Multicore Computing, fall 2016
Project

HeeHoon Kim, 2013–11395

December 20, 2016

1 개요

본 프로젝트에서는 CNN 모델 중 하나인 VGG16으로 이미지를 분류하는 프로그램을 OpenCL을 이용하여 구현하였습니다, 사용한 CPU는 Intel Xeon E5–2650 (Sandy Bridge-EP, 8-core 2.00GHz), GPU는 AMD Radeon HD 7970입니다.

2 표기

$ceil(x, y)$는 x를 가장 가까운 y의 배수로 올림한 것입니다.

3 구현A:2 CPU

3.1 Convolution

VGG16 모델에서의 Convolution은 input 크기 $B(batch)*CI(inputchannel)*N(height)*N(width)$, kernel 크기 $CO(outputchannel)*CI*3(kernelwidth)*3(kernelheight)$, output 크기 $B*CO*N*N$에 대해 이루어집니다. 뼈대코드대로 6중 포문으로 구현하면 연산량은 이미지 하나당 $N*N*CO*CI*3*3$으로 정상이지만 CPU의 최대 성능을 얻을 수 없습니다. 그 이유는 다음 두 가지를 들 수 있겠습니다.

- 메모리 접근 방식이 좋지 않아 캐시를 통한 데이터 reuse가 잘 일어나지 않는다.
- SIMD 명령을 활용하기 어렵다.

그래서 보통 사용하는 방법이 Convolution lowering[1]입니다. 이 방법은 kernel을 $(CO, CI*3*3)$, input을 $(CI*3*3, N*N)$ 크기의 행렬로 생각하여 convolution을 행렬곱으로 대체합니다. 장점은 위의 문제점들을 해결하는 고도로 최적화된 행렬곱 커널을 사용할 수 있다는 점이고, 단점은 kernel과 input을 행렬곱에 알맞은 형태로 한번 변환해 주어야 한다는 점입니다. convWeightPad 커널함수는 kernel을 $(CO, ceil(CI*3*3, 2))$ 크기로 변환합니다. convInputPad 커널함수는 input을 $(ceil(CI*3*3, 2), ceil(N*N, 8))$ 크기로 변환합니다. 이렇게 하여 앞선 과제에서 작성했던 8-SPFP FMA SIMD를 사용하는 행렬곱 구현을 그대로 사용할 수 있었습니다. convOutputUnpad 커널함수는 결과로 나온 $(CO, ceil(N*N, 8))$ 행렬을 $(CO, N*N)$으로 되돌려 줍니다.

3.2 Pool, FC, softmax

Pooling, softmax 레이어 커널은 특이사항 없이 단순하게 구현하였습니다. FC 레이어의 경우 (CO, CI) 행렬과 $(CI, 1)$ 행렬의 곱으로 생각하여 convolution과 유사하게 구현하였습니다.

3.3 Host code

weight, bias, layer를 위해 malloc으로 할당된 모든 메모리를 clCreateBuffer를 사용하여 device 메모리에 대신 할당하였습니다. weight, bias의 경우 한 번만 로드하여 재사용합니다. kernel 실행 오버헤드를 줄이기 위해서, 커널들이 batch 단위로 이미지를 처리할 수 있게 구현하고 여러장의 이미지를 한 번에 연산하도록 하였습니다.

4 구현B:1 GPU

4.1 Convolution

GPU는 CPU와는 특성이 많이 다르기 때문에 위의 구현을 그대로 사용하면 성능이 좋지 않습니다. 그래서 GPU convolution kernel에는 다음과 같은 최적화들을 하였습니다.

4.1.1 느린 memory bandwidth 숨기기

GPU는 연산 throughput 대 메모리 throughput 비가 CPU보다 높기 때문에 L1 및 L2 cache, local memory, register를 이용한 데이터 재사용으로 메모리 bandwidth를 줄이는데 집중해야 합니다. 그렇지 않으면 연산이 아무리 빨라도 memory-bound가 되어 성능이 높지 않습니다. 행렬곱의 경우 bandwidth를 줄이기 위해 행렬을 부분행렬로 나누어 한 계층 위의 메모리에 저장한 후 계산하는 방식을 사용합니다. [3]의 계산방식을 참고하면, AMD Radeon HD 7970의 register와 global memory의 throughput은 약 86배 차이가 나고 local memory와는 약 6배 차이가 납니다. 그러므로 최소한 global memory 수준에서 $(128, 128)$, local memory 수준에서 $(8, 8)$ 크기로 행렬을 나누는 blocking을 해야 memory bandwidth에 제약받지 않을 수 있습니다. 그런데 VGG16에서의 행렬 크기들이 크지 않아서, 크기가 128보다 작은 행렬을 128이 되도록 padding하면 연산량이 너무 많이 늘어나 오히려 느려질 수 있습니다. 그래서 각각 $(64, 64)$, $(4, 4)$ 크기의 blocking을 하였습니다.

4.1.2 메모리 레이아웃 변경

GPU는 연속한 work-item이 연속한 메모리를 접근할 때 가장 좋습니다. conv_preA 커널함수는 kernel을 transposition 후, $(ceil(CI * 3 * 3, 16), CO)$ 크기로 변환함과 동시에 CBL (Column block layout)[2]으로 저장합니다. 이를 통해 행렬곱 커널의 연속한 work-item들이 연속한 메모리를 접근하도록 합니다. input 쪽은 explicit 하게 $(ceil(CI * 3 * 3, 16), ceil(N * N, 64))$ 행렬로 변환하는 경우 메모리 양이 약 9배 커지기 때문에, bandwidth에서 오히려 손해를 볼 수 있습니다. 또한 GPU 메모리를 많이 차지하여 최대 batch size가 작아지게 됩니다. 그래서 커널을 이용해 따로 변환하는 대신, 행렬곱 커널에서 복잡한 인덱싱을 거쳐 직접 local memory로 로드합니다.

4.1.3 더블 버퍼링

현재 한 work-item의 루프 안에서 하는 일은 다음과 같습니다.

1. kernel에서 local memory로 float 4개 로드
2. input에서 local memory로 float 4개 로드
3. local memory에서 $(4, 16)$, $(16, 4)$ 크기의 행렬을 가져와서 행렬곱

앞 두개는 memory-intensive하고 뒤는 compute-intensive합니다. 이 두개를 최대한 오버랩 시켜주어야 memory latency를 숨길 수 있습니다. 그래서 다음과 같이 수정하였습니다.

1. kernel에서 local memory로 float 1개 로드
2. input에서 local memory로 float 1개 로드
3. local memory에서 $(4, 4)$, $(4, 4)$ 크기의 행렬을 가져와서 행렬곱
4. 4번 반복

4.1.4 Instruction mixing

위의 행렬곱 부분에서, 'local memory에서 가져오기'와 '행렬곱'은 register 수준에서 read-after-write 의존성이 있습니다. 이를 순서를 잘 섞어서 최대한 의존성을 줄였습니다. 커널코드의 GEMM44 매크로를 보면 확인할 수 있습니다.

4.2 Pooling

local memory로 행렬을 로드 한 뒤, work-item 4개 중 1개가 최대값을 구해 저장합니다.

4.3 Fully-connected layer

work-item 256개로 구성된 work-group 하나가 출력 하나를 담당합니다. 즉, work-group CO개가 CI차원 벡터 내적을 수행합니다. 각 work-item은 CI/256차원 벡터 내적을 수행한 뒤 로컬 메모리에 기록하고, work-group 단위로 sum-reduce하여 저장합니다.

4.4 Host code

3GB의 GPU 메모리를 잘 활용하기 위해 버퍼들을 모두 재활용하도록 수정하였습니다. 각 레이어별로 버퍼를 할당하는 대신 충분히 큰 크기의 버퍼 2개를 할당하여 input, output으로 번갈아 사용하였습니다. 또한 conv_preA, conv_postC에서 사용하는 버퍼도 한개만 할당하였습니다.

5 구현C:4 GPU

구현B와 같은 코드를 pthread를 통해 4개의 쓰레드에서 실행했습니다. data parallelism으로 이미지를 1/4로 나누어 분배했습니다.

6 구현D:16 GPU with MPI

구현B와 같은 코드를 MPI를 통해 16개의 프로세스에서 실행했습니다. data parallelism으로 이미지를 1/16로 나누어 분배했습니다. 연산이 끝난후 MPI_Bcast로 결과를 모아주었습니다.

7 구현E:16 GPU with SnuCL

구현C와 같은 코드를 NUM_GPU 상수만 16으로 고쳐주었습니다. clReleaseXXX 함수들이 의문의 세그멘테이션 폴트를 일으켜서 주석처리 하였습니다.

8 결과

Table 1: 이미지 개수 별 수행시간 (s)

구현	0	16	64	256	512	1024	이미지 당 수행시간 (ms)
CPU	0.469	7.002	26.592	104.436	208.618	415.258	405.067
1GPU	1.345	1.788	2.850	7.200	12.961	24.588	22.698
4GPU	2.474	2.594	2.879	4.016	5.393	8.362	5.750
16GPU with MPI	1.835	2.144	2.684	2.790	2.965	3.881	1.998
16GPU with SnuCL	6.326	7.157	6.917	7.233	7.236	7.993	x

이미지 0장인 경우는, 이미지 처리를 제외한 오버헤드를 확인하기 위해 측정했습니다. 이미지 당 수행시간은 오버헤드를 제외한 수행시간을 이미지 개수로 나눈 값입니다. GPU 사용개수에 거의 반비례인 것을 확인할 수 있습니다.

오버헤드의 경우 분석 결과 다음과 같이 이루어져 있습니다.

- clGetPlatformIDs: 약 0.8초
- clBuildProgram: 약 0.3초
- MPI sync: 약 0.5초

특히 clBuildProgram의 경우 thread를 쓰더라도 순차실행이 되는것으로 보아 내부적으로 글로벌 락이 존재하는 것으로 추측됩니다. 결과적으로 4GPU인 경우 1GPU인 경우보다 clBuildProgram을 3번 호출한만큼의 시간이 더 걸립니다. MPI의 경우 쓰레드 대신 프로세스 단위로 나누어줬기 때문에 오히려 오버헤드가 적습니다. SnuCL은 clCreateBuffer가 매우 오래 걸리고, 실행 시간도 편차가 심했습니다.

9 실행방법

image_list.txt 파일을 작성한 후, make run을 실행합니다.

References

[1] K. Chellapilla, S. Puri, and P. Simard. High Performance Convolutional Neural Networks for Document Processing. In G. Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), Oct. 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[2] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. Implementing a code generator for fast matrix multiplication in opencl on the gpu. In *2012 IEEE 6th International Symposium on Embedded Multicore SoCs*, pages 198–204, Sept 2012.

[3] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.