트림라인 툴을 이용해 알고리즘 단계 동안 프로세서가 어떻게 로드되는지 시각화할 수 있으며, 이러한 정보를 사용하여 구조적 기능의 장점을 적용하고 컴파일러에 지시하도록 코드를 재구성할 수 있다. 또한 최적화 흐름의 각 단계마다 예제, 권고, 결과가 제공된다.

가장 먼저 DPD 알고리즘의 기본적인 바닐라 구현을 C 코드로 작성했으며, 다음으로 아래와 같은 최적화 단계가 수행되었다.

- 정밀도 최적화
- 알고리즘 최적화
- 컴파일러 최적화 및 옵션
- NEON™ 기술을 이용한 벡터화
- 멀티스레딩
- FPGA 가속화

소 개

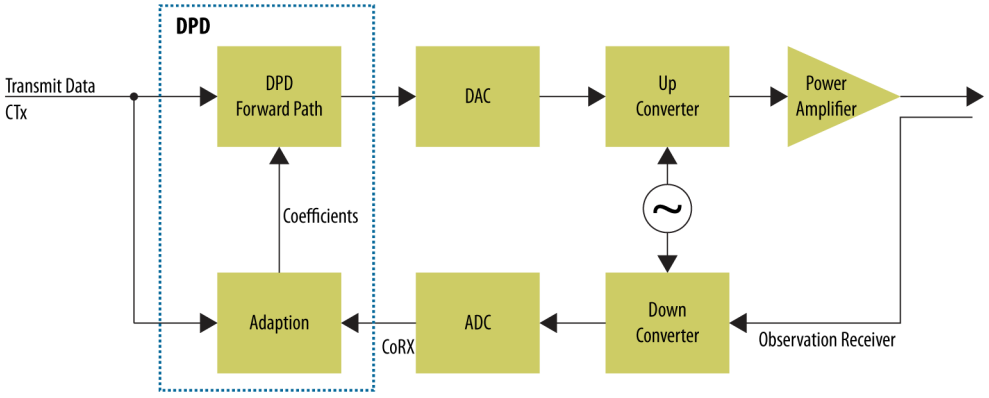
DPD는 원격 라디오 헤드(RRH, remote radio head)의 핵심 기능이다. 높은 전력 효율을 달성하기 위해 전력 증폭기(PA)는 비선형적인 동작 영역 내로 구동된다. DPD 알고리즘은 시스템의 출력을 선형화하고 표준 기관에서 요구하는 스펙트럼 방사 요구사항을 만족할 수 있도록 이러한 비선형 특성의 역을 적응형 방식으로 적용하는 데 사용된다.

알테라의 SoC는 하드웨어(FPGA) 또는 소프트웨어 하드 프로세서 시스템(HPS)에서 이러한 알고리즘을 구현하는 플랫폼을 제공한다. 이 플랫폼은 설계자에게 DPD를 최적으로 구현하도록 결정할 수 있는 유연성을 제공한다. 이 밖에 알테라와 ARM은 설계자와 개발자가 하드웨어와 소프트웨어를 간편하게 설계, 시뮬레이션, 구현 및 디버그할 수 있게 도와주는 툴을 제공한다. FPGA에서 DSP 빌더와 Quartus® II 소프트웨어 같은 툴을 사용하면 특정 디바이스의 기능에 대한 최적화된 구현을 자동으로 생성할 수 있다. HPS에서는 컴파일러 설정과 운영 시스템을 고려해 소프트웨어 코드를 통해 최적화를 달성할 수 있다.

여기에 설명되는 DPD 알고리즘은 두 개의 서브시스템으로 구성된다. DPD 적응은 적절한 보정

계수를 결정하는 과정이며, 순방향 경로는 이러한 보정 계수를 데이터 경로에 적용하는 과정이다. 적응 서브시스템은 기준 신호로 송신된 데이터로부터 샘플을 캡처하고, 이를 PA로부터 출력된 신호의 관찰 경로에 의한 동시 캡처된 신호와 비교함으로써 지속적으로 PA 특성에 대한 추정을 수행한다. DPD 순방향 경로 서브시스템은 적응 엔진에 의해 생성된 계수를 적용하는 필터 탭으로 배열된다.

그림 1. DPD **및 전력 증폭기의 블록 다이어그램**



샘플은 전력 증폭기의 입력(CTx)과 전력 증폭기의 출력(CoRx)으로부터 취한다. CoRx 벡터는 다항식 행렬인 'G' 행렬을 생성하는 데 사용된다. G 행렬을 생성한 후 전력 증폭기의 새로운 역 계수를 얻기 위한 연산은 다음과 같은 방정식으로 나타낼 수 있다.

$$coefficients = [INV(G' \times g) \times (G' \times CTx)]$$

아래의 표는 방정식이 코드에서 다섯 가지 함수로 구성되는 것을 보여준다. 각 함수의 출력은 다음 함수에 대한 입력으로 사용되며, 각 함수는 순차적으로 실행된다.

표 1. DPD **함수의 구성** (1/2)

함수 이름	설명	입력 행렬의 크기	수행되는 연산
CovarMatrix()	CoRx 벡터로부터 생성기 행렬(G 행렬) 계산	2048 x 1	G 행렬은 CoRx 벡터의 다항식 항으로 구성된 행렬이다. 출력 행렬의 크기: (36 x 2040)
MM1_Gt_x_G()	입력으로 G 행렬을 사용하여 G' x G를 계산	2040 x 36	G 전치 행렬과 G 행렬의 행렬 곱셈은 크기가 (36 x 36)인 정방 행렬이 된다. (36 x 2040) *(2040 x 36) = 36 x 36
MM2_Gt_x_CTx()	입력으로 G 행렬과 CTx 벡터를 사용하여 G' x CTx 계산	2040 x 36 및 2040 x 1	G 전치 행렬과 CTx 벡터의 행렬 곱셈은 크기가 (36 x 1)인 벡터가 된다. (36 x 2040) * (2040 x 1)=(36 x 1)
MI_inv()	MM1_Gt_x_G 함수로부터 얻은 (G' x G) 정방 행렬의 역행렬을 수행	36 x 36	차원에 대한 정방 행렬의 역행렬

표 1. DPD 함수의 구성 (2/2)

함수 이름	설명	입력 행렬의 크기	수행되는 연산
MI_x_MM2()	MI_inv 및 MM2_Gt_x_CTX 함수로부터 행렬 출력의 행렬 곱셈을 계산	36 x 36 및 36 x 1	G' x CTx 연산으로부터 역행렬과 벡터의 행렬 곱셈. 기본적으로 Inv(GT x G) x (GT x CTx)를 의미한다. (36 x 36) * (36 x 1) 출력 행렬의 크기: (36x1) 이 연산의 결과는 DPD의 순방향 경로에 적용되는 계수이다.

온도, 전력 레벨 변화 및 기타 요소들이 PA 특성에 영향을 미치므로 DPD는 가능한 신속하게 이러한 변화에 적응해야 한다. 따라서 적응 반복 간 지연은 최소화해야 한다. 최근의 원격 라디오 헤드는 주로 MIMO 시스템에 사용되며, 일반적으로 이러한 라디오 헤드 구현은 단일 적응 엔진을 사용한다. 각각의 안테나는 자체적인 DPD 순방향 경로를 필요로 하기 때문에 적응 엔진은 다수의 안테나 경로들 간에 공유된다.

알테라 SoC의 개요

알테라 SoC는 FPGA 패브릭을 통합한 차세대 집적 고성능 애플리케이션 프로세서이다(그림 1 참조). 알테라 SoC는 고대역폭 인터커넥트 백본을 사용하는 FPGA 패브릭과 함께 듀얼 코어 ARM 프로세서, 주변장치, 메모리 컨트롤러로 구성된 ARM 기반 HPS를 통합하고 있다.

ARM 기반 하드 프로세서 시스템

HPS는 듀얼 코어 ARM Cortex®-A9 MPCore™ 프로세서, 다양한 주변장치, 그리고 FPGA의 로직과 공유되는 멀티포트 메모리 컨트롤러로 구성된다. HPS는 하드 IP(intellectual property)의 성능 및 비용 절감과 프로그래밍 가능한 로직의 유연성을 모두 제공한다.

임베디드 주변장치는 이러한 기능을 프로그래머블 로직에 구현할 필요를 없애주므로, 애플리케이션에 특정한 커스텀 로직에 더 많은 FPGA 자원을 할당하고 전력 소모를 줄일 수 있게 한다.

하드 멀티포트 SDRAM 메모리 컨트롤러는 프로세서와 FPGA 로직 간에 공유되며, 에러 교정 코드(ECC) 지원과 함께 DDR2 SDRAM, DDR3 SDRAM, LPDDR2 디바이스를 지원한다.

고속 인터커넥트

HPS와 FPGA 패브릭 사이의 높은 쓰루풋 데이터 경로는 2칩 솔루션으로는 달성할 수 없는 인터커넥트 성능을 제공한다. HPS와 FPGA 패브릭의 긴밀한 통합은 125Gbps 이상의 피크 대역폭뿐 아니라 프로세서와 FPGA 간의 통합된 데이터 일관성을 제공한다.

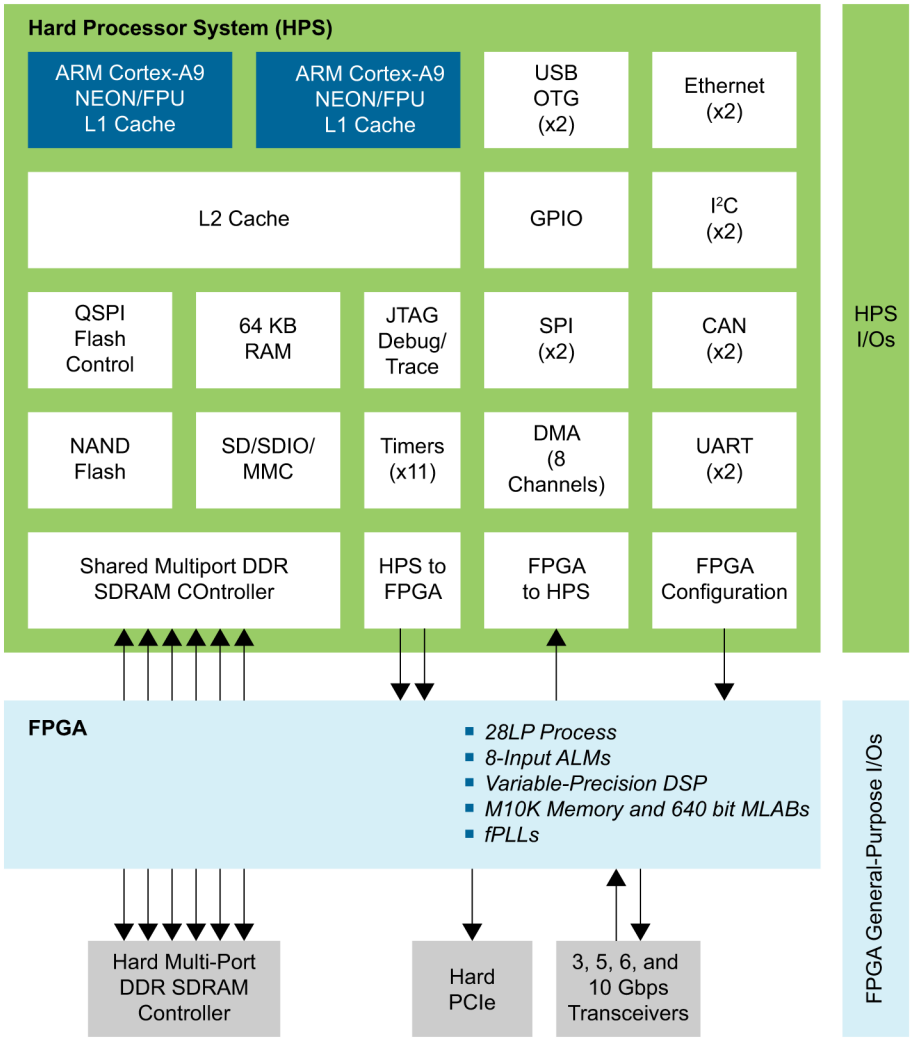
유연한 FPGA 패브릭

FPGA 로직 패브릭에 의해 제공되는 유연성은 설계 안에 커스텀 IP나 알테라 또는 협력사의 사전 구성된 상용 IP를 구현함으로써 시스템을 차별화할 수 있게 한다. 또한 다음을 수행할 수 있다.

- 변화하는 다양한 인터페이스와 프로토콜 표준에 빠르게 적응할 수 있다.
- 커스텀 하드웨어를 FPGA에 추가함으로써 시간에 민감한 알고리즘을 가속화할 수 있다.

FPGA 내에 구현된 하여 하드 로직 기능은 PCI Express®(PCIe®) 포트와 추가적인 멀티포트 메모리 컨트롤러를 포함해 전력 소모와 FPGA 리소스 요구사항을 감소시킨다.

그림 2. 알테라 Cyclone® V SoC 블록 다이어그램



스트림라인 툴의 개요

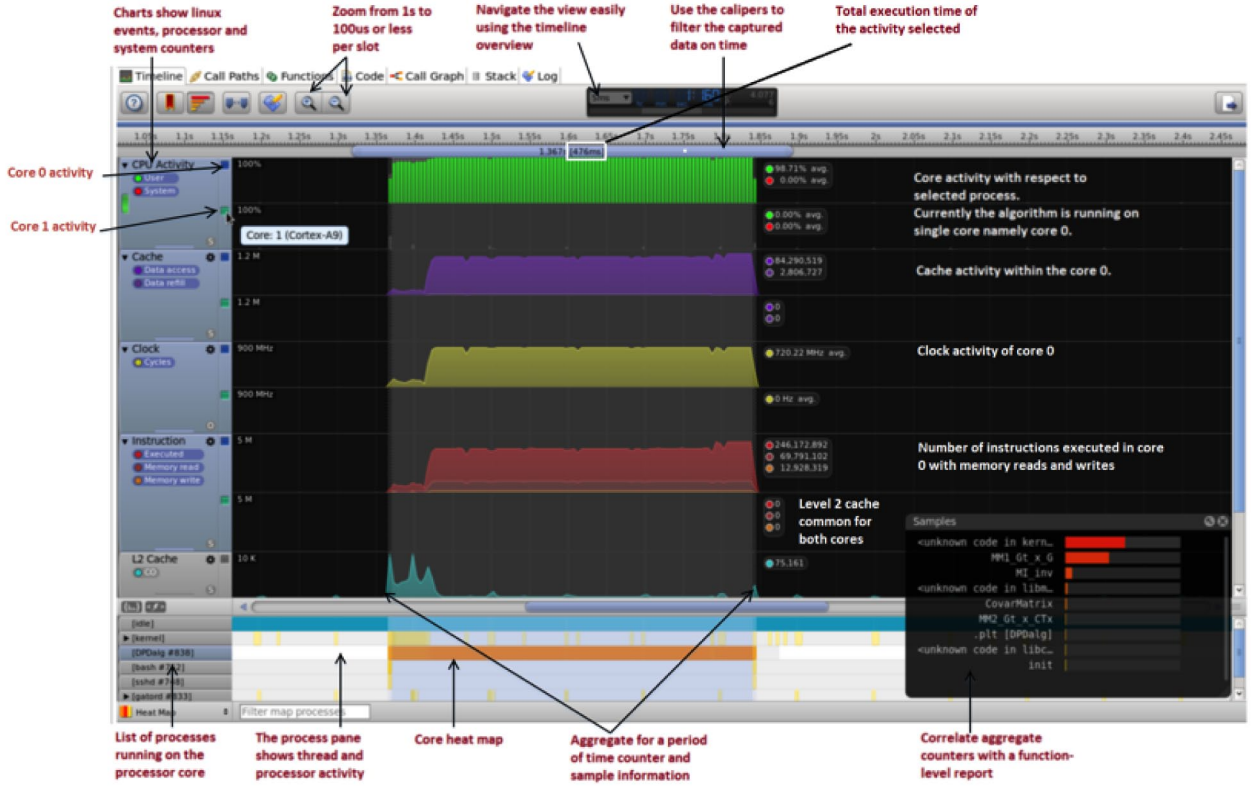
ARM DS-5 알테라 에디션 툴킷(Altera Edition Toolkit)은 알테라 SoC 임베디드 디자인 수트(EDS)에 포함되며, 알테라 SoC 상에서 임베디드 소프트웨어 개발을 위한 종합적인 임베디드 개발 툴을 제공한다. 스트림라인은 코드가 어떻게 실행되는지 그에 대한 직관적 통찰력과 코드 최적화를 위한 풍부하고 유용한 정보를 제공하는 성능 분석 툴이다.

이 글은 스트림라인 툴을 사용해 ARM 코어에서 코드를 실행하는 데 필요한 실행 시간을 시각화하여 추가적인 최적화 단계를 결정할 수 있는 최적화 흐름을 설명한다. C 코드를 오브젝트 코드로 생성하면, 이 코드는 스트림라인 툴에서 실행할 수 있다. 코어 동작을 프로파일링 하려면 드라이버와 gator daemon을 실행해야 한다.

타임라인 뷰

스트림라인 화면은 코어에서 실행하는 다양한 동작의 타임라인을 보여준다. 그림 3은 첫 번째 탭 타임라인 뷰이다. 이 탭은 시스템의 수행을 한눈에 보여주는 대시보드 개요를 제공한다. 캘리퍼를 사용해 타임라인의 한 부분을 선택하면 녹색 막대 위에 코드의 실행 시간이 표시된다. 메인 창은 코어, 메모리, 클록, 명령 동작을 보여준다.

그림 3. DS-5 스트림라인의 타임라인 뷰



'샘플' 헤드업 디스플레이(HUD, heads up display)는 선택된 횡단면에 시간이 어디에서 소요되었는지를 나타내는 컬러 히스토그램을 제공한다. 디버그 기호가 표본화된 함수에 제공되는 경우, 함수 이름이 샘플 HUD에 표시된다. 기호가 없는 샘플의 경우에는 프로세스 또는 공유된 라이브러리의 이름이 꺾쇠 괄호〈 〉 안에 표시되며, 이는 일부 알려지지 않은 함수 안에서 시간이 소요되었다는 것을 나타낸다. 이러한 알려지지 않은 함수는 주로 라이브러리 액세스와 커널 동작이다. 프로세스 창(process pane)은 리눅스 커널 스케줄러로부터 유추된 능동 프로세스를 보여주는 뷰를 제공한다. 샘플 세부사항 창(Samples detail pane)은 선택된 프로세스 기간 동안 함수에 의한 동작을 보여준다.

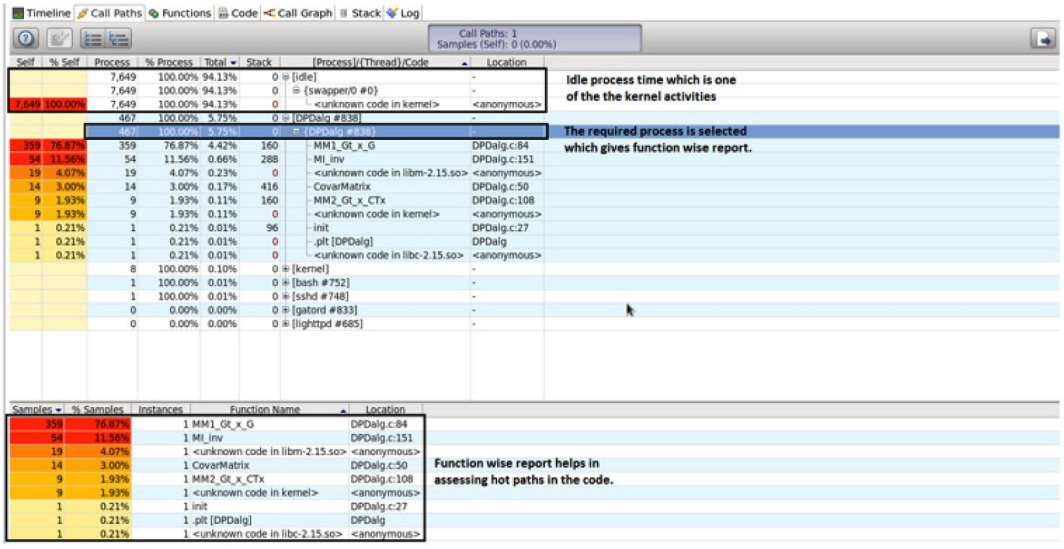
호출 경로 뷰

호출 경로 뷰는 코드의 통계를 보는 데 사용되며, 코드에서 가장 높은 지연을 발생시키는 라인이나 함수를 식별하는 데 도움을 준다(그림 4 참조).

호출 경로 뷰는 다음과 같은 열 헤더(column header)를 갖는다.

- *Self* – Self 시간은 함수 또는 프로세스에서 소요된 시간을 나타낸다. 이 시간은 프로그램이 인터럽트 이벤트에 프로그램 호출 스택을 기록할 때 측정된다. 여기에는 이후의 함수들에서 소요되는 시간은 포함되지 않는다. 여기에서 시간의 양은 샘플의 총수로 보고된다.
- *Process* – 이 함수를 실행하는 데 소요된 총 시간을 나타내며, 이후의 함수를 실행하는 데 소요된 시간을 포함한다.
- *Total* – Total 시간은 함수와 함수가 호출한 모든 함수에 의해 사용된 시간의 양을 나타낸다.
- *Stack* – 이 함수에서 스택에 의해 사용된 바이트 수. 함수의 스택 사용을 알 수 없는 경우 물음표가 표시된다.
- *Process/Thread/Code* – 모든 프로세스, 스레드 및 함수를 보여주는 계층적 뷰. 모든 [프로세스]는 리스트에서 대괄호 안에 표시되며, {스레드}는 중괄호로 묶는다.

그림 4. DS-5의 스트림라인 툴에서 함수 단위로 생성된 부하를 보여주는 호출 경로 뷰



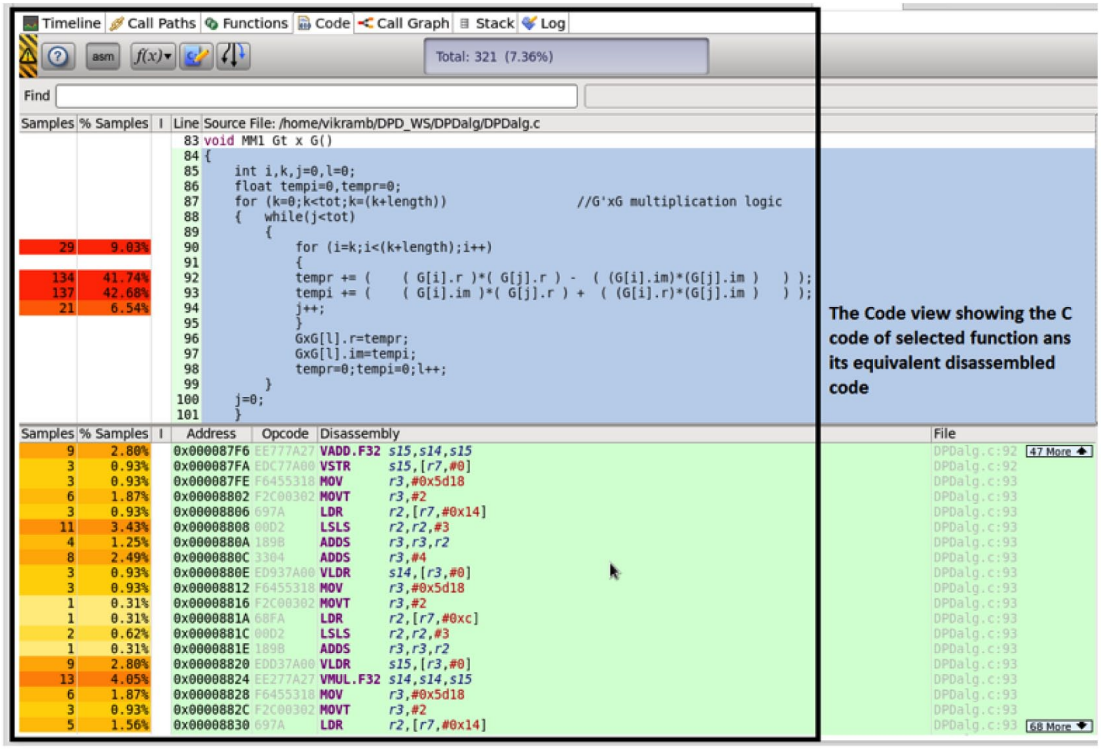
위의 예는 어느 함수가 총 실행 시간에서 가장 많은 시간을 소비하는지를 얼마나 쉽게 식별할 수 있는지를 보여준다. 따라서 최적화 노력을 어느 곳에 집중해야 하는지 명확한 방향을 알려준다.

코드 뷰

코드 뷰는 함수 레벨의 '핫스팟(hot spot)'을 발견하는 데 도움을 준다. 코드 뷰는 통계를 압축해 이를 소스 및 디스어셈블리 레벨로 표시한다. 디폴트로 코드 뷰는 소스 코드와 그 옆에 컬러 코드 통계를 함께 표시한다. 각 항목에 대한 기여율(percentage contribution)이 함수에 대해 표시된다. 이 뷰는 또한 코드가 실행되는 동안 수집된 샘플을 보여준다. 샘플은 고정된 시간 간격으로 툴

에 의해 수집되며, 코드의 매 라인에 대해 수집된 샘플의 수는 핫스팟 경로를 알 수 있게 한다. 디스어셈블리 뷰는 벡터화 등 사용된 명령 유형을 식별하는 데 사용할 수 있다.

그림 5. DS-5 스트림라인 툴의 코드 뷰



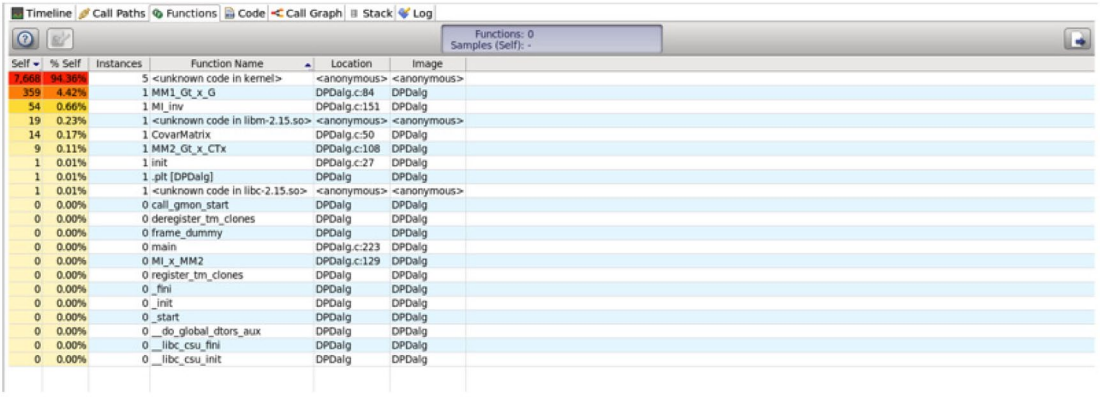
함수 뷰

함수 뷰는 사용 데이터와 함께 캡처 세션 동안 호출된 모든 함수의 리스트를 보여준다. 함수 뷰는 다음과 같은 열 헤더를 갖는다.

- *Self* – Self 시간은 함수에서 소요된 시간을 나타내며, 이후의 함수에서 소요되는 시간은 포함되지 않는다. 여기에서 시간의 양은 샘플의 총수로 보고된다.
- *Instances* – 함수가 호출 경로 뷰에 나타난 횟수
- *Location* – 이 열은 함수의 위치를 보고하며, 선언의 파일 이름과 라인을 모두 표시한다.
- *Image* – 함수를 포함하는 이미지 파일

함수 뷰는 프로파일링 기간 전체에 걸쳐 수집된 샘플을 보여준다. 이와 달리 호출 경로 뷰는 캘리퍼에서 설정된 기간을 반영하는 프로세스와 샘플 비율을 보여준다. 호출 경로 뷰에서는 요구되는 프로세스나 애플리케이션과 관련해 함수에 대한 부하 비율을 볼 수 있다. 이 DPD 알고리즘에서는 어떤 함수가 캡처 시간 동안 다른 함수와 관련해 가장 많은 시간을 소모하는지 명확하게 알 수 있다.

그림 6. DS-5 스트림라인 툴의 함수 뷰

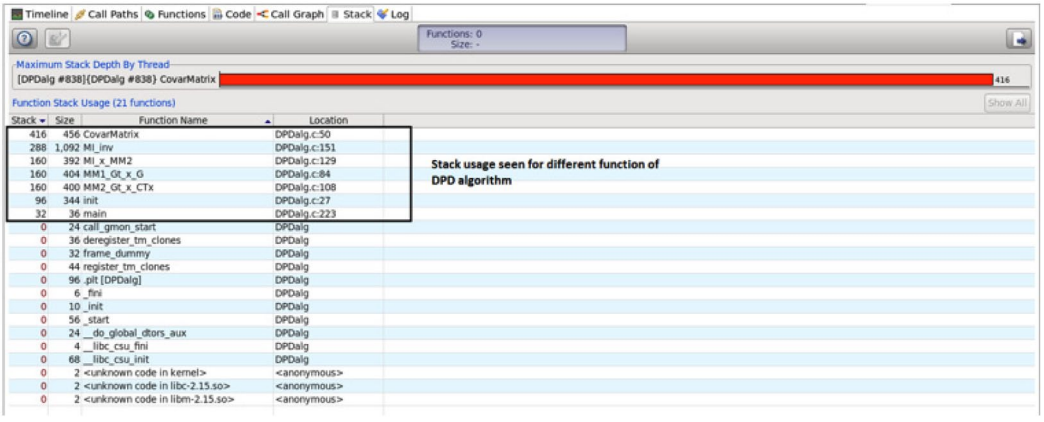


스택 뷰

스택 뷰는 최대 스택 깊이에 미치는 영향의 정도에 따라 함수에 순위를 매김으로써 스택 최적화를 위한 최적의 대상을 제공한다. 스택 뷰는 다음과 같은 열 헤더를 갖는다.

- *Stack* – 이 함수에서 스택에 의해 사용된 바이트 수. 함수의 스택 사용을 알 수 없는 경우 여기 총수(total) 옆에 물음표가 표시된다.
- *Size* – 함수의 전체 크기를 바이트로 표시.
- *Function Name* – 소스 코드에서 지정된 함수의 이름.
- *Location* – 이 열은 함수의 위치를 보고하며, 선언의 파일 이름과 라인을 모두 표시한다.

그림 7. DS-5 스트림라인 툴의 스택 뷰



호출 그래프 뷰

호출 그래프 뷰는 각각의 함수를 호출되는 지점에 따라 배치하고, 화살표를 사용해 호출 함수를 연결함으로써 코드 계층을 시각적으로 보여준다. 이 뷰는 자신의 계층에서 핫스팟을 빠르게 식별하는 데 사용할 수 있다. 호출 그래프 뷰는 다음과 같은 요소로 구성된다.

- *Function boxes* – 호출 그래프 뷰는 호출 계층에서 각 함수에 대해 하나의 함수 상자를 그린

다. 호출 그래프 뷰는 중요한 함수를 호출 그래프 뷰에서 빠르게 식별할 수 있도록 전체 샘플에 따라 함수를 컬러 코딩한다. 이들 컬러 범위는 밝은 빨간색에서부터 밝은 노란색까지이며, 빨간 색은 최고값을, 밝은 노란색은 최저값을 갖는다. 컬러는 미니맵에서 쉽게 식별할 수 있어, 중요한 함수로 빠르게 스크롤 할 수 있다.

- *Call arrows* – 화살표 방향은 어떤 함수가 호출 함수(calling function)인지를 나타낸다. 함수를 가리키는 화살표는 함수가 호출 당하는 함수(callee)이며, 라인이 시작되는 함수가 호출 함수라는 것을 알려준다.
- *Mini-map* – 호출 그래프 뷰의 왼쪽 아래 모서리에는 미니맵이 있다. 계층이 너무 커서 Eclipse의 에디터 섹션에 맞추기 어려울 때 이 미니맵을 사용하면 호출 그래프 뷰 안을 쉽게 이동할 수 있다.

단계 1: Numeric 정밀도 분석

Numerical 정밀도는 설계 사이클 초기에 고려해야 하는 중요한 분석이다. 이 정밀도는 지연을 직접적으로 결정할 뿐 아니라 어떤 자원을 사용할 수 있는지 지시한다. 예를 들어 NEON 액셀러레이터는 배정밀도 데이터 유형에서는 사용할 수 없다.

DPD 알고리즘의 경우 각 함수는 개별적으로 고려되며, MATLAB 레퍼런스에 대해 비교된다. 결과는 표 2에 제공된다. 원하는 정확도와 지연을 달성하기 위해 필요한 정밀도와 관련하여 이 보고서에 기반해 결정을 내릴 수 있다.

표 2. DPD 함수 단위로 생성된 에러와 지연 비교

함수	단정밀도		배정밀도	
	에러	지연(ms)	에러	지연(ms)
CovarMatrix()	수용 가능	27	수용 가능	34
MM1_Gt_x_G()	수용 가능	321	수용 가능	338
MM2_Gt_x_CTx()	수용 가능	21	수용 가능	29
MI_inv()	초과	53	수용 가능	65
MI_x_MM2()	수용 가능	6	수용 가능	10
총 시간		428		476

지연을 최소화하려면 모든 경우에 단정밀도를 선택하는 것이 바람직하지만, 역행렬 단계의 MI_inv() 함수에서 발생하는 에러의 영향은 수용하기에는 너무 높다. 다른 모든 함수에서는 단정밀도가 선택되었다.

단계 2: 알고리즘 최적화

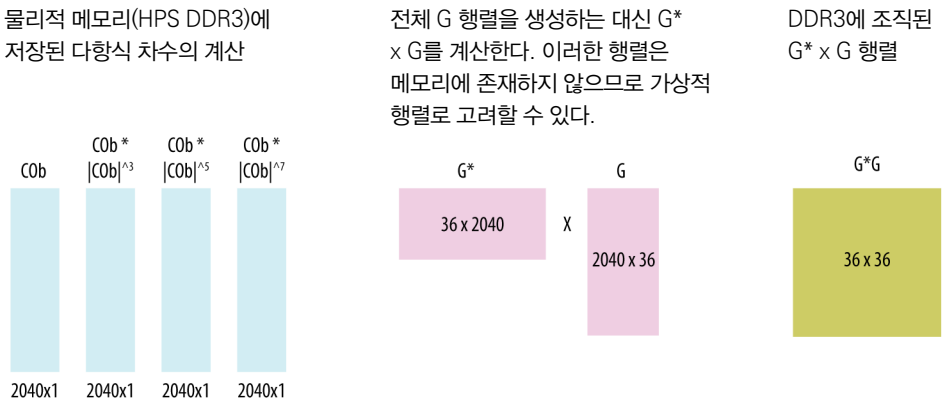
지연을 감소시키기 위해 다음의 두 가지 알고리즘 최적화가 수행되었다.

- G 행렬 계산에서 대칭을 이용한 곱셈의 수 줄이기
- 출력 행렬의 대칭적 특성을 이용한 행렬 곱셈 최적화

G 행렬 계산의 최적화

CovarMatrix() 함수는 G 행렬을 계산한다. G 행렬은 기본적으로 CoRx 백터의 서로 다른 거듭제곱에 대한 주기적 배열을 갖는 다항식 행렬이다. 여기에서는 전체 G 행렬을 계산하는 대신, 서로 다른 거듭제곱의 개별 벡터를 생성한다. 크기가 2040x4인 부분 G 행렬이 생성되며, 이후의 행렬 곱셈 연산은 이러한 다항식 벡터를 사용하여 직접 수행된다.

그림 8. 전체 G 행렬 계산 대신 다항식 벡터의 계산에 의해 달성되는 알고리즘 최적화



위의 알고리즘은 다음의 분석을 기반으로 구현된다.

1. G 행렬에서 행의 수 = COb_Size−M = 2048−8 = 2040.
2. G 행렬에서 열의 수 = K x MD = 4 x 9 = 36
3. G 행렬의 각 열은 다음의 계산을 사용해 생성된다.
   - [col vector]2040x1 .* (abs([col vector] 2040x1))O(k)
     a. 절대 복소 배열
     b. 3, 5, 7 등의 거듭제곱을 취한다. 이것은 크기가 2040인 원소와 원소의 곱이다.
     c. 행렬 G는 배열에서 기본적으로 적절한 위치에 일차원 배열과 포인터를 사용하여 구성된다. 따라서 G 행렬은 물리적으로 생성되지 않으며, G*G 행렬은 직접 계산된다.
4. 2040x1 배열의 각 원소는 단정밀도 부동 소수점으로 복소수를 저장한다. 따라서 4가지 배열은 총 4x2040x8 = 65280 바이트를 사용한다. G와 G*를 구성하지 않는 방식으로 인해 절약되는 메모리는 2x36x2040x8 = 2x587520 bytes = 1175040 바이트(1.175MB)이다.

행렬 곱셈의 최적화

행렬 곱셈의 실행 속도를 향상시키기 위해 사용되는 또 다른 최적화 기법은 '전치로 곱한 행렬은 정방 대칭 행렬을 생성한다'라는 수학적 규칙이다. 이것은 GT x G 연산을 수행하면 정사각형이면서 대칭인 행렬이 나온다는 것을 의미한다. 이러한 특성은 복소수 곱셈의 수를 거의 절반으로 줄일 수 있게 한다.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix}$$

아래 삼각행렬(lower triangular matrix)의 원소는 위 삼각행렬(upper triangular matrix)로부터 얻을 수 있다. 계산의 수는 다음과 같이 줄일 수 있다.

$$\frac{n^2}{2} - \frac{n}{2}$$

각 원소의 계산은 'n' 복소수 곱셈과 'n-1' 덧셈을 한다. 각각의 복소수 곱셈은 4번의 실수 곱셈과 2번의 실수 덧셈에 의해 수행된다. 전체 절약은

$4n \times \left(\frac{n^2}{2} - \frac{n}{2}\right)$ 곱한 후 $2(n-1) \times \left(\frac{n^2}{2} - \frac{n}{2}\right)$ 더한다.

알고리즘 최적화 후 ARM 프로파일링

프로파일링은 위에서 언급한 알고리즘 수정을 통합한 후 코드에 대해 수행된다.

그림 9. 코드 뷰는 최고 샘플 비율의 라인과 해당 디스어셈블리를 보여준다.

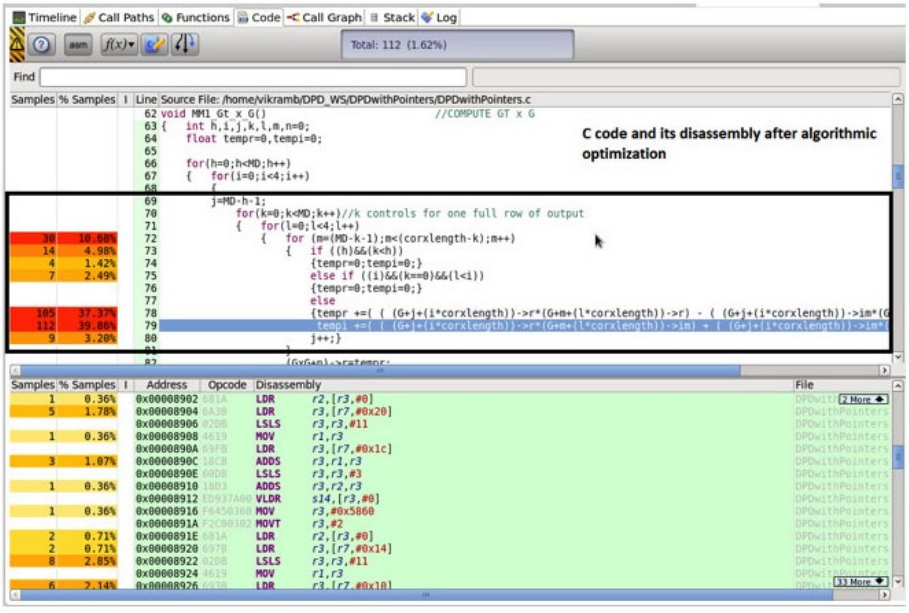
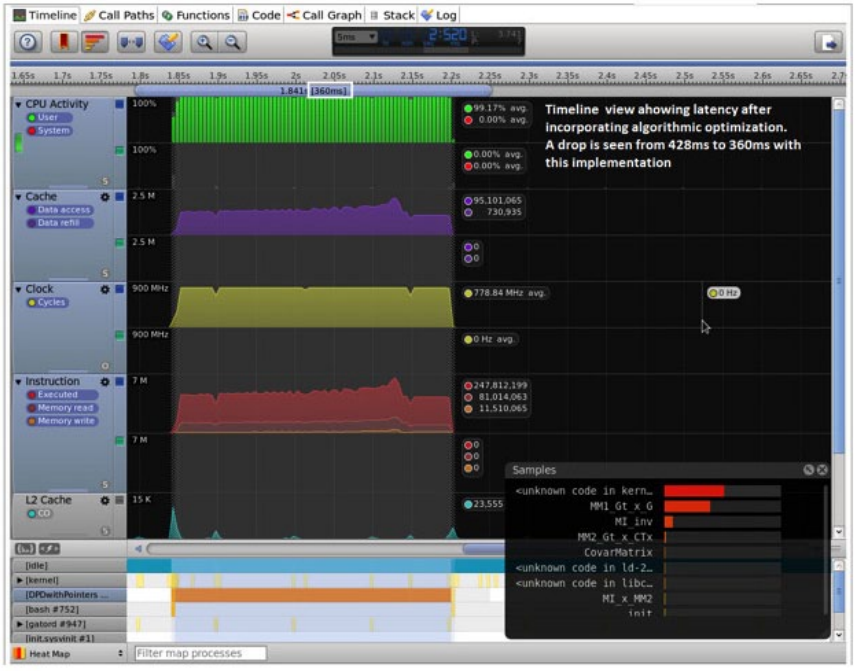


그림 10. 알고리즘 수정을 사용한 ARM 프로파일 보고서



ARM 프로파일 보고서 – 알고리즘 최적화 후

코드는 코드 변경 후 유사한 방식으로 DS-5 툴을 사용하여 프로파일링된다.

결과는 표 3에 제공된다.

표 3. 알고리즘 최적화 후 ARM 프로파일 보고서

번호	함수 이름	예외 시간	
		알고리즘 최적화 전	알고리즘 최적화 후
1	CovarMatrix()	27	16
2	MM1_Gt_x_G()	321	262
3	MM2_Gt_x_CTx()	21	21
4	MI_inv()	58	55
5	MI_x_MM2()	6	6
총 시간		433ms	360ms

단계 3: 컴파일러 옵션을 사용한 최적화

컴파일러 옵션은 최적화를 위한 간편한 방법을 제공한다. 컴파일러 옵션은 어떤 지연 향상을 얻을 수 있는지 결정하기 위해 사용한다. 이러한 이득은 매우 큰 것으로 입증되었다.

GCC는 기본 -O1(기본 최적화, 최고 디버깅)에서부터 -O2(우수한 최적화, 항상 성능 향상), 그리고 -O3(최고 성능, 일부 영역에서 회귀 가능)까지 코드를 컴파일링하는 동안 켤 수 있는 광범위한 최적화 레벨을 갖는다.

컴파일러 옵션 소개

컴파일러를 호출하는 명령은 다음과 같다: armcc [options] [source]

여기서,

- *Options* – 특성 또는 컴파일러에 영향을 미치는 컴파일러 명령행 옵션
- *Source* – C 또는 C++ 소스 코드를 포함하는 하나 이상의 텍스트 파일의 파일 이름을 제공한다.

사용되는 일부 옵션들은 표 4에 설명되어 있다.

표 4. 컴파일러 옵션과 설명

옵션	설명
-arm-linux	이 옵션은 ARM 리눅스 컴필레이션에 적합한 디폴트값으로 다른 옵션 세트를 구성한다.
-gnueabihf	ARM 아키텍처를 위한 GNU C 프로세서(cpp)
--gcc	GCC는 특정 C 코드를 컴파일하기 위한 호출이다.
-g3	최대 디버깅 가능
-Wall	생성된 모든 경고 메시지를 보이게 할 수 있다.
-c	이 옵션은 컴파일러에게 링크 단계를 제외한 컴필레이션 단계를 수행하도록 명령한다.
-O(num)	여기서 num은 다음 중 하나이다. 최소 최적화(-O0): 대부분의 최적화를 끈다. 이것은 가능한 최고의 디버그 뷰와 최저 레벨의 최적화를 제공한다. 디폴트 최적화 레벨. 제한적인 최적화(-O1): 사용하지 않는 인라인 함수와 사용하지 않는 정적 함수를 제거한다. 심각하게 디버그 뷰를 저하시키는 최적화를 끈다. 높은 최적화(-O2): --debug와 함께 사용할 경우 오브젝트 코드와 소스 코드의 매핑이 항상 명확한 것은 아니기 때문에 디버그 뷰는 덜 만족스러울 수 있다. 최대 최적화(-O3): -O2와 동일한 최화를 수행하지만, 생성된 코드에서 공간과 시간 최적화 간 균형이 -O2에 비해 공간 또는 시간으로 더 많이 가중된다.

-O1 컴파일러 옵션을 사용한 ARM 프로파일링

그림 11. -O1 최적화에 대한 타임라인 뷰

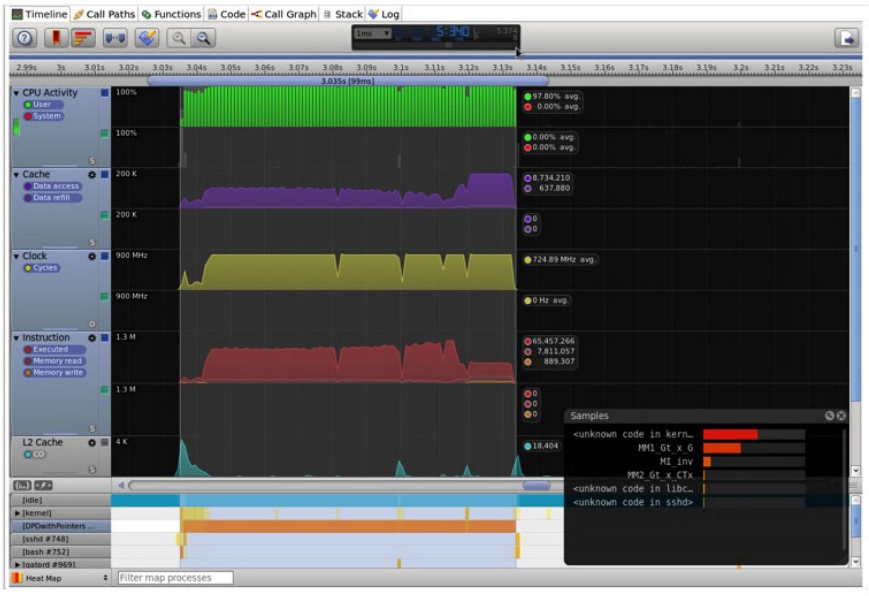
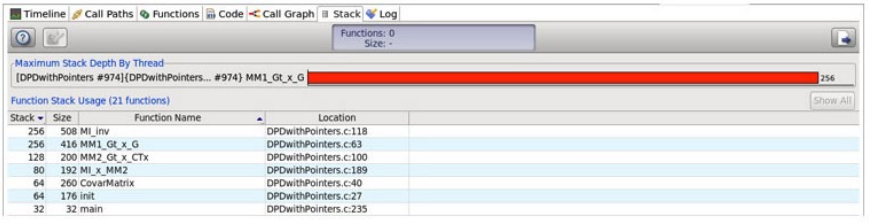


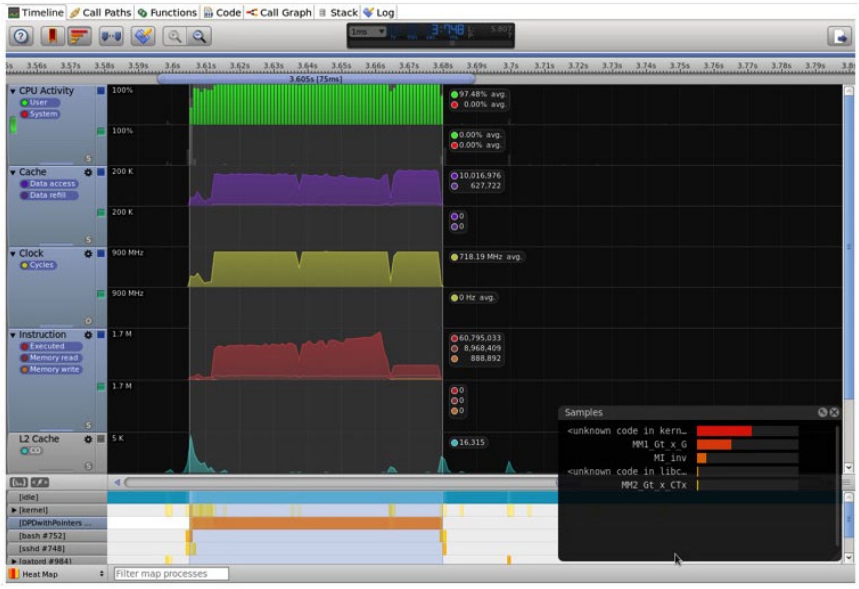
그림 12. -O1 최적화 플래그에 대한 스택 뷰



-O2 컴파일러 옵션을 사용한 ARM 프로파일링

이 최적화 레벨은 -O2로 설정되며, 그림 13의 결과는 -O1 컴파일러 옵션을 사용한 앞의 구현에 비해 더욱 향상된 지연을 보여준다.

그림 13. -O2 최적화를 사용한 타임라인 뷰



-O3 컴파일러 옵션을 사용한 ARM 프로파일링

-O3 플래그는 벡터라이저와 SMS(Swing Modulo Scheduling)와 같은 많은 첨단 기능들을 구현한다. GCC 벡터라이저는 데이터 병렬처리를 갖는 코드를 인식하고, 이를 한 번에 많은 값으로 동일한 연산을 수행하도록 다시 쓴 다음, 등가 병렬 명령으로 변환한다. 최상의 경우는 코드가 8개의 독립적인 단일 바이트 연산을 하나의 8바이트 폭 연산으로 변환할 수 있으며, 이 경우 핫루프를 최대 550%까지 빠르게 만드는 것으로 나타났다.

SMS는 높은 지연, 특히 메모리 부하를 갖는 명령을 포함하는 루프를 인식하고, 이를 프롤로그, 바디, 에필로그로 다시 쓴다. 프롤로그는 맨 먼저 사용 가능한 값을 만들고, 바디의 처음 절반은 n+1 값을 로드하고, 나머지 절반은 n 값을 사용한다. 에필로그는 마지막 값을 마친다. 이것은 특히 높은 부하 지연을 갖는 A9에서 유용하다. 일부 루프는 30% 더 빠르게 실행된다.

이전 구현의 관찰에 의하면 컴파일러 최적화 옵션을 사용하지 않은 코드는 DPD 알고리즘을 구현하는 최상의 방법이 아닐 수 있다는 것을 보여주므로, 따라서 실행 시간을 향상시키기 위해 -O3 컴파일러 플래그를 사용한다. 그림 14의 결과는 컴파일하기 위해 -O3 플래그를 사용한 후 얻은 결과이다.

그림 14. -O3 컴파일러 플래그를 사용한 ARM 프로파일 보고서

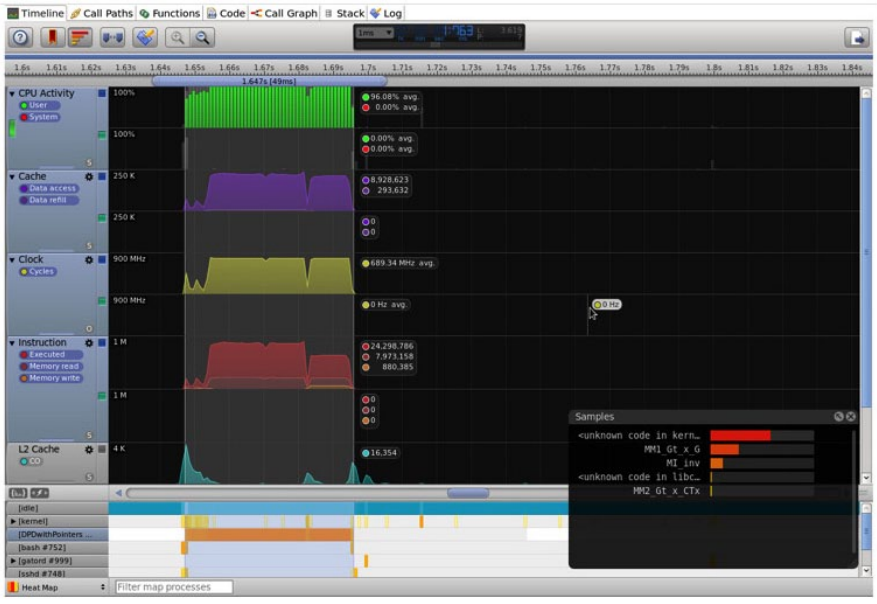
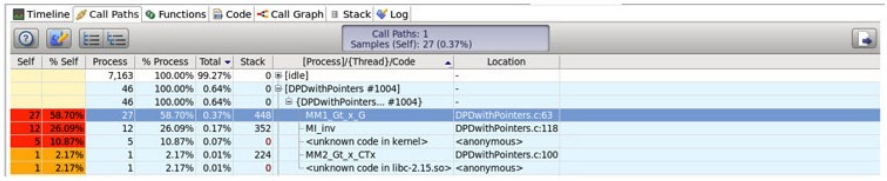


그림 15. -O3 컴파일러 플래그를 사용한 호출 경로 뷰



ARM 프로파일 보고서 – 컴파일러 옵션

표 5는 함수 단위로 생성된 보고서로, 최적화 스위치의 증가하는 레벨과 함께 각 함수를 구현하는 실행 시간을 잘 보여준다.

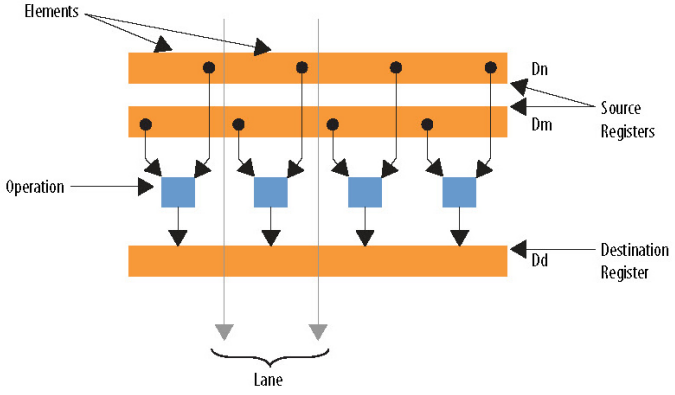
표 5. -O3 플래그를 사용한 ARM 프로파일 함수 단위로 생성된 보고서

번호	함수 이름	실행 시간			
		컴파일러 옵션 사용 전	-O1 사용	-O2 사용	-O3 사용
1	CovarMatrix()	16	9	5	3
2	MM1_Gt_x_G()	262	53	40	26
3	MM2_Gt_x_CTx()	21	10	8	5
4	MI_inv()	55	25	20	14
5	MI_x_MM2()	6	3	2	1
총 시간		**360ms**	**99ms**	**75ms**	**49ms**

단계 4: NEON 벡터화

알테라 HPS에 탑재된 ARM A9 프로세서 코어는 각기 NEON 액셀러레이터 엔진을 포함하고 있다. NEON 액셀러레이터는 신호 처리를 가속화하기 위해 사용되는 범용 단일 명령 다중 데이터(SIMD) 엔진이다. NEON 액셀러레이터는 단일 명령을 사용하여 동일한 유형과 크기를 갖는 다중 데이터 요소에서 병렬로 동일한 연산을 수행한다(SIMD). 레지스터는 동일한 데이터 유형을 갖는 요소의 벡터로 간주된다. 데이터 유형은 부호/비부호 8비트, 16비트, 32비트 또는 단정밀도 부동소수점이 될 수 있다. 명령은 최대 4개 라인의 각각에서 동일한 연산을 수행한다(그림 16 참조).

그림 16. 명령의 연산을 보여주는 NEON 기술



NEON 엔진을 사용하려면 함수의 코드 루프가 벡터화되어야 한다. 다음으로 ARM 벡터화 컴파일러가 벡터화된 루프를 벡터 연산의 시퀀스로 효과적으로 변환하며, 이렇게 변환된 시퀀스는 NEON 엔진에서 병렬로 구현할 수 있다. DS-5 툴에 있는 자동 벡터화 기능은 컴파일러 플래그 '-ftree-vectorize'를 사용하면 활성화된다.

NEON을 이용하는 프로파일링을 위한 컴파일러 옵션

이 예에서 어떤 소스 코드도 변경할 필요가 없지만, 컴파일러 옵션과 코딩 연습을 사용해 컴파일러를 지원할 수 있다.

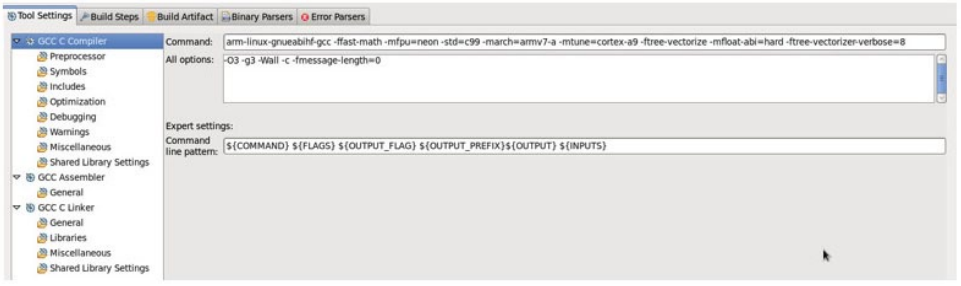
표 6. NEON 벡터화에 사용되는 컴파일러 플래그 (1/2)

파라미터	설명
-ffast-math	부동 소수점 축약 벡터화를 구현한다.
-mfpu=neon	이 플래그는 타겟에서 사용 가능한 부동 소수점 하드웨어(또는 하드웨어 에뮬레이션)를 지정한다.
-std=c99	C 컴파일 시 언어 표준을 결정한다.
-march=armv7-a	이 플래그는 타켓 ARM 아키텍처의 이름을 지정한다. GCC는 이 이름을 사용하여 어셈블리 코드를 생성할 때 어떠한 종류의 명령을 발생할지 결정한다.
-mtune=cortex-a9	이 옵션은 GCC가 코드의 성능을 튜닝할 타겟 ARM 프로세서의 이름을 지정한다. 일부 ARM 구현에서 이 옵션을 사용하면 더 향상된 성능을 얻을 수 있다.
-ftree-vectorize	이 플래그로 벡터화를 구현할 수 있다.

표 6. NEON **벡터화에 사용되는 컴파일러 플래그** (2/2)

파라미터	설명
-mfloat-abi=hard	어떤 부동 소수점 ABI를 사용할지 지정한다. 'hard'는 floatingpoint 명령을 생성할 수 있게 하고, FPU에 특정한 호출 규약을 사용한다.
-ftree-vectorizer-verbose=5	0(보고하는 정보 없음)에서부터 10(모든 정보 보고)까지 벡터화 보고서를 제어한다.
-O3	-O3 모든 최적화를 켠다.

그림 17. NEON **벡터화를 위한 컴파일러 옵션의 사용**



NEON 벡터화 가이드라인

정수 랩어라운드 연산의 최종 정밀도를 보존하면 벡터화를 진행할 수 있다. 예를 들어 32비트 오른쪽 시프트 연산자는 최종 저장된 값이 16비트 정수이면 16비트 모드에서 벡터화되지 않는다. 또한 모든 정수 연산을 벡터화할 수 있는 것은 아니다.

부동 소수점 수에서 동작하는 루프의 경우 ARM은 산술 연산자 더하기(+), 빼기(-), 곱하기(*), 나누기(/)에 대해 SIMD 명령어를 제공한다.

벡터화가 가능하려면 루프는 다음과 같은 상태가 되어야 한다.

- *셀 수 있어야 한다* - loop trip 카운트는 컴파일 시에 알려진 상태일 필요가 없지만, 런타임 시에는 루프에 진입할 때 알려진 상태이어야 한다(즉, trip 카운트는 변수일 수 있지만, 변수는 루프 기간 동안 일정한 상태를 유지해야 한다). 이것은 루프 종료는 데이터 종속적이어서는 안 된다는 것을 의미한다.
- *하나씩 들어가고 하나씩 나온다* - 이 말이 의미하는 바와 같이 루프는 셀 수 있어야 한다.
- *직선 코드를 포함해야 한다* - SIMD 명령은 맨 처음(original) 루프를 여러 번 반복하면서 데이터 요소에 대해 동일한 연산을 수행하므로, 서로 다른 반복이 서로 다른 제어 흐름을 갖는 것은 가능하지 않다. 즉, 분기해서는 안 된다. 스위치(switch) 문을 허용해서는 안 된다. 그러나 만약 문장을 허용하고, 그 문장을 마스크를 씌워준 할당으로 구현할 수 있다면, 이는 항상 다음과 같다. 계산은 모든 데이터 요소에 대해 수행되지만, 결과는 마스크가 참(true)으로 검증한 요소에 대해서만 저장된다.
- *중첩의 가장 안쪽 루프이어야 한다* - 유일한 예외는 맨 처음의 바깥쪽 루프가 unrolling, loop collapsing 또는 interchange 같은 일부 다른 앞선 최적화 단계의 결과로서 내부 루프로 바뀌

거나, 맨 처음의 맨 바깥쪽 루프가 loop materialization으로 인해 맨 안쪽 루프로 바뀌는 경우이다.

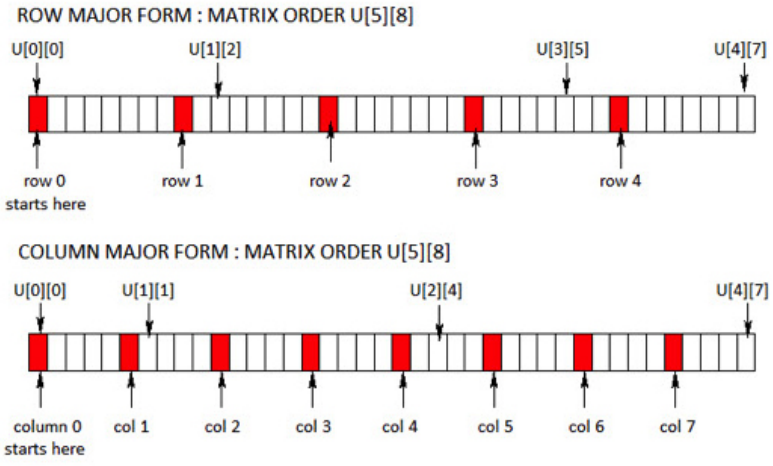
- *함수 호출이 없어야 한다* - 프린트(print) 문은 루프가 벡터화되지 못하게 한다. 벡터화 보고서 메시지는 일반적으로 비표준 루프는 벡터화 후보가 아니라는 것이다. 두 가지 주요 예외는 인트린직(intrinsic) 수학 함수와 인라인(inline)이 될 수 있는 함수의 경우이다.

다차원 배열을 일차원 배열로 변환하기

GCC는 다차원 배열의 벡터화를 지원하지 않는다. 따라서 코드는 모든 다차원 배열을 일차원 배열로 변환할 수 있게 수정되었다.

2D 행렬을 1D 배열로 변환하기 위해 행렬을 행 우선 방식 또는 열 우성 방식으로 배열할 수 있다. 아래의 코드 예제에서는 행 우선 방식이 사용됐지만, 이 두 방식 중 어느 하나가 다른 하나에 대해 특별한 장점을 갖는 것은 아니다. 행 우선 방식의 경우, 행을 취한 다음 모든 원소는 배열의 연속적인 메모리 위치에 배열되고, 해당 특정 행의 마지막 원소의 끝에서 행렬의 다음 행이 선택되며, 이러한 과정이 마지막 원소에 도달할 때까지 계속된다. 예제는 그림 18에 나와 있으며, 그림은 두 양식 중 하나로 원소가 어떻게 배열되는지를 보여준다.

그림 18. 2D 배열을 1D 배열로 변환하기



CovarMatrix 함수에 대한 코드 스니펫의 2D 버전은 다음과 같다.

```
void CovarMatrix_2D()//2D version of code for CovarMatrix
{
 MD=M+1;
 K = sizeof(Order)/sizeof(int);
 i=0;j=0;l=0;

 for (m=0;m<=M;m++)
 {
   for (k=0;k<K;k++)
   {
       i=0;
       for (j=(MD-m-1);j<( (sizeof(CoRx)/(2*sizeof(double)) - m) );j++)
       {
       G[i][l].r=
(CoRx[j].r*(pow(sqrtf((CoRx[j].r*CoRx[j].r)+(CoRx[j].i*CoRx[j].i)),Order[k])));


G[i++][l].i=(CoRx[j].i*(pow(sqrtf((CoRx[j].r*CoRx[j].r)+(CoRx[j].i*CoRx[j].i)),Order[k
]));            }
       l++;
   }
  }
}
```

CovarMatrix에 대한 코드 스니펫의 1D 버전은 다음과 같다. 이 코드는 다소 읽기 어렵다.

```
void CovarMatrix_1D()
{   int i=0,j=0;
   int Order[4] ={0,1,3,5};
   int K,MD,M=8;
   int m,k,a;
   double tempr,tempi;
   MD=M+1;K = sizeof(Order)/sizeof(int);
   for (m=0;m<=M;m++)
       {
          for (k=0;k<K;k++)
          {
                 a=MD-m-1;
                 for (j=0;j<2040;j++)
                 {   double val=1;
                    tempr=CoRx[j+a].r;tempi=CoRx[j+a].im;
                    double temp1;int temp2;
                    temp1= sqrtf(  (tempr*tempr)+(tempi*tempi)   );
                    temp2=Order[k];
                    if(temp2==0)
                       val=1;
                    else if(temp2==1)
                       val=temp1;
                    else if(temp2==3)
                       val=temp1*temp1*temp1;
                    else if(temp2==5)
                       val=temp1*temp1*temp1*temp1*temp1;
                    G[i].r   =  (  tempr *  val );
                    G[i++].im=  (  tempi *  val  );
                 }
          }
       }
}
```

‘restrict’ 키워드의 사용

'restrict’ 키워드는 정렬되지 않은 포인터 액세스를 다루는 데 사용된다. 'restrict’는 포인터 선언으로 사용할 수 있는 키워드이다. restrict 키워드는 프로그래머가 컴파일러에게 제공하는 의도를 선언한 것이다. 이것은 포인터의 수명 동안 오직 포인터 또는 포인터 + 1과 같은 포인터로부터 직접 유추된 값만, 포인터가 가리키는 객체에 액세스하는 데 사용된다는 것을 나타낸다. 이것은 컴파일러 최적화를 보조하는 포인터 앨리어싱을 제한한다. 만약 의도의 선언 다음에 뒤따르는 동작이 없고, 객체가 독립적인 포인터에 의해 액세스되면, 이것은 정의되지 않은 특성을 초래한다.

```
void MM1_Gt_x_G(struct DPD_alg * restrict buf)
```

다른 방법은 모든 포인터를 전역 포인터로 선언하는 것이다. 이 경우 'restrict’키워드는 신중하게 사용해야 한다.

그림 19. 필요한 옵션 사용 시 컴파일러에 의해 생성된 벡터화 보고서

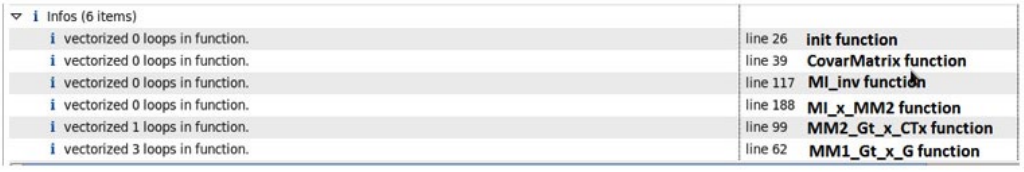


그림 19는 컴파일러에 의해서 생성된 보고서를 보여주며, 보고서는 벡터화되는 함수를 식별한다. 벡터화 보고서는 함수 단위로 생성되며, 각 함수의 시작 지점의 라인 번호를 표시한다. 각 함수는 여러 개의 루프를 가질 수 있으며, 이 보고서는 해당 함수에 얼마나 많은 루프가 벡터화화되었는지 알려준다.

함수 MM2_Gt_x_G() and MM1_Gt_x_CTx()의 루프들은 함수 호출 또는 "더블" 데이터 유형 변수를 포함하지 않으므로 성공적으로 벡터화된다.

일부 함수의 루프들은 벡터화되지 않는다.

- *CovarMatrix()* – sqrt() 함수에 대한 함수 호출을 포함하므로 벡터화할 수 없다.
- *MI_inv()* – 이 함수의 변수는 "더블" 데이터 유형으로 정의된다. 컴파일러는 "더블" 데이터 유형의 벡터화를 지원하지 않는다.
- *MI_x_MM2()* – 이 함수는 역행렬의 결과를 사용한다. 따라서 더블 데이터 유형이므로 벡터화할 수 없다. 다음에 주의한다. 역행렬 함수 결과에 대한 형변환은 정밀도 손실을 초래하므로 수행되지 않는다. 형변환을 수행하는 경우 해당 함수의 루프는 벡터화할 수 있지만, 정밀도는 손상된다.

그림 20. '-ftree-vectorize' 및 '-O3' 컴파일러 플래그를 사용한 ARM 프로파일 보고서

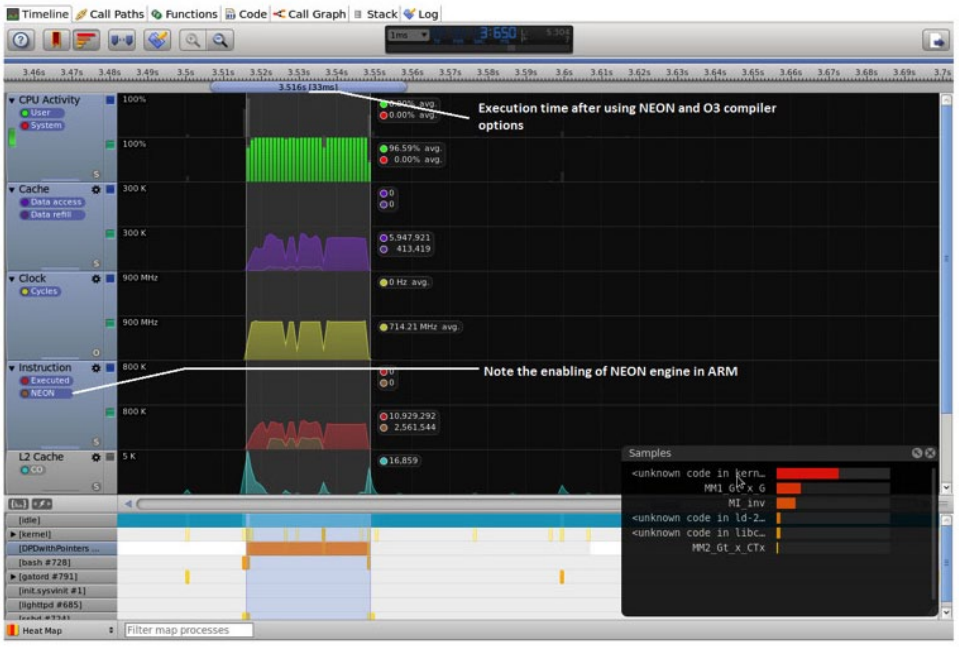


그림 21. '-ftree-vectorize' 및 '-O3' 컴파일러 플래그를 사용한 ARM 프로파일 보고서

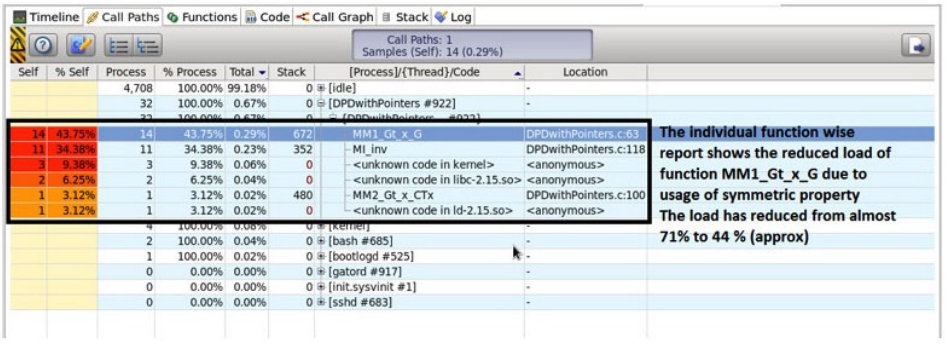


그림 22. '-ftree-vectorize' 및 '-O3' 컴파일러 플래그를 사용한 C 코드 및 디스어셈블리

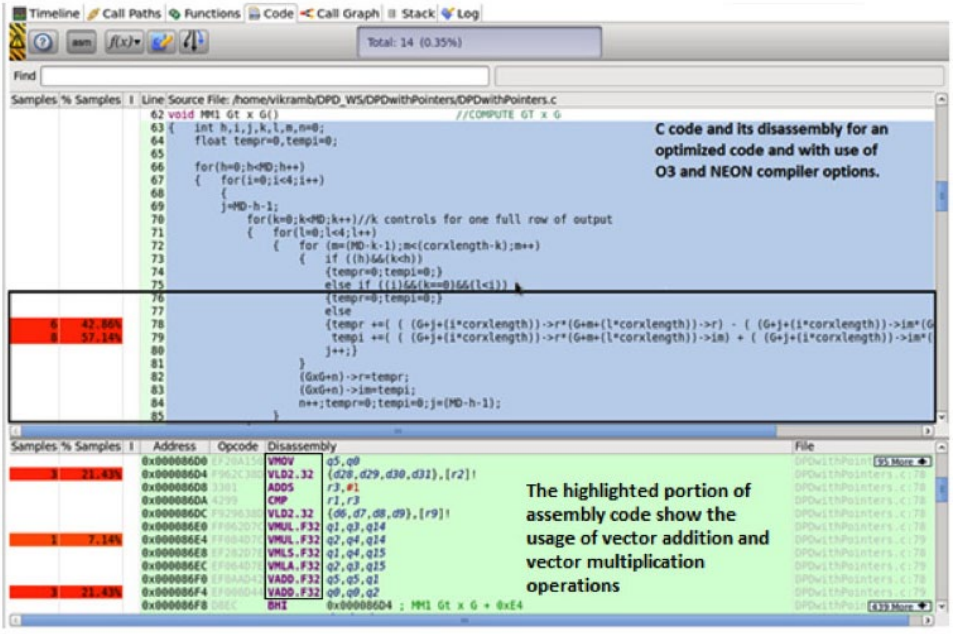


그림 22의 코드 뷰는 이전 코드 뷰와 비교할 때 샘플 비율과 디스어셈블리 코드에서 많은 차이를 보여준다. NEON 벡터화에서 일부 루프는 더 효율적으로 구현되었으며, 따라서 해당 라인에서 최대 샘플 비율이 감소되었다. 어셈블러 명령어 'VMUL' 및 'VADD'는 벡터 연산이 수행되는 것을 명백히 보여준다.

ARM 프로파일 보고서 – NEON 플래그

표 7. 'free-vectorize 및 ' - O3' 컴파일러 플래그를 사용한 ARM 프로파일 함수 단위로 생성된 보고서

번호	함수 이름	실행 시간(ms)	
		NEON 벡터화 이전	NEON 벡터화 이후
1	CovarMatrix()	3	1.8
2	MM1_Gt_x_G()	25	18
3	MM2_Gt_x_CTx()	6	4
4	MI_inv()	13	9
5	MI_x_MM2()	1	〈1
총 시간		48ms	33ms

단계 5: 멀티코어 최적화

공유 메모리 멀티코어 프로세서 아키텍처에서 스레드는 병렬처리를 구현하는 데 사용할 수 있다. UNIX 시스템의 경우 표준화된 C 언어 스레드 프로그래밍 인터페이스는 IEEE POSIX 1003.1c 표준에 의해 지정되고 있다. 이 표준에 부합하는 구현은 POSIX 스레드 또는 Pthread라고 한다. 스레드는 운영 시스템에 의해 실행되도록 스케줄링할 수 있는 독립적인 명령의 흐름으로 정의할 수 있다.

프로그램이 서로에 대해 독립적인 여러 개의 프로시저를 포함하고 있는 경우, pthread를 사용하여 전체 지연을 줄일 수 있다. 스레드는 랜덤 인스턴스에서 초기화되며, 운영 시스템에서 정의된 다양한 스케줄링 메커니즘으로 실행된다. 따라서 견고한 프로그램은 특정한 순서나 특정 코어에서 실행하는 스레드에 의존해서는 안 된다. pthread 사용을 보여주는 스니펫은 다음과 같다.

```
#include <pthread.h>
#include <stdio.h>

pthread_t thread1, thread2;
int main()
{
    int rc;
    rc = pthread_create(&thread1, NULL, dpd_algorithm,(void*)&dpd1);  //Antenna1
    if (rc) {
        exit(-1);
    }
    rc = pthread_create(&thread2, NULL, dpd_algorithm,(void*)&dpd2);  // Antenna2
    if (rc) {
        exit(-1);
    }
    pthread_exit(NULL);
    return 0;
}
```

pthread을 사용하는 프로파일링을 위한 컴파일러 옵션

pthread를 사용할 수 있으려면 아래의 다음과 같은 컴파일러 옵션을 사용한다.

- -pthread – 컴파일러에게 운영 시스템에 의해 정의된 스케줄링 메커니즘에 따라 pthread를 사용하여 다양한 프로시저를 실행하도록 내리는 명령

스레드 프로그램 설계를 위한 가이드라인

pthread는 병렬 프로그래밍에 특히 적합하며, 병렬 프로그램 설계를 위한 일부 고려사항은 다음과 같다.

- 문제 분할
- 부하 분산
- 통신
- 데이터 종속성
- 동기화 및 경합 조건
- 메모리 문제

- I/O 문제
- 프로그램 복잡성
- 프로그래머의 노력/비용/시간

일반적으로, 프로그램이 pthread를 이용하기 위해서는 프로그램은 동시에 실행할 수 있는 개별적이고 독립적인 작업으로 조직할 수 있어야 한다. 예를 들어 routine1과 routine2를 교환하고, 인터리빙 및/또는 오버래핑할 수 있다면, 이들 루틴은 스레딩의 후보가 될 수 있다. 다음의 특성을 갖는 프로그램은 pthread에 적합할 수 있다.

- 다중 작업에 의해 동시적으로 실행할 수 있는 작업 또는 연산할 수 있는 데이터
- 잠재적으로 긴 I/O 대기를 위한 블록
- 일부 지점에서는 많은 CPU 사이클을 사용하지만 다른 지점에서는 그렇지 않은 경우
- 비동기 이벤트에 응답해야 한다.
- 일부 작업은 다른 작업보다 더 중요하다(우선순위 인터럽트).

공유 메모리 모델

공유 메모리 모델에서 모든 스레드는 동일한 전역, 공유 메모리에 대한 액세스를 갖는다. 스레드는 또한 자체적인 개인 데이터를 갖는다. 프로그래머는 전역적으로 공유되는 데이터에 대해, 이를 보호하면서, 액세스를 동기화하는 작업을 담당한다.

스레드 안전성

스레드 안정성은 애플리케이션이 공유된 데이터를 "독식하거나" "경합" 조건을 발생시키지 않으면서 여러 스레드를 동시에 실행할 수 있는 것을 말한다. 예를 들어 애플리케이션이 여러 개의 스레드를 생성하고 이들 스레드가 각각 동일한 라이브러리 루틴을 호출한다고 한다면, 이러한 라이브러리 루틴은 메모리에서 전역 구조나 위치에 액세스하고 이를 수정한다. 각 스레드가 이러한 루틴을 호출할 때 스레드는 동시에 이러한 전역 구조와 메모리 위치를 수정하려고 할 수 있다. 만약 루틴이 일정한 종류의 동기화 구조를 사용해 데이터 훼손을 방지하지 않는다면, 스레드 안정성이 없다고 할 수 있다. 안정성이 의심스러운 경우, 달리 입증될 때까지 스레드 안정성이 없다고 가정한다. 이것은 불확실한 루틴에 대한 호출을 "직렬화함으로써" 수행할 수 있다.

pthread를 사용한 ARM 프로파일링

pthread의 개념을 이용하면 ARM과 NEON의 코어를 동시에 사용해 2개의 안테나 캐리어에 대해 DPD 알고리즘의 두 개 프로세스를 실행할 수 있다. 이것은 컴파일러 옵션 – pthread를 사용하고, 또한 코드를 변경해 DPD 알고리즘에 대해, 각각 개별적인 안테나 캐리어로부터 발생하는 데이터를 사용하는 2개의 다른 스레드를 생성하도록 함으로써 수행된다. 이 구현에 대한 결과와 관찰은 그림 23에 제공된다.

그림 23. P-thread를 사용하여 멀티 스레딩을 구현하는 경우 스트림라인 툴의 타임라인 뷰

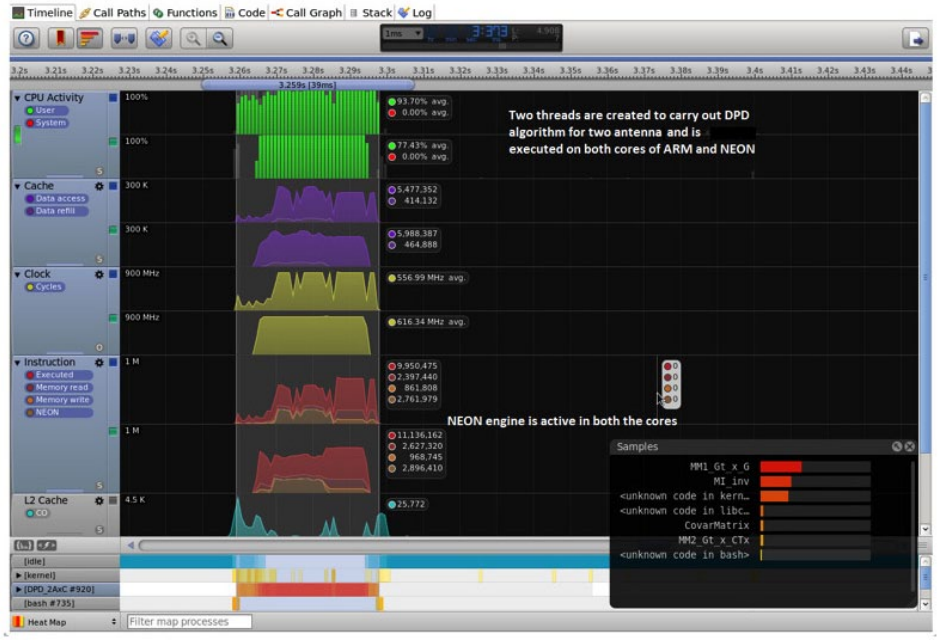
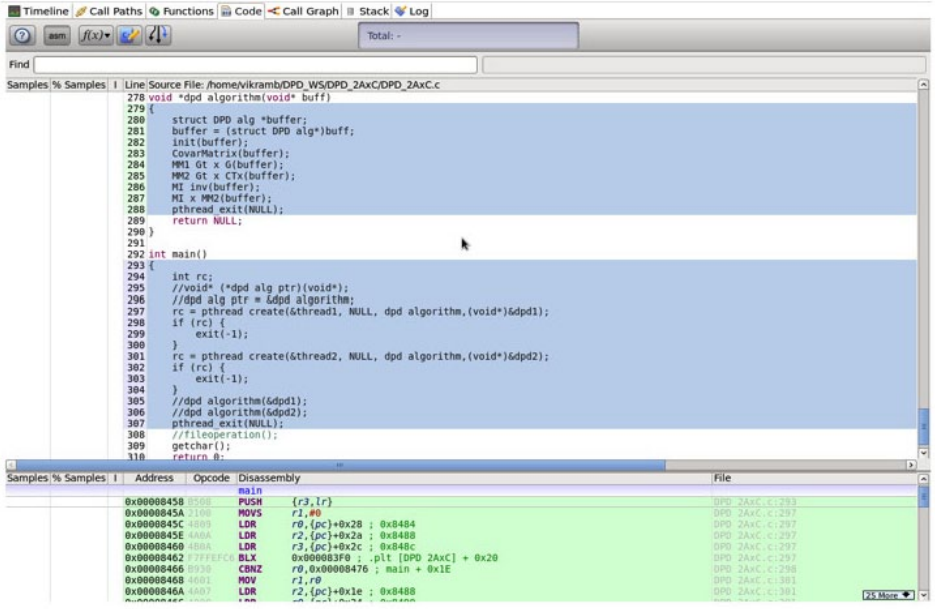


그림 24. pthread를 통합하기 위해 작성된 코드 변경



ARM 프로파일 보고서 – pthread

멀티스레드 방법으로 전환하면 코드를 두 코어에 모두 실행할 수 있다. 지연은 거의 절반으로 감소된다.

표 8. pthread 사용 후 ARM 프로파일 보고서(Neon 및 03에서)

실행 시간	
2개 안테나에 대해 pthread를 사용하지 않은 경우. 직렬 처리	2개 안테나에 대해 pthread를 사용한 경우. SMP
2x33= 66ms	39ms

단계 6. FPGA 가속화

표 7은 MM1_Gtx_G() 함수가 처리 시간의 거의 절반(18/33)을 소모한다는 것을 보여준다. FPGA에서는 고도로 병렬적인 복소수 곱셈을 매우 효율적으로 수행할 수 있다. 가속화는 FPGA 리소스와 지연 간의 간단한 트레이드오프가 된다. 데이터 전송에서 약간의 오버헤드는 고려해 넣어야 하지만, 데이터 크기는 상대적으로 작으며, 알테라 HPS-FPGA 패브릭 인터페이스는 넓은 폭(128비트)과 빠른 속도를 제공한다. Arria®10 및 Stratix® 10 디지털 신호 처리(DSP) 블록은 단정밀도 연산을 위한 하드 부동 소수점을 포함하고 있다.

그림 25. FPGA 가속화

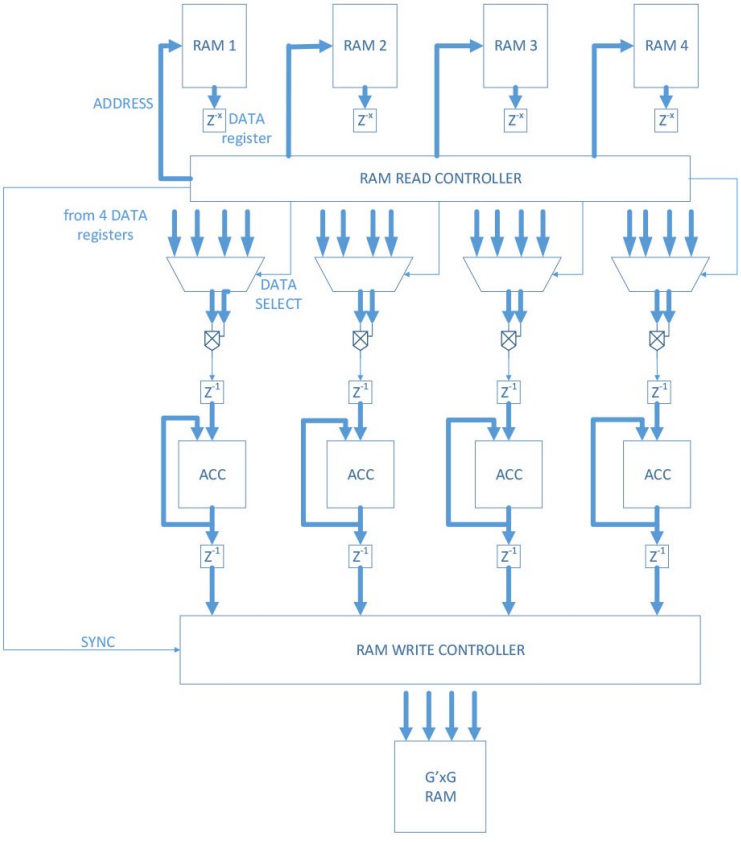


표 9. FPGA 가속화에서 지연

FPGA 리소스`	가속화 없음	4레인	16레인
복소수 곱셈기의 수 복소수 덧셈기의 수	0 0	4 4	16 16
가속화 지연	0	1.5ms	0.5ms
총 지연(2개 안테나에 대한)	39ms	22.5ms	21.4ms

보고서 요약

이 글의 본문에서 보고하는 지연은 Cyclone® V 디바이스에서 수행된 것이다. Arria V 및 Arria10은 동일한 HPS 시스템을 사용하고 있지만, 보다 높은 성능 클록 속도를 제공한다. Stratix 10은 쿼드 A-53 코어를 내장한 새로운 HPS를 탑재해 더욱 높은 성능을 제공한다.

표 10. 상대적 프로세서 속도

디바이스	코어	코어 클록 속도 (MHz)	상대 속도(추정)	비고
Cyclone V SoC	Dual A9	925	1.00	이 보고서의 본문
Arria V SoC	Dual A9	1050	1.14	보다 빠른 실리콘 공정
Arria 10 SoC	Dual A9	1500	1.62	20nm로 클록 증가
Stratix 10 SoC	Quad A53	1500	2.50	14nm, 쿼드 코어, 마이크로아키텍처 향상으로 클록 증가

표 12는 다양한 최적화 단계에서 얻은 모든 결과를 요약한 것이다. 다른 실리콘 디바이스에서의 성능은 HPS 시스템의 상대적 속도를 기초로 추정되었다.

표 11. 다양한 컴파일러 옵션을 사용하여 수행된 모든 구현 요약

디바이스	사용된 코어	최적화	안테나당 시간(ms)
Cyclone V SoC	1	OP-0 (바닐라 구현)	433
	1	OPT-0 (알고리즘 최적화)	360
	1	OPT-1	99
	1	OPT-2	75
	1	OPT-3	48
	1	OPT-3 + Neon 벡터화	33
	1	OPT-3 + Neon + pthread (2개 안테나에대한)	39/2
	2	OPT-3 + Neon + pthread (2개 안테나에대한) + FPGA 가속화	21/2
Arria V SoC	2	OPT-3 + Neon + pthread (2개 안테나에대한) + FPGA 가속화	19/2
Arria 10 SoC	2	OPT-3 + Neon + pthread (2개 안테나에대한) + FPGA 가속화	13/2
Stratix 10 SoC	2	OPT-3 + Neon + pthread (2개 안테나에대한) + FPGA 가속화	9/2

개발 환경

이 분석에 사용된 개발 환경은 다음과 같다.

- Cyclone V SoC:
  - Linux Version: 3.9
  - GCC Version: 4.7.3

표 12. 소프트웨어 개발 환경

툴 이름	운영 시스템	툴 버전	빌드
Altera SoC Embedded Design Suite Subscription Edition Software	Linux	13.1	162
DS-5	Linux	5.18	5180018
MATLAB	Linux	R2013a	8.1.0.604
Visual Studio Express	Windows 8.1	2013	

결론

알테라 SoC는 ARM 프로세서를 탑재한 HPS를 통합하고 있으며, ARM DS-5 알테라 에디션 툴킷은 스트림라인 툴을 포함하고 있다. 이 툴은 엔지니어와 알고리즘 설계자들이 코드 지연을 함수별, 라인별로 손쉽게 시각화할 수 있게 한다. 이러한 신속한 시각화를 통해 알고리즘 최적화와 코드 최적화를 효과적으로 수행할 수 있다. 알테라 SoC는 하드웨어 FPGA 로직과 소프트웨어 간에 계산을 분할하는 다양하고 풍부한 선택을 제공한다. 이 글은 스트림라인 툴을 사용하여 엔지니어가 어떻게 효과적으로 코드를 설계해 최적화를 구현하고, 적절한 분배를 통해 지연을 줄일 수 있는지를 보여준다. 또한 이 글의 예제들은 어떻게 특정 DPD 적응 알고리즘에서 프로세싱 시간을 바닐라 C 코드 구현 시 400ms 이상에서 10ms 미만 지연까지 줄일 수 있는지 보여준다.

추가 정보

- SoC 개요
  www.altera.com/socfpga
- Cyclone V SoC 하드 프로세서 시스템:
  www.altera.com/cvsoc
- ARM DS-5 알테라 에디션 툴킷
  www.altera.com/DS5AE
- Altera DSPBA
  www.altera.co.uk/technology/dsp/advanced-blockset/dsp-advancedblockset.html
- Quartus II 소프트웨어
  www.altera.com/products/software/quartus-ii/web-edition/qts-we-index.html

- 소프트웨어 디버그
  www.altera.com/literature/wp/wp-01198-fpga-software-debug-soc.pdf
- 스트림라인
  ds.arm.com/ds-5/optimize/streamline-features
- DS-5
  www.altera.com/devices/processor/arm/cortex-a9/software/proc-armdevelopment-suite-5.html
- ARM 코드 최적화
  infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html
- ARM 컴파일러 설정
  infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0472c/BABDGGBI.html
- ARM Cortex-A53
  www.arm.com/products/processors/cortex-a50/cortex-a53-processor.php
- EDS
  www.altera.co.uk/devices/processor/arm/cortex-a9/software/proc-socembedded-design-suite.html
- Neon C 팁
  www.altera.co.uk/devices/processor/arm/cortex-a9/software/proc-socembedded-design-suite.html

도움주신 분

- Richard Maiden, Senior Manager, Wireless Systems Solutions Engineering, Altera

문서 개정 이력

표 13은 이 문서에 대한 개정 이력을 보여준다.

표 13. 문서 개정 이력

날짜	버전	변경
2015년 2월	1.0	최초 발행