

CUDA C 모범 관례 지침서 (best practices guide)

원문 : <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide>
http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

v7.5 - 2015 년 9 월 1 일에 마지막으로 바뀜
2016 년 5 월 19 일에 한국어로 옮겨짐

목차

머리말

1. 응용 프로그램 평가하기

2. 이기종 컴퓨팅

2.1. 호스트와 디바이스 사이 차이

2.2. 무엇을 쿠다 가능 디바이스에서 실행하나요?

3. 응용 프로그램 프로파일링

3.1. 프로파일

3.1.1. 프로파일 만들기

3.1.2. 핫스팟 찾기

3.1.3. 스케일링 이해하기

3.1.3.1. 강한 스케일링과 암달의 법칙

3.1.3.2. 약한 스케일링과 구스타프슨의 법칙

3.1.3.3. 강한 그리고 약한 스케일링 적용하기

4. 응용 프로그램 병렬화하기

5. 시작하기

5.1. 병렬 라이브러리

5.2. 컴파일러 병렬화하기

5.3. 병렬성을 드러내기 위한 코딩

6. 바른 정답 얻기

6.1. 검증

6.1.1. 참조 비교

6.2. 디버깅

6.3. 수치 정확도와 정밀도

6.3.1. 단정밀 대 배정밀

6.3.2. 부동 소수점 수학은 결합적이지 않다

6.3.3. 더블 형으로 승격 그리고 플롯 형으로 강등

6.3.4. IEEE 754 준수

6.3.5. x86 80-bit 계산

7. 쿠다 응용 프로그램 최적화하기

8. 성능 계량

8.1. 타이밍

8.1.1. CPU 타이머 사용하기

[8.1.2. 쿠다 GPU 타이머 사용하기](#)

[8.2. 대역폭](#)

[8.2.1. 이론적 대역폭 계산](#)

[8.2.2. 실효 대역폭 계산](#)

[8.2.3. 비주얼 프로파일러로 보고된 처리량](#)

[9. 메모리 최적화](#)

[9.1. 호스트와 디바이스 사이 데이터 전송](#)

[9.1.1. 핀으로 고정된\(Pinned\) 메모리](#)

[9.1.2. 계산과 함께 비동기 그리고 중첩된 전송](#)

[9.1.3. 제로 복사](#)

[9.1.4. 통합된 가상 번지 지정](#)

[9.2. 디바이스 메모리 공간](#)

[9.2.1. 글로벌 메모리로 응집된 접근](#)

[9.2.1.1. 단순 접근 패턴](#)

[9.2.1.2. 순차적이나 정렬되지 않은 접근 패턴](#)

[9.2.1.3. 정렬되지 않은 접근의 효과](#)

[9.2.1.4. 건너뛰며 접근](#)

[9.2.2. 공유 메모리](#)

[9.2.2.1. 공유 메모리와 메모리 뱅크](#)

[9.2.2.2. 행렬 곱셈에서 공유 메모리 \(\$C=AB\$ \)](#)

[9.2.2.3. 행렬 곱셈에서 공유 메모리 \(\$C=AA^T\$ \)](#)

[9.2.3. 로컬 메모리](#)

[9.2.4. 텍스처 메모리](#)

[9.2.4.1. 추가적 텍스처 능력](#)

[9.2.5. 상수 메모리](#)

[9.2.6. 레지스터](#)

[9.2.6.1. 레지스터 압력](#)

[9.3. 할당](#)

[10. 실행 구성 최적화](#)

[10.1. 점유](#)

[10.1.1. 점유 계산하기](#)

[10.2. 동시 커널 실행](#)

[10.3. 여러 개의 문맥](#)

[10.4. 레지스터 의존성 감추기](#)

[10.5. 스레드와 블록 휴리스틱스](#)

[10.6. 공유 메모리의 효과](#)

[11. 명령어 최적화](#)

[11.1. 산술 명령어들](#)

[11.1.1. 나누기 나머지 연산들](#)

[11.1.2. 제곱근의 역수](#)

[11.1.3. 다른 산술 명령어들](#)

[11.1.4. 작은 분수 인자들을 가진 누승법](#)

[11.1.5. 수학 라이브러리들](#)

[11.1.6. 정밀도 관련 컴파일러 플래그들](#)

[11.2. 메모리 명령어들](#)

12. 제어 흐름

12.1. 분기와 발산

12.2. 분기 예측

12.3. 루프 카운터 Signed 대 Unsigned

12.4. 루프에서 발산적 스레드 동기화하기

13. 쿠다 응용 프로그램 배포

14. 프로그래밍 환경 이해하기

14.1. 쿠다 계산 능력

14.2. 추가적 하드웨어 데이터

14.3. 쿠다 런타임 그리고 드라이버 응용 프로그램 인터페이스 버전

14.4. 어느 계산 능력을 타겟으로 할 것인가

14.5. 쿠다 런타임

15. 배포를 위한 준비

15.1. 쿠다 사용 가능성 시험

15.2. 오류 처리

15.3. 최대 호환성을 위한 건설

15.4. 쿠다 런타임과 라이브러리 분배하기

15.4.1. 쿠다 도구모음 라이브러리 재배포

15.4.1.1. 어떤 파일들을 재배포 하나

15.4.1.2. 재배포된 쿠다 라이브러리들을 어디에 설치하나

16. 배포 기반시설 도구들

16.1. Nvidia-SMI

16.1.1. 조회할 수 있는 상태

16.1.2. 고칠 수 있는 상태

16.2. NVML

16.3. 클러스터 관리 도구들

16.4. 컴파일러 JIT 캐시 관리 도구들

16.5. CUDA_VISIBLE_DEVICES

A. 권고 그리고 모범 관례

A.1. 전체적 성능 최적화 전략들

B. nvcc Compiler Switches

B.1. nvcc

공지

Trademarks

Copyright

머리말

이 문서는 무엇인가?

이 모범 관례 안내서는 개발자들이 엔비디아 쿠다 지피유(NVIDIA® CUDA® GPUs)에서 가장 좋은 성능을 얻도록 돕기 위한 안내서입니다. 이 안내서는 오래 쓰여 관습처럼 굳어진 병렬화와 최적화 기법들을 제시합니다. 그리고 이 안내서는 쿠다를 쓸 수 있는 GPU 구조에서 프로그래밍을 크게 단순화시킬 수 있는 코딩 은유(metaphor)와 관용구들을 설명합니다.

이 안내서의 내용은 참고서로 쓰일 수 있습니다. 하지만 다양한 프로그래밍과 구성(configuration) 주제들이 여러 다른 맥락에서 다시 다뤄집니다. 따라서, 처음 읽으시는 분은 이 안내서를 시작부터 차례대로 읽으시기를 추천합니다. 이 접근법은 여러분이 효과적인 프로그래밍 관례를 이해하는 데 큰 도움을 주고, 이 안내서를 나중에 참고자료로 더 잘 사용할 수 있게 할 것입니다.

누가 이 안내서를 읽어야 하나?

이 안내서에는 모두 C 언어가 사용되므로, 이 글을 읽으시는 분은 C 코드를 편하게 읽을 수 있어야 합니다.

이 안내서는 몇몇 다른 문서의 참조를 요구합니다. 우리는 그 문서들을 가지고 있어야 합니다. 모든 문서는 쿠다 웹사이트(<http://developer.nvidia.com/cuda-downloads>)에서 무료로 얻으실 수 있습니다. 그 중, 다음 문서들은 특히 중요합니다:

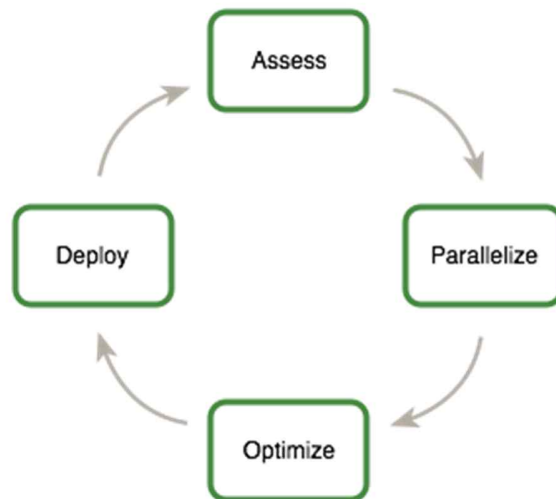
- 쿠다 설치 안내서
- 쿠다 C 프로그래밍 안내서
- 쿠다 도구모음(toolkit) 참고서

이 안내서의 최적화 절은 우리가 이미 쿠다 도구 모음을 성공적으로 내려 받아 설치했다고 가정합니다 (만약 아니면, 자신의 플랫폼에 맞는 적절한 쿠다 설치 안내서를 참고해 주십시오). 그리고 이 안내서는 우리가 기본적인 쿠다 C 프로그래밍 언어와 환경에 친숙하다고 가정합니다 (만약 아니면, 쿠다 C 프로그래밍 안내서를 참고해 주십시오).

평가, 병렬화, 최적화, 배포

이 안내서에서는 응용 프로그램의 평가, 병렬화, 최적화, 배포로 구성되는 설계 사이클이 소개됩니다. 이 사이클의 목표는 응용 프로그램 개발자들이 어떤 코드에서 GPU 가속으로 가장 쉽게 이득을 볼 수 있는 부분을 빨리 찾아, 신속하게 그 이득을 실현하고, 가능한 한 빨리 그 결과 가속을 제품에 이용하기 시작하는 것입니다.

평가, 병렬화, 최적화, 배포는 순환 프로세스입니다. 이 프로세스를 통해, 맨 처음 가속 결과가 가장 적은 시간 투자로 얻어지고, 시험 되고, 배포될 수 있습니다. 그리고 사이클은 다시 더 많은 최적화 기회들을 찾습니다. 더 가속될 수 있을 부분을 찾아, 훨씬 더 빠른 버전의 응용 프로그램을 제품으로 배포합니다.



평가(assess), 병렬화(parallelize), 최적화(optimize), 배포(deploy) 사이클

평가

기존 과제를 위한 첫 단계는 그 응용 프로그램에서 수행 시간을 가장 많이 차지하는 부분을 찾는 일입니다. 이 지식을 가지로, 개발자는 병렬화를 위해 그 병목 구간을 평가하고 GPU 로 가속할 수 있을지를 조사하기 시작할 수 있습니다.

최종 사용자의 요구와 제한 사항들을 이해함으로써 그리고 암달과 구스타프슨의 법칙을 적용함으로써, 개발자는 응용 프로그램의 핫스팟(hotspot)을 가속함으로써 얻을 수 있는 속도 향상의 상한을 결정할 수 있습니다.

병렬화

핫스팟들이 밝혀지고 목적과 기대치를 설정하기 위한 기본 연습이 끝나면, 개발자는 코드를 병렬화합니다. 원본 코드는 1) `cuBLAS`, `cuFFT`, 또는 `Thrust` 같은 기존 GPU 최적화된 라이브러리를 불러오으로써 또는 2) 적은 수의 전처리 디렉티브들(directives)을 추가하여 병렬화 컴파일러에게 힌트로 줌으로써 병렬화될 수 있습니다.

다른 한편, 몇몇 응용 프로그램들은 내재한 병렬성을 드러내기 위해 약간의 리팩토링을 요구할 것입니다. 미래의 GPU 아키텍처들조차 순차적 응용 프로그램들의

성능을 향상 시키기 위해 또는 단순히 성능을 유지하기 위해 병렬성 노출을 요구할 것입니다.

쿠다 병렬 프로그래밍 언어들(CUDA C/C++, CUDA Fortran 등)은 이 병렬성을 가능한 한 단순하게 표현하는 것을 목표로 하면서, 동시에 최대 병렬 처리량을 위해 설계된 쿠다 가능 GPU에서의 연산을 가능하게 합니다.

최적화

쓰일 수 있는 많은 최적화 기법들이 있습니다. 응용 프로그램의 요구들(needs)을 이해하는 것은 그 과정을 부드럽게 만드는 데 도움이 됩니다. 그러나, 평가, 병렬화, 최적화, 배포의 전체 사이클과 마찬가지로, 프로그램 최적화도 (최적화를 위한 기회를 찾고, 그 최적화를 적용 및 시험하고, 얻어진 속도 향상을 검증하고, 그 과정을 반복하는) 반복적 절차입니다. 프로그래머는 좋은 속도 향상을 보기도 전에 수많은 최적화 전략들을 기억하는 데 시간을 쓸 필요가 없습니다. 대신, 우리는 그 전략들을 점진적으로 학습하며 적용해 나갑니다.

계산과 검치는 데이터 전송들부터 부동 소수점 시퀀스들의 미세조정에 이르기까지 최적화는 다양한 수준에서 적용될 수 있습니다. 프로파일링 도구들은 이 절차를 가이드하는 데 매우 유용합니다. 이 도구들은 개발자들이 다음으로 진행하면 좋을 최적화 방법을 제안하고, 이 지침서의 적절한 최적화 절을 참고 자료로 제공합니다.

배포

응용 프로그램에서 하나 이상의 요소가 GPU 가속되면, 그 결과물은 원래 기대치와 비교될 수 있습니다. 초기 평가 단계에서 우리는 주어진 핫스팟들을 가속함으로써 얻을 수 있는 잠재적 속도 향상의 상한을 결정할 수 있었습니다.

전체적인 속도 향상을 위해 다른 핫스팟들을 추적하기 전에, 개발자는 부분적으로 병렬화된 구현을 제품에 적용하는 것을 고려해야 합니다. 이는 여러 이유로 중요합니다. 예를 들어, 이는 사용자가 가능한 한 빨리 투자에 대한 이익을 얻도록 돕습니다 (그 수행 시간 단축은 부분적일 수도 있지만, 여전히 소중한 것입니다). 그리고 이는 응용 프로그램을 혁명적이기보단 진화적인 방식으로 바꿈으로써 개발자와 사용자가 겪을 수 있는 위험을 최소화합니다.

권고와 모범 관례 (Recommendations and Best Practices)

이 지침서 곳곳에서, 구체적인 쿠다 C 코드 설계와 구현이 권고됩니다. 이 권고들은 우선순위에 따라 분류됩니다. 그 우선순위는 권고의 효과와 범위를 합친 것입니다. 대부분의 쿠다 응용 프로그램들에서 상당한 성능 향상을 보이는 조치들은 가장 높은

우선순위를 가집니다. 반면, 매우 특정한 상황들에서만 효과를 보이는 작은 최적화들에는 더 낮은 우선순위가 주어집니다.

더 낮은 우선순위 권고들을 구현하기 전에, 이미 적용된 기법과 관련된 더 높은 우선순위 권고들이 지켜 졌는지 확실히 하는 것이 좋습니다. 이 접근법은 투자 시간 대비 가장 좋은 결과를 보이는 경향이 있습니다. 이 접근법을 사용하여, 여러분은 성숙하지 않은 최적화에서 생길 수 있는 문제들을 피할 수 있을 것입니다.

우선순위의 이점과 범위의 판정 기준은 프로그램의 특성에 따라 바뀝니다. 이 지침서에서, 만들어진 우선순위들은 한 전형적인 경우를 나타냅니다. 여러분의 코드는 다른 요소들을 반영할 수도 있습니다. 그렇다 하더라도, 더 낮은 우선순위 항목들이 수행되기 전에 더 높은 우선순위 권고들을 못 보고 넘어가지 않도록 검증하는 습관을 들이는 것이 좋습니다.

주의: 이 지침서에 있는 코드 샘플들은 간결함을 위해 오류 검사를 생략합니다. 그러나, 제품 코드에서는 꼭 체계적으로 오류 코드를 확인해야 합니다. 오류 코드는 각 응용 프로그램 인터페이스 (API) 호출에 의해 리턴됩니다. 그리고 `cudaGetLastError()`를 호출함으로써 커널 실행이 실패하지 않았는지 확인해야 합니다.

1. 응용 프로그램 평가하기 (Assessing Your Application)

슈퍼컴퓨터에서 휴대 전화까지, 현대 프로세서들은 점점 더 성능을 내기 위해 병렬성에 의존합니다. 제어, 산술, 레지스터들, 그리고 전형적인 약간의 캐시를 포함하는 핵심 계산 유닛은 여러 개로 자기 복제됩니다. 그리고 네트워크를 통해 메모리로 연결됩니다. 결과적으로, 그 프로세서의 계산 능력을 모두 사용하기 위해, 모든 현대 프로세서는 병렬 코드를 요구합니다.

프로세서들이 더 작은 입자로 된 병렬성을 프로그래머에게 드러내기 위해 진화하고 있지만, 많은 기존 응용 프로그램들은 직렬 코드로 또는 큰 입자로 된 병렬 코드로 진화해 왔습니다 (예를 들어, 데이터가 병렬로 처리되는 영역들로 분해된 곳에서, MPI 를 사용하여 공유된 부영역들을 가진). GPU 가 포함된 한 현대 프로세서 아키텍처에서 이득을 얻기 위한 첫 단계는 핫스팟을 찾기 위해 그 응용 프로그램들을 평가하고, 그것이 병렬화될 수 있는지 결정하고, 지금과 미래에 적절한 작업량을 이해하는 일입니다.

2. 이기종 컴퓨팅 (Heterogeneous Computing)

쿠다 프로그래밍은 실행 코드를 다른 두 플랫폼에 동시에 연루시킵니다. 그 두 플랫폼은 하나 이상의 중앙 처리 장치(CPU)를 가진 호스트 하나와 한 개 이상의 쿠다 가능 엔비디아 GPU 디바이스들입니다.

비록 엔비디아 GPU 들이 그래픽스에 자주 관련되나, 엔비디아 GPU 는 또한 수천 개의 가벼운 스레드들을 병렬로 실행할 수 있는 강력한 산술 엔진입니다. GPU 는 병렬 실행으로 이득을 얻을 수 있는 계산에 적합합니다.

그러나 디바이스는 호스트 시스템과는 뚜렷하게 다른 설계에 기반을 두고 있습니다. 쿠다를 효과적으로 사용하기 위해서는 어떻게 그 차이가 쿠다 응용 프로그램의 성능을 결정하는지를 이해해야 합니다.

2.1. 호스트와 디바이스 사이 차이 (Differences between Host and Device)

주요 차이들은 스레딩 모델(threading model)과 분리된 물리 메모리입니다.

스레딩 자원 (Threading resources)

호스트 시스템의 실행 파이프라인들은 제한된 수의 동시 스레드만 지원할 수 있습니다. 오늘날 여섯 개 코어를 가진 프로세서 네 개를 가진 서버들은 오직 24 개 스레드만 동시에 실행할 수 있습니다 (만약 중앙 처리 장치가 하이퍼스레딩을 지원하면 48 개). 그에 비해, 쿠다 디바이스에서 실행할 수 있는 가장 작은 병렬성 유닛은 스레드 32 개입니다 (우리는 이를 한 워프라 부릅니다). 현대 엔비디아 GPU 들은 멀티프로세서 당 1536 개의 활성화된 스레드들을 동시에 지원할 수 있습니다 (쿠다 C 프로그래밍 지침서의 [특징과 설계 명세서](#) 절을 보십시오). 16 개 멀티프로세서를 가진 GPU 에서는 동시에 24,576 개의 활성 스레드들을 동시에 지원할 수 있습니다.

스레드 (Threads)

중앙 처리 장치의 스레드들은 일반적으로 무겁습니다. 멀티스레딩 능력을 제공하기 위해, 운영 체제는 반드시 스레드들을 중앙 처리 장치 실행 채널 안 또는 밖으로 교체해야(swap) 합니다. 그러므로 (두 스레드가 교환될 때) 문맥 교환은 느리고 비쌉니다. 그에 비해, GPU 상의 스레드들은 극히 가볍습니다. 한 전형적인 시스템에서, 수천 개의 스레드들은 작업을 위해 (각자 32 개 스레드로 구성된 워프 단위로) 큐(queue)에 쌓입니다. 만약 GPU 가 한 워프의 스레드들을 반드시 기다려야 하면, GPU 는 단순히 또다른 워프를 실행하기 시작합니다. 모든 활성화된 스레드들에 별도의 레지스터가 할당되므로, GPU 스레드들을 교환할 때 레지스터들 또는 다른 상태(state)의 교환은 발생할 필요가 없습니다. 각 스레드가 그 스레드의 실행을 완전히 끝마칠 때까지, 자원들은 각 스레드에 할당되어 있습니다. 요약하면, 중앙 처리 장치 코어들은 한 번에 각각 하나 또는 두 스레드의 지연 시간을 최소화하기

위해 설계되었습니다. 반면, GPU 는 처리량을 최대화하기 위해, 많은 수의 가벼운
동시 스레드들을 다루기 위해 설계되었습니다.

램 (RAM)

호스트 시스템과 디바이스는 각각 자신의 별도로 부착된 물리 메모리들을 가집니다.
호스트와 디바이스 메모리들은 PCI 익스프레스(PCIe) 버스로 분리되므로, 호스트
메모리에 있는 항목들은 반드시 그 버스를 건너 디바이스 메모리로 또는 그 반대로
통신됩니다 ([무엇이 쿠다 가능 디바이스에서 실행되나요?](#)에서 설명되었듯).

이것이 병렬 프로그래밍에서, CPU 호스트들과 GPU 디바이스들 사이 주요한 하드웨어
차이입니다. 그 밖의 차이는 이 문서에 나올 때 다시 논의됩니다. 이 차이들을 가진 응용
프로그램들은 호스트와 디바이스를 하나의 결합된 이기종 시스템으로 다뤄질 수 있습니다.
그 시스템에서 최선의 작업을 하기 위해 각 처리 유닛이 이용됩니다. 호스트에서는 순차적
작업이 그리고 디바이스에서는 병렬 작업이 수행됩니다.

2.2. 무엇을 쿠다 가능 디바이스에서 실행하나요?

한 응용 프로그램에서 어떤 부분을 디바이스에서 실행할 지를 결정할 때, 다음 논점들이
고려되어야 합니다.

- 이상적으로, 디바이스는 많은 데이터 요소들이 동시에 병렬로 실행될 수 있는
계산에 적합합니다. 이는 (행렬 같은) 큰 데이터 세트들의 연산과 관련됩니다. 이
같은 연산은 (수백만까지는 아니어도) 수천 개 요소에 대해 동시에 수행될 수
있습니다. 이는 쿠다에서 좋은 성능을 얻기 위한 요건입니다. 소프트웨어는 반드시
동시에 많은 수(일반적으로 수천 또는 수만 개)의 스레드를 사용할 수 있어야
합니다. 많은 스레드를 병렬로 처리하기 위한 지원은 위에서 설명된 쿠다의 가벼운
스레딩 모델에서 나온 것입니다.
- 가장 좋은 성능을 얻기 위해, 디바이스에서 실행되고 있는 인접한 스레드들은
메모리에 응집된 접근을 해야 합니다. 어떤 메모리 접근 패턴은 여러 데이터
아이템들의 읽기 또는 쓰기 연산들을 하드웨어가 한 번의 연산으로 응집할 수 있게
합니다. 응집될 수 없게 펼쳐진 데이터, 또는 L1 또는 텍스처 캐시가 효과적으로
쓰일 수 없게 배치된 데이터는 쿠다 연산에서 더 적은 속도 향상을 보이는 경향이
있습니다.
- 쿠다를 쓰기 위해, 데이터값들은 호스트에서 디바이스로 PCI 익스프레스 (PCIe)
버스를 통해 전송되어야 합니다. 이 전송들은 (전송 속도가 느려) 성능에 나쁜
영향을 주므로 최소화되어야 합니다. ([호스트와 디바이스 사이 데이터 전송을](#)
보십시오.) 이 비용은 몇몇 파생된 결과들을 가집니다.

- 연산들의 복잡도는 디바이스로 또는 디바이스에서 데이터를 이동하는 비용을 정당화할 수 있어야 합니다. 코드를 간결히 쓰기 위해, 적은 수의 스레드로 데이터를 전송하는 코드는 성능 이득을 거의 못 볼 것입니다. 이상적 시나리오는 많은 스레드들로 상당한 크기의 작업을 수행하는 것입니다.
- 예를 들어, 행렬 덧셈을 위해 디바이스로 두 행렬을 전송했다가 그 결과를 다시 호스트로 전송하는 것은 큰 성능 향상을 얻을 수 없을 것입니다. 여기서 핵심은 전송된 데이터 요소 당 수행되는 연산들의 횟수입니다. 행렬들의 차원이 $N \times N$ 이라 가정합니다. 연산(덧셈)은 N^2 개 그리고 전송되어야 하는 요소들은 $3N^2$ 개 있습니다. 그러므로 전송된 요소 개수에 대한 연산 개수의 비율은 1:3 또는 $O(1)$ 입니다. 성능 이득은 이 비율이 더 높을 때 더 쉽게 얻어집니다. 예를 들어, 같은 행렬들에 대한 행렬 곱은 N^3 연산(곱-합)을 요구합니다. 따라서, 전송된 요소 대 연산 횟수의 비율은 $O(N)$ 입니다. 이 경우, 행렬이 더 커질수록 성능 이득도 더 많아집니다. 연산 타입도 한 가지 추가 요소입니다. 예를 들어 덧셈은 삼각 함수와는 다른 복잡도 프로파일을 가집니다. 어떤 연산들이 호스트 또는 디바이스에서 수행되어야 할지를 결정할 때, 호스트와 디바이스 사이 데이터 전송 시간도 꼭 추가되어야 합니다.
- 데이터는 디바이스에 가능한 한 오래 머물러야 합니다. 같은 데이터에 대해 여러 커널들을 실행하는 프로그램은 전송 횟수를 줄여야 하므로 커널 호출 사이에도 데이터를 되도록 디바이스에 남겨두어야 합니다. 즉, 다음 계산을 위해 중간 결과를 호스트로 전송했다가 다시 디바이스로 가져와서는 안 됩니다. 그래서, 이전 예제에서, 더해진 두 행렬은 이전 계산 결과로서 그 디바이스에 머물러 있었습니다. 만약 그 덧셈 결과가 다음 계산에도 쓰일 수 있으면, 행렬 덧셈은 디바이스에서 지역적으로 수행됩니다. 계산 시퀀스의 단계 중 하나가 호스트에서 수행되는 게 더 빠를 때조차 이 접근이 쓰여야 합니다. 만약 한 번 이상 PCIe 전송을 피할 수 있으면, 상대적으로 느린 커널을 실행하는 것조차 이로울 수도 있습니다. [호스트와 디바이스 사이 데이터 전송](#)에서는 호스트와 디바이스 사이 대역폭 측정 및 디바이스 안에서의 대역폭에 대한 더 자세한 사항이 소개됩니다.

3. 응용 프로그램 프로파일링 (Application Profiling)

3.1. 프로파일 (Profile)

많은 코드들이 상대적으로 적은 양의 코드로 작업의 많은 부분을 처리합니다. 프로파일러를 사용하여, 개발자는 그런 핫스팟들을 찾을 수 있고 병렬화해야 할 후보들의 목록을 작성할 수 있습니다.

3.1.1. 프로파일 만들기 (Creating the Profile)

코드를 프로파일링하기 위해 사용할 수 있는 많은 접근법들이 있습니다. 모든 경우들에서, 목적은 그 응용 프로그램에서 가장 많은 수행 시간을 쓰고 있는 함수 또는 함수들을 찾아내는 것입니다.

주의: 우선순위 높음: 개발자 생산성을 최대화하기 위해, 핫스팟과 병목을 찾기 위해 응용 프로그램 프로파일하십시오.

프로파일링에서 가장 중요한 고려 사항은 작업량이 현실적인지 확실히 하는 것입니다. 다시 말해, 시험에서 얻은 정보와 그 정보에 기반을 둔 결정들이 실제 데이터와 관련되어 있는지 확실히 하는 것입니다. 비현실적인 작업량은 최적이지 아닌 결과로 이끌 수 있고 노력이 낭비되게 만들 수 있습니다. 개발자들이 비현실적인 문제 크기를 위해 최적화하게 함으로써 그리고 개발자들이 틀린 함수들에 집중하게 함으로써 말이지요.

프로파일을 만드는 데 쓰일 수 있는 많은 도구들이 있습니다. 다음 예제는 gprof 에 기반을 둡니다. gprof 은 지엔유 바이너리 유틸 컬렉션(GNU Binutils collection)에 있는 리눅스 플랫폼을 위한 오픈 소스 프로파일러입니다.

```
$ gcc -O2 -g -pg myprog.c
$ gprof ./a.out > profile.txt
Each sample counts as 0.01 seconds.
```

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
33.34	0.02	0.02	7208	0.00	0.00	genTimeStep
16.67	0.03	0.01	240	0.04	0.12	calcStats
16.67	0.04	0.01	8	1.25	1.25	calcSummaryData
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01				mcount
0.00	0.06	0.00	236	0.00	0.00	tzset
0.00	0.06	0.00	192	0.00	0.00	tolower
0.00	0.06	0.00	47	0.00	0.00	strlen
0.00	0.06	0.00	45	0.00	0.00	strchr
0.00	0.06	0.00	1	0.00	50.00	main
0.00	0.06	0.00	1	0.00	0.00	memcpy
0.00	0.06	0.00	1	0.00	10.11	print
0.00	0.06	0.00	1	0.00	0.00	profil
0.00	0.06	0.00	1	0.00	50.00	report

3.1.2. 핫스팟 찾기 (Identifying Hotspots)

위 예제에서, 우리는 `genTimeStep()` 함수가 전체 수행 시간의 삼 분의 일을 차지하고 있음을 볼 수 있습니다. 이 함수는 우리의 첫 번째 병렬화를 위한 후보 함수가 되어야 합니다. [스케일링 이해하기](#)는 우리가 그 병렬화를 통해 기대할 수 있는 잠재적 이득에 대해 논의합니다.

위 예제에서 `calcStats()`와 `calcSummaryData()` 같은 몇몇 다른 함수들도 전체 수행 시간에서 상당한 부분을 차지합니다. 이 함수들을 병렬화하는 것 또한 잠재적으로 우리 응용 프로그램을 더 가속할 것입니다. 그러나, 평가, 병렬화, 최적화, 배포가 한 사이클이므로, 우리는 이 함수들을 다음 사이클에서 병렬화하고자 합니다. 그렇게 함으로써 현재 우리가 해야 할 작업 범위를 점진적으로 바뀌는 더 작은 일들로 좁힐 수 있습니다.

3.1.3. 스케일링 이해하기 (Understanding Scaling)

한 응용 프로그램에서 얻을 수 있는 성능 이득의 양은 그 응용 프로그램이 쿠다를 통해 병렬화될 수 있는 정도(extent)에 전적으로 의존합니다. 충분히 병렬화되지 않은 코드는 호스트에서 실행되어야 합니다. 그렇지 않으면 호스트와 디바이스 사이 초과 전송만 유발될 것이기 때문입니다.

노트: 우선순위 높음: 쿠다에서, 최대 이득을 얻기 위해, 우선 순차 코드를 병렬화 하는 방법을 찾는 데 집중하십시오.

어떻게 응용 프로그램들이 스케일할 수 있는지 이해함으로써, 점진적 병렬화 전략을 계획하고 기대치를 설정할 수 있습니다. [강한 스케일링과 암달의 법칙](#)은 강한 스케일링을 설명합니다, 이는 우리가 고정된 문제 크기를 가진 속도 향상의 상한을 정할 수 있게 합니다. [약한 스케일링과 구스타프슨의 법칙](#)은 약한 스케일링 설명합니다, 여기서 가속은 문제 크기가 증가함에 따라 얻어집니다. 많은 응용 프로그램들에서, 강한 그리고 약한 스케일링을 조합하는 것이 바람직합니다.

3.1.3.1. 강한 스케일링과 암달의 법칙 (Strong Scaling and Amdahl's Law)

전체 문제 크기가 고정된 경우, 강한 스케일링은 한 시스템에 더 많은 프로세서가 추가될수록 어떻게 솔루션까지 도달하는 시간을 줄일지를 측정하는 척도입니다. 선형 강한 스케일링을 보이는 한 응용 프로그램은 사용된 프로세서 수만큼 속도가 향상됩니다.

강한 스케일링은 주로 암달의 법칙과 동일시됩니다. 암달의 법칙은 순차 코드의 일부를 병렬화하여 얻을 수 있는 최대 속도 향상을 특정합니다. 특히, 암달의 법칙은 한 프로그램 최대 속도 향상 S 를 설명합니다.

$$S = 1 / (1 - P) + P / N$$

여기서 P 는 전체 수행 시간에서 병렬화 될 수 있는 코드의 수행 시간이 차지하는 비율입니다. 그리고 N 은 코드의 병렬 부분이 동작하는 프로세서 개수입니다.

M (프로세서 개수)이 더 커질수록, P/N 부분은 더 작아집니다. N 을 매우 큰 수로 보면, 이 식은 더 단순해집니다. 다시 말해, 위 등식은 $S = 1 / (1 - P)$ 이 됩니다. 이제, 만약 한 순차 프로그램에서 실행 시간의 $3/4$ 이 병렬화되면, 순차 코드의 최대 속도 향상은 $1 / (1 - 3/4) = 4$ 입니다.

현실에서, 대부분 응용 프로그램들은 완전한 선형 강한 스케일링을 보이지 않습니다. 응용 프로그램들이 어느 정도는 강한 스케일링을 보인다 하더라도 말이죠. 대부분, 요점은 병렬화할 수 있는 부분 P 가 더 커질수록, 잠재적 속도 향상도 더 커진다는 점입니다. 반대로, 만약 P 가 작은 수라면 (다시 말해, 그 응용 프로그램이 충분히 병렬화될 수 없다면), 프로세서 수(M) 증가는 성능 향상에 거의 도움이 안됩니다. 그러므로, 고정된 크기의 문제에서 가장 큰 속도 향상을 얻기 위해서는, P 를 높여야 합니다. 다시 말해, 병렬화될 수 있는 코드의 양을 최대화해야 합니다.

3.1.3.2. 약한 스케일링과 구스타프슨의 법칙 (Weak Scaling and Gustafson's Law)

프로세서 당 고정된 크기의 문제를 가진 시스템에서, 약한 스케일링은 더 많은 프로세서가 추가됨에 따라 어떻게 솔루션까지 도달하는 데 걸리는 시간이 바뀌는지를 측정하는 척도입니다. 다시 말해, 전체 문제 크기는 프로세서 수가 증가함에 따라 증가합니다.

약한 스케일링은 종종 구스타프슨의 법칙과 동일시됩니다. 구스타프슨의 법칙은 문제 크기가 프로세서 수에 따라 함께 바뀔을 말합니다. 이 때문에, 한 프로그램의 최대 속도 향상 S 는 다음과 같습니다.

$$S = N + (1 - P) (1 - N)$$

여기서 P 는 전체 직렬 실행 시간에서 병렬화될 수 있는 코드가 차지하는 실행 시간입니다. 그리고 N 은 코드의 병렬 부분을 실행하는 프로세서 개수입니다.

구스타프슨의 법칙을 보는 또 다른 방법은 우리가 시스템 크기를 키움에 따라 상수 남는 것이, 문제 크기가 아니라 실행 시간이라는 점입니다. 유념하십시오. 구스타프슨의 법칙은 직렬 대 병렬 실행의 비율이 상수라고 가정합니다. 더 큰 문제를 다루고 설정하는 다른

추가

비용을

반영하면서요.

3.1.3.3. 강한 그리고 약한 스케일링 적용하기 (Applying Strong and Weak Scaling)

어떤 타입의 스케일링이 그 응용 프로그램에 가장 적절한지 이해하는 것은 응용 프로그램의 속도가 얼마나 향상될지 추정하는 데 중요합니다. 몇몇 응용 프로그램에서, 문제 크기는 상수로 고정됩니다. 따라서 오직 강한 스케일링만 적용될 수 있습니다. 예를 들어, 분자 개수(sizes)가 고정된 한 예제는 어떻게 두 분자가 서로 상호작용하는지를 모델링할 것입니다.

다른 응용 프로그램들에서, 문제 크기는 사용 가능한 프로세서들을 채우기 위해 증가합니다. 그 예로는 그물망(meshes) 또는 격자(grids) 같은 유체 또는 구조물과 몬테카를로 시뮬레이션들이 포함됩니다. 이 문제에서는 문제 크기가 증가할수록 정확도도 증가합니다.

응용 프로그램 프로파일을 이해했으면, 개발자는 다음 두 가지를 이해해야 합니다. 1) 만약 계산 성능이 바뀌면 문제 크기가 어떻게 바뀔지. 그리고 2) 최대 속도 향상을 정하기 위해 암달의 법칙을 적용할지 구스타프슨의 법칙을 적용할지.

4. 응용 프로그램 병렬화하기 (Parallelizing Your Application)

핫스팟들을 찾았고 목표와 기대치 설정을 위한 기본 활동이 끝났으면, 다음 차례는 코드를 병렬화하는 것입니다. 원래 코드를 가지고, 이는 `cuBLAS`, `cuFFT`, 또는 `Thrust` 같은 기존 GPU 에서 최적화된 라이브러리를 호출함으로써 간단히 수행될 수 있습니다. 또한, 이는 병렬화 컴파일러에 전처리기 디렉티브를 추가하여 힌트를 줌으로써 간단히 수행될 수 있습니다.

반면, 몇몇 응용 프로그램은 내재된 병렬성을 드러내기 위해 약간의 리팩토링이 필요할 것입니다. 미래 CPU 구조들조차 순차 응용 프로그램의 성능 향상 또는 성능 유지를 위해 이 병렬성 노출을 요구할 것입니다. 따라서, 쿠다 병렬 프로그래밍 언어들(CUDA C/C++, CUDA Fortran 등)의 목표는 1) 병렬성을 가능한 단순하게 표현하는 것과 2) (최대 병렬 처리량을 위해 설계된) 쿠다 가능 GPU 들 에서 연산을 가능하게 하는 것입니다.

5. 시작하기 (Getting Started)

순차 코드를 병렬화하기 위한 몇 가지 핵심 전략이 있습니다. 어떻게 이 전략들을 특정 응용 프로그램에 적용할지는 복잡하고 문제마다 다른 주제지만, 여기에 열거된 일반적 주제들은 멀티 코어 중앙 처리 장치에서 동작하는 코드이든 쿠다 GPU 에서 동작하는 코드이든 상관없이 적용될 수 있습니다.

5.1. 병렬 라이브러리 (Parallel Libraries)

응용 프로그램을 병렬화하기 위한 가장 간단한 접근법은 기존 라이브러리들을 이용하는 것입니다. 그 기존 라이브러리들은 병렬 구조를 잘 이용하도록 설계되었습니다. 쿠다 도구 모음(CUDA Toolkit)에는 cuBLAS 와 cuFFT 등과 같은 엔비디아 쿠다 GPU 를 위해 미세조정된 라이브러리들이 많이 포함되어 있습니다.

여기서 핵심은 라이브러리들이 응용 프로그램의 요구(needs)와 잘 어울릴 때, 라이브러리들이 가장 유용하다는 것입니다. 예를 들어, 이미 다른 BLAS 라이브러리를 사용하고 있는 응용 프로그램은 꽤 쉽게 cuBLAS 로 바뀔 수 있습니다. 반면, 선형 대수를 거의 하지 않는 응용 프로그램들은 cuBLAS 를 거의 쓰지 못할 것입니다. 이는 다른 쿠다 도구모음 라이브러리들에도 똑같이 적용됩니다. 이를테면, cuFFT 는 FFTW 와 비슷한 인터페이스를 가지고 있는 식입니다.

또한 주목할 것은 스러스트(Thrust) 라이브러리입니다. 스러스트는 C++ STL(Standard Template Library)과 비슷한 병렬 C++ 템플릿 라이브러리입니다. 스러스트는 scan, sort, 그리고 reduce 같은 풍부한 데이터 병렬 기본 요소들을 제공합니다. 이 요소들을 조합하여 우리는 간결하고 잘 읽히는 소스 코드를 구현할 수 있습니다. 이 고수준 추상화들의 조합으로 우리의 계산을 묘사함으로써, 우리는 가장 효율적인 구현을 쉽게 얻을 수 있습니다. 결과적으로, 스러스트는 강인성과 절대 성능이 중요한 제품 뿐 아니라 프로그래머 생산성이 가장 중요한 쿠다 응용 프로그램의 빠른 프로토타이핑에도 활용될 수 있습니다.

5.2. 컴파일러 병렬화하기 (Parallelizing Compilers)

또다른 순차 코드들을 병렬화하기 위한 흔한 접근법은 병렬화 컴파일러를 이용하는 것입니다. 종종 이는 디렉티브 기반 접근을 뜻합니다. 여기서 근본적인 코드 자체를 고치거나 개작할 필요 없이 어디에서 병렬성을 찾을 수 있는지에 대한 힌트를 컴파일러에게

주기 위해, 프로그래머는 `pragma` 또는 다른 비슷한 표기법을 사용합니다. 병렬성을 컴파일러에게 드러냄으로써, 디렉티브들은 컴파일러가 계산을 병렬 구조로 매핑(mapping)하는 세부 작업을 할 수 있게 합니다.

OpenACC 스탠더드(standard)는 꼭 호스트에서 디바이스로 복사되어야하는 루프들과 코드 영역들을 스탠더드 C, C++, 그리고 Fortran 으로 특정하기 위한 일련의 컴파일러 디렉티브들을 제공합니다. 가속기 디바이스를 관리하는 세부사항들은 OpenACC 가능 컴파일러와 런타임에 의해 암묵적으로 다뤄집니다.

자세한 사항들은 <http://www.openacc.org/>을 참고하십시오.

5.3. 병렬성을 드러내기 위한 코딩 (Coding to Expose Parallelism)

기존 병렬 라이브러리들 또는 병렬화 컴파일러들에서 제공할 수 있는 추가 기능이나 성능 이상이 필요한 응용 프로그램들에서, 기존 순차 코드와 매끄럽게 결합될 수 있는 쿠다 C/C++ 같은 병렬 프로그래밍 언어는 필수입니다.

일단 응용 프로그램의 프로파일 평가에서 핫스팟 위치를 찾았고 기존 코드가 가장 좋은 접근법이라고 결정되었으면, 쿠다 커널로 우리 코드의 일부에 있는 병렬성을 드러내기 위해 우리는 쿠다 C/C++를 쓸 수 있습니다. 우리는 이 커널을 GPU 에서 실행할 수 있습니다. 우리는 응용 프로그램의 나머지 코드들을 크게 바꾸지 않고도 결과를 얻을 수 있습니다.

이 접근법은 응용 프로그램의 전체 수행 시간의 대부분이 코드의 고립된 일부에서 소모될 때 가장 간단합니다. 매우 고른 프로파일을 가진 응용 프로그램들은 병렬화하기가 더 어렵습니다. 다시 말해, 응용 프로그램의 넓은 부분에서 수행 시간을 고르게 소모하는 응용 프로그램들은 병렬화하기가 더 어렵습니다. 고른 프로파일을 가진 응용 프로그램에서는 응용 프로그램에 내재된 병렬성을 드러내기 위해 약간의 리팩토링이 필요할 수도 있습니다. 그러나 유념하십시오. 이 리팩토링 작업은 CPU 그리고 GPU 와 비슷한 미래 구조들에서도 도움이 될 것입니다. 따라서, 그 작업에 노력을 기울일 가치가 있습니다.

6. 바른 정답 얻기 (Getting the Right Answer)

바른 정답을 얻는 것은 분명히 모든 계산에서 중요한 목표입니다. 병렬 시스템에서는 전통적인 순차 프로그램들에서는 볼 수 없던 문제들이 생길 수 있습니다. 여기에는 스레딩

이슈, 부동 소수점 값들이 계산되는 방식 차이 때문에 생기는 기대하지 않은 값들, 그리고 CPU 와 GPU 프로세서가 동작하는 방식 차이 때문에 생기는 어려움들이 포함됩니다. 이 장은 리턴된 데이터의 정확성에 영향을 끼칠 수 있는 이슈들을 조사하고 적절한 해결책들을 제시합니다.

6.1. 검증 (Verification)

6.1.1. 참조 비교 (Reference Comparison)

기존 프로그램을 바꾸고 그것이 정확함을 검증하기 위해 중요한 것은 대표적인 입력에서 얻은 이전에 잘 알려진 좋은 참조 출력과 새 결과들을 비교할 수 있는 어떤 메커니즘을 만드는 것입니다. 코드를 바꾼 뒤에는, 특정 알고리즘에 어떤 판정 기준이든 적용하여 얻어진 결과가 그 판정 기준을 통과하는지 꼭 확인하십시오. 몇몇 사람들은 비트 단위까지 같은 결과가 나오길 기대할 것입니다. 그러나 특히, 부동 소수점 연산이 고려되는 응용에서, 그것이 항상 가능하지는 않습니다. [수치 정확도와 정밀도](#)를 보십시오. 다른 알고리즘에서, 만약 구현들과 참조 결과 사이 오차가 매우 작은 값인 엡실론(epsilon)보다 작으면, 우리는 그 구현이 옳다고 여길 수도 있습니다.

유념하십시오. 수치 결과들을 검증하기 위해 사용되는 절차들은 성능 결과들을 검증하기 위해서도 쉽게 확장될 수 있습니다. 우리는 코드가 바뀔 때마다 우리가 바꾼 코드가 정확한지와 성능을 향상시키는지(그리고 얼마나 향상시키는지)를 확실히 알고 싶어 합니다. 순환적 평가, 병렬화, 최적화, 배포 절차의 중요한 부분으로 이 결과들을 자주 확인함으로써 우리는 우리가 바라는 결과에 최대한 빨리 도달할 수 있을 것입니다.

6.1.2. 단위 테스트 (Unit Testing)

위에서 설명된 참조 비교와 대응되는 방법은 유닛 수준에서 쉽게 검증할 수 있도록 코드 자체를 구조화하는 것입니다. 예를 들어, 우리는 우리의 쿠다 커널들을 하나의 큰 단일 `__global__` 함수가 아닌 많은 짧은 `__device__` 함수들로 구성할 수 있습니다. 그 함수들을 다함께 [후킹](#)하기 전에, 각 디바이스 함수는 독립적으로 시험될 수 있습니다.

예를 들어, 많은 커널들은 실제 계산 외에도 메모리 접근을 위한 복잡한 어드레싱(addressing) 논리를 가집니다. 대량의 계산을 하기 전에, 만약 우리가 우리의 어드레싱 논리를 따로 검증하면, 이는 나중에 디버깅에 들어가는 수고를 줄일 것입니다. (주의하십시오. 쿠다 컴파일러는 글로벌 메모리에 결과를 쓰기 않는 디바이스 코드는 죽은 것으로 여겨 제거 대상으로 삼습니다. 그러므로 이 전략을 성공적으로 적용하기 위해, 우리는 최소한 어드레싱 논리의 결과로 무언가는 글로벌 메모리로 내보내 써야합니다.)

한 단계 더 나아가서, 만약 대부분의 함수들이 단지 `__device__`가 아니라 `__host__ __device__`로 정의되어 있다면, 이 함수들은 시피유와 GPU 모두에서 시험될 수 있습니다. 그렇게 함으로써, 그 함수의 출력이 정확하고 결과에 예상치 못한 차이가 없을 것이라는 확신을 높일 수 있습니다. 만약 차이가 있더라도, 그 차이들은 빨리 발견될 것이고 단순한 함수의 맥락에서 이해될 수 있을 것입니다.

유용한 부작용으로, 이 전략은 우리에게 우리의 응용 프로그램에서 CPU 와 GPU 실행 경로 모두에 포함되어야 하는 코드의 중복을 피할 수 있는 방법을 제공합니다. 만약 우리의 쿠다 커널에서 대부분의 작업이 `__host__ __device__` 함수에서 수행되면, 우리는 그 함수들을 호스트 코드와 디바이스 코드 모두에서 중복 없이 쉽게 호출할 수 있습니다.

6.2. 디버깅 (Debugging)

CUDA-GDB 는 리눅스와 맥에서 동작하는 GNU 디버거의 한 포트(port)입니다. <http://developer.nvidia.com/cuda-gdb> 를 보십시오.

마이크로소프트 비주얼 스튜디오를 위한 무료 플러그인으로 마이크로소프트 윈도우즈 비스타와 윈도우즈 7 을 위한 엔비디아 페럴렐 엔사이트(NVIDIA Parallel Nsight) 디버깅과 프로파일링 도구 또한 사용할 수 있습니다. <http://developer.nvidia.com/nvidia-parallel-nsight> 를 보십시오.

몇몇 제 3 의(third-party) 디버거들도 이제 쿠다 디버깅을 지원합니다. 더 자세한 사항들은 <http://developer.nvidia.com/debugging-solutions> 을 보십시오.

6.3. 수치 정확도와 정밀도 (Numerical Accuracy and Precision)

부정확하거나 기대하지 않은 결과들은 주로 부동 소수점 정확도 이슈들에서 발생합니다. 이는 (GPU 에서) 부동 소수점 값들이 계산되고 저장되는 방식이 (CPU 에서와) 다르기 때문입니다. 다음 절들은 흥미로운 주요 항목들을 설명합니다. 부동 소수점 연산의 다른 특이한 점들은 쿠다 C 프로그래밍 가이드의 특징과 기술적 설계 명세서 그리고 웨비나(webinar)와 백서(whitepaper)에서 제시됩니다.

6.3.1. 단정밀 대 배정밀 (Single vs. Double Precision)

계산 능력 1.3 이상을 가진 디바이스들은 기본적으로 배정밀 (64 비트 값) 부동 소수점 값들을 제공합니다. 반올림 이슈와 더 큰 정밀도 때문에, 배 정밀 연산을 사용해 얻어진 결과들은 단 정밀 연산을 통해 수행된 같은 연산 결과와 자주 다릅니다. 그러므로, 비슷한 정밀도의 값들을 비교하여 (두 결과가 같은지) 확실히 하는 것은 중요합니다. 그리고 두 결과가 정확하게 같을 것이라고 기대하기보단 그 결과들이 특정한 오차 범위 안에서 거의 같게 표현된다고 보는 편이 낫습니다.

6.3.2. 부동 소수점 수학은 결합적이지 않다 (Floating Point Math Is not Associative)

각 부동 소수점 산술 연산은 일정량의 반올림과 연루됩니다. 결과적으로, 산술 연산들이 수행되는 순서는 중요합니다. 만약 A, B, 그리고 C 가 부동 소수점 값들이면, $(A+B)+C$ 와 $A+(B+C)$ 의 결과는 같다고 보장되지 않습니다. 우리가 계산들을 병렬화할 때, 잠재적으로 연산 순서를 바꿉니다. 따라서, 병렬 결과들이 순차적 결과들과 일치하지 않을 수도 있습니다. 이 한계는 쿠다 뿐 아니라 부동 소수점 값들에 대한 병렬 계산들에 두루 나타나는 문제입니다.

6.3.3. 더블 형으로 승격 그리고 플로트 형으로 강등 (Promotions to Doubles and Truncations to Floats)

호스트와 디바이스에 있는 float 변수들의 계산 결과들을 비교할 때, 호스트 상의 배정밀 승격이 수치 결과가 달라지는 이유가 되지 않도록 확실히 하십시오. 예를 들어, 만약 다음과 같은 코드가 있다고 가정합시다.

```
float a;  
...  
a = a*1.02;
```

계산 능력 1.2 이하의 디바이스에서 또는 (위에서 언급되었듯) 배 정밀을 쓰지 않도록 컴파일된 계산 능력 1.3 을 가진 디바이스에서 수행되었다면, 곱셈은 단 정밀로 수행되었을 것입니다. 그러나, 만약 그 코드가 호스트에서 수행되었다면, 1.02 는 배 정밀 양(quantity)으로 해석되어 double 형으로 승격되었을 것입니다. 따라서 곱셈이 배 정밀로 수행된 결과가 다시 float 형으로 강등됨으로써 살짝 다른 결과를 만들어 냈을 것입니다. 그러나, 만약 1.02 를 1.02f 로 바꾸었다면, 그 결과가 double 형으로 승격되지 않아 모든 경우들에서 같았을 것입니다. 계산들이 단 정밀로 수행됨을 보장하기 위해, 항상 float 리터럴들을 사용하십시오.

정확도 외에도, [명령어 최적화](#)에 논의되었듯 double 과 float 사이 상호 변환은 성능에 나쁜 영향을 끼칩니다.

[6.3.4. IEEE 754 준수 \(Compliance\)](#)

모든 쿠다 디바이스들은 몇몇 예외들을 제외하고 이진 부동 소수점 표현을 위해 IEEE 754 표준을 따릅니다. (그러나) 쿠다 C 프로그래밍 가이드의 특징과 기술적 설명서에 자세히 소개된 이 예외들 때문에, 우리는 호스트 시스템에서 계산된 IEEE 754 값들과 다른 결과들을 얻을 수 있습니다.

중요한 차이 중 하나는 결합된 곱-합(fused multiply-add, FMA) 명령어입니다. 이 명령어는 곱셈과 덧셈을 한 명령어 실행으로 결합합니다. 이 명령어의 결과는 이따금 두 연산을 따로 한 결과와 살짝 다릅니다.

[6.3.5. x86 80-bit 계산 \(Computations\)](#)

x86 프로세서들은 부동 소수점 계산들이 수행될 때, 확장된 정밀도의 80 비트 double 수학(math)을 사용할 수 있습니다. 이 계산 결과들은 쿠다 디바이스에서 수행된 순수한 64 비트 연산들과 자주 다를 수 있습니다. 더 비슷한 값들을 얻기 위해, x86 호스트 프로세서는 정규(regular) double 또는 단정밀을 사용하도록 설정하십시오 (각각 64 비트 그리고 32 비트로). 이는 `FLDCW` x86 어셈블리 명령어 또는 그와 동등한 운영 체제 API 로 수행될 수 있습니다.

[7. 쿠다 응용 프로그램 최적화하기 \(Optimizing CUDA](#)

[Applications\)](#)

응용 프로그램 병렬화의 각 단계가 완료된 뒤에, 개발자는 성능 향상을 위해 구현 최적화 단계로 옮겨갈 수 있습니다. 고려될 수 있는 많은 가능한 최적화들이 있으므로, 그 응용 프로그램의 요구(needs)를 잘 이해하는 것이 최적화 절차가 부드럽게 진행되는 데 도움이 될 수 있습니다. 그러나, 전체 평가, 병렬화, 최적화, 배포와 마찬가지로, 프로그램 최적화도 (최적화를 위한 기회를 찾고, 그 최적화를 적용하여 시험하고, 얻은 속도 향상을 검증하고, 반복하는) 반복적 프로세스입니다. 이는 프로그래머가 좋은 속도 향상을 보기도 전에 수많은 모든 가능한 최적화 전략들을 기억하는 데 많은 시간을 쓸 필요가 없음을 뜻합니다. 대신, 그 전략들은 배우면서 점진적으로 적용될 수 있습니다.

최적화는 계산과 검치는 데이터 전송부터 부동 소수점 연산 시퀀스들의 미세 조정에 이르기까지 다양한 수준에 적용될 수 있습니다. 사용할 수 있는 프로파일링 도구들은 이 절차들을 안내하는 데 매우 유용합니다. 왜냐하면, 그 도구들은 개발자들이 다음으로 집중해서 최적화해야 할 일을 제안하고 이 안내서의 적절한 최적화 절을 가리키는 참고자료를 제공하기 때문입니다.

8. 성능 계량 (Performance Metrics)

쿠다 코드를 최적화하려 할 때, 1) 어떻게 성능을 정확히 측정하는지 아는 것과 2) 성능 측정에서 대역폭의 역할이 무엇인지 이해하는 것은 도움이 됩니다. 이 장은 CPU 타이머와 쿠다 이벤트를 사용하여 어떻게 정확히 성능을 측정하는지를 논의합니다. 그리고 어떻게 대역폭이 성능 계량에 영향을 끼치고 어떻게 대역폭이 만드는 저항을 줄일지에 대해 살펴봅니다.

8.1. 타이밍 (Timing)

쿠다 호출과 커널 실행 시간은 CPU 또는 GPU 타이머를 사용하여 측정될 수 있습니다. 이 절은 두 접근법의 기능, 이점, 그리고 함정을 다룹니다.

8.1.1. CPU 타이머 사용하기 (Using CPU Timers)

CPU 타이머는 쿠다 호출 또는 커널 실행에 걸린 시간을 측정하는 데 쓰일 수 있습니다. 다양한 CPU 타이밍 접근법들의 세부 사항들은 이 문서의 범위를 벗어납니다. 다만, 개발자들은 꼭 그 타이밍 호출이 제공하는 시간의 분해능을 알고 있어야 합니다.

CPU 타이머를 사용할 때, 많은 쿠다 API 함수들이 비동기적이라는 사실을 기억하는 것은 중요합니다. 즉, 그 함수들은 작업을 마치기 전에, 제어(control)를 호출한 CPU 스레드에 돌려줍니다. 모든 커널 실행들 그리고 이름에 `Async` 접미사를 가진 메모리 복사 함수들은 비동기적입니다. 따라서, 특정 호출 또는 순차적 쿠다 호출들의 경과 시간을 정확히 측정하기 위해, CPU 타이머를 시작하고 멈추기 바로 전에 `cudaDeviceSynchronize()`를 호출함으로써 CPU 스레드와 GPU를 동기화시킬 필요가 있습니다. `cudaDeviceSynchronize()`는 그 스레드에서 이전에 실행한 모든 쿠다 호출들이 완료될 때까지 CPU 스레드 호출을 차단합니다.

또한, GPU 상에서 CPU 스레드를 특정 스트림 또는 이벤트와 동기화시키는 것도 가능합니다. 그러나, 기본 스트림을 제외하고 이 동기화 함수들은 스트림에서 타이밍 코드를 위해서는

적절하지 않습니다. `cudaStreamSynchronize()`는 이전에 주어진 스트림으로 실행된 모든 쿠다 호출들이 완료될 때까지 CPU 스레드를 차단합니다. `cudaEventSynchronize()`는 한 특정 스트림에서 주어진 이벤트가 GPU 에 의해 기록될 때까지 (CPU 스레드를) 차단합니다. 드라이버가, 다른 디폴트가 아닌(other non-default) 스트림들의 쿠다 호출들을 실행에 끼워 넣을 수도 있으므로, 다른 스트림들에서의 호출들이 타이밍에 포함될 수도 있습니다.

디폴트 스트림(스트림 0)은 그 디바이스에서의 작업을 위해 직렬화 동작을 보이므로 (디폴트 스트림에 있는 한 연산은 어떤 스트림에 있는 모든 이전 호출들이 완료된 다음에만 시작할 수 있습니다. 그리고 어떤 스트림의 뒤따르는 연산도 그것을 끝마칠 때까지 시작할 수 없습니다.), 이 함수들은 디폴트 스트림에서의 타이밍을 위해 높은 신뢰도로 사용될 수 있습니다. 이 절에서 언급된 CPU 와 GPU 동기화 지점들은 GPU 의 처리 파이프라인에서 멈춤(stall)을 뜻합니다. 따라서 성능에 주는 영향을 줄이기 위해 드물게 사용되어야 합니다.

8.1.2. 쿠다 GPU 타이머 사용하기 (Using CUDA GPU Timers)

쿠다 이벤트 API 는 여러 호출들을 제공합니다. 그 호출들에는 이벤트를 만들고, 없애고, (타임스탬프를 통해) 이벤트를 기록하고, 타임스탬프들의 차이를 밀리초 단위의 부동 소수점 값으로 바꾸는 호출들이 포함됩니다. [쿠다 이벤트를 사용하여 어떻게 코드의 시간을 재는가](#)는 그 호출들의 사용법을 설명합니다.

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );
kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS);
cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

여기서 `cudaEventRecord()`는 `start` 와 `stop` 이벤트를 디폴트 스트림(스트림 0)에 배치하기 위해 사용됩니다. 그것이 그 스트림 안의 이벤트에 도착할 때, 디바이스는 그 이벤트를 위한 한 타임스탬프를 기록할 것입니다. `cudaEventElapsedTime()` 함수는 `start` 와 `stop` 이벤트의 기록 사이에 경과된 시간을 리턴합니다. 이 값은 밀리초로 표현되고 대략 0.5 마이크로초 정도의 분해능을 가집니다. 이 목록의 다른 호출과 같이, 그것들의 측정 연산, 파라미터들, 그리고 리턴 값들은 쿠다 도구 모음 참조 설명서(*CUDA Toolkit Reference Manual*)에 설명되어 있습니다. 유념하십시오. 타이밍들은 GPU 클럭으로 측정됩니다. 그래서 타이밍 분해능은 운영 체제와 독립적입니다.

8.2. 대역폭 (Bandwidth)

대역폭은 성능을 위한 가장 중요한 요소 중 하나입니다. 거의 모든 코드 고침은 어떻게 그 고침이 대역폭에 어떤 영향을 주는가의 맥락에서 수행되어야 합니다. 이 지침서의 [메모리 최적화](#)에서 설명되었듯, 대역폭은 데이터가 어느 메모리에 저장되는지, 어떻게 데이터가 그 메모리에 놓이는지, 그리고 그 데이터가 어떤 순서로 접근되는지 등에 따라 극적인 영향을 받을 수 있습니다.

성능을 정확하게 측정하기 위해, 이론적 대역폭과 실효 대역폭을 계산하는 것은 유용합니다. 실효 대역폭이 이론적 대역폭보다 훨씬 낮을 때, 설계 또는 구현 세부 사항들은 대역폭을 낭비하고 있을 가능성이 높습니다. 그리고 최적화의 다음 주 목표는 대역폭 사용량 증가가 되어야 합니다.

노트: 우선순위 높음: 성능과 최적화 이득을 측정할 때, 측정 기준으로 (계산의) 실효 대역폭을 사용하십시오.

8.2.1. 이론적 대역폭 계산 (Theoretical Bandwidth Calculation)

이론적 대역폭은 제품 인쇄물에 들어있는 하드웨어 명세서를 사용하여 계산될 수 있습니다. 예를 들어, 엔비디아 테슬라 M2090 은 클럭 레이트가 1.85 GHz 이고 384 비트 넓이 메모리 인터페이스를 가진 GDDR5 RAM 을 사용합니다.

이 데이터가 사용될 때, 엔비디아 테슬라 M2090 의 최대 이론적 메모리 대역폭은 177.6 GB/s 입니다.

$$(1.85 \times 10^9 \times (384 / 8) \times 2) \div 10^9 = 177.6 \text{ GB/s}$$

이 계산에서, 메모리 클럭 율은 Hz 로 바뀌고, 인터페이스 폭(비트를 바이트로 바꾸기 위해 8 로 나뉜)으로 곱해지고, 더블 데이터 레이트(double data rate, DDR) 때문에 2 로 곱해집니다. 마침내, 이 곱은 그 결과를 GB/s 로 바꾸기 위해 10^9 으로 나뉘집니다.

노트: 몇몇 계산들은 최종 계산을 위해 10^9 대신 1024^3 을 사용합니다. 그런 경우, 대역폭은 165.4GB/s 였을 것입니다. 그 비교가 마땅하려면 이론적 대역폭과 실효 대역폭을 계산할 때 같은 값으로 나누어야 합니다.

노트: ECC 가 쓰일 수 있을 때, 실효 최대 대역폭은 메모리 체크섬을 위한 추가적 트래픽 때문에 약 20% 정도 줄어듭니다. ECC 가 대역폭에 미치는 정확한 영향은 메모리 접근 패턴에 의존적입니다.

8.2.2. 실효 대역폭 계산 (Effective Bandwidth Calculation)

실효 대역폭은 타이밍 특정한 프로그램 활동들과 어떻게 그 프로그램이 데이터에 접근하는지로 계산됩니다. 그렇게 하기 위해, 이 등식을 사용하십시오.

$$\text{실효 대역폭} = ((B_r + B_w) \div 10^9) \div \text{시간}$$

여기서, 실효 대역폭의 단위는 GB/s 이고, B_r 은 커널 당 읽기 바이트 개수, B_w 는 커널 당 쓰기 바이트 개수, 그리고 시간은 초로 주어집니다.

예를 들어, 2048 x 2048 행렬 복사의 실효 대역폭을 계산하기 위해, 다음 공식이 사용될 수 있습니다.

$$\text{실효 대역폭} = ((2048^2 \times 4 \times 2) \div 10^9) \div \text{시간}$$

요소들의 개수는 각 요소의 크기로 곱해지고 (float 하나를 위해서는 4 바이트), (읽기와 쓰기 이므로) 2 로 곱해지고, (GB 로 바꾸기 위해) 10^9 (또는 1024^3)으로 나뉘집니다. 이 수는 GB/s 로 바꾸기 위해 초 단위의 시간으로 나뉘집니다.

8.2.3. 비주얼 프로파일러로 보고된 처리량 (Throughput Reported by Visual Profiler)

계산 능력 2.0 이상의 디바이스들에서, 비주얼 프로파일러는 몇 가지 다른 메모리 처리량들을 수집하기 위해 쓰일 수 있습니다. 다음 처리량 측정 기준들은 Details 또는 Detail Graphs view 에서 보여질 수 있습니다.

- Requested Global Load Throughput (요청된 글로벌 로드(불어오기) 처리량)
- Requested Global Store Throughput (요청된 글로벌 스토어(저장하기) 처리량)
- Global Load Throughput (글로벌 로드 처리량)
- Global Store Throughput (글로벌 스토어 처리량)
- DRAM Read Throughput (DRAM 읽기 처리량)
- DRAM Write Throughput (DRAM 쓰기 처리량)

요청된 글로벌 로드 처리량과 요청된 글로벌 스토어 처리량 값들은 그 커널에서 요청된 글로벌 메모리 처리량을 가리킵니다. 따라서 [실효 대역폭 계산](#)에서 소개된 방법에 따라 계산된 실효 대역폭에 해당됩니다.

메모리 처리의 최소 크기가 대부분 워드 크기보다 더 크기 때문에, 한 커널을 위해 요청된 실제 메모리 처리량은 그 커널에 의해 사용된 데이터의 전송을 포함할 수 있습니다. 글로벌 메모리 접근들을 위해, 이 실제 처리량은 글로벌 로드 처리량과 글로벌 스토어 처리량 값들로 보고됩니다.

두 수 모두 유용함을 유념하는 것은 중요합니다. 실제 메모리 처리량은 그 코드가 하드웨어 한계에 얼마나 가까운지를 보여줍니다. 그리고 실효 또는 요청된 대역폭을 실제 대역폭에 비교하는 것은 우리가 최적이지 아닌 응집된 메모리 접근에 의해 얼마나 대역폭이 버려지고 있는지를 추정할 수 있게 합니다 ([글로벌 메모리의 응집된 접근](#)을 보십시오). 글로벌 메모리 접근들을 위해, 요청된 메모리 대역폭 대 실제 메모리 대역폭의 이 비교는 글로벌 메모리 로드 효율성(Global Memory Load Efficiency)과 글로벌 메모리 스토어 효율성(Global Memory Store Efficiency) 측정 기준으로 보고됩니다.

9. 메모리 최적화

메모리 최적화는 좋은 성능을 얻기 위해 가장 중요한 분야입니다. 목표는 대역폭을 최대화함으로써 하드웨어 사용을 최대화하는 것입니다. 대역폭은 1) 빠른 메모리를 많이 사용함으로써 그리고 2) 느린 접근 속도를 가진 메모리를 가능한 한 적게 사용함으로써 가장 잘 얻어집니다. 이 장은 호스트와 디바이스에 있는 다양한 종류의 메모리와 데이터 항목들이 메모리를 효과적으로 사용하기 위해 얼마나 최적인지를 논의합니다.

9.1. 호스트와 디바이스 사이 데이터 전송

디바이스(글로벌) 메모리와 GPU 사이 최대 이론적 대역폭(이를테면 엔비디아 테슬라 M2090에서는 177.6 GB/s)은 호스트와 디바이스 사이 이론적 대역폭(PCIe x16 Gen 2에서 8

GB/s)보다 훨씬 높습니다. 따라서, 응용 프로그램에서 가장 좋은 성능을 얻기 위해, 호스트와 디바이스 사이 데이터 전송은 최소화되어야 합니다. 여기에는 심지어 GPU 에서 실행되는 커널들이 호스트 CPU 에서 그 커널들을 실행시키는 것보다 느린 경우도 포함됩니다.

노트: 우선순위 높음: 호스트와 디바이스 사이 데이터 전송을 최소화 하십시오. 여기에는 심지어 GPU 에서 실행되는 커널들이 호스트 CPU 에서 그 커널들을 실행시키는 것보다 느린 경우도 포함됩니다.

중간 자료 구조들은 꼭 1) 그 디바이스에서 실행되고 2) (한 번도 호스트에 의해 사상되거나 호스트 메모리로 복사됨 없이 제거되는) 디바이스 메모리에 만들어져야 합니다.

또한, 각 전송과 관련된 간접비용 때문에, 많은 작은 전송들을 한 큰 전송으로 묶는 것은 각 전송을 따로 하는 것보다 훨씬 더 좋게 동작합니다. 심지어 그렇게 하는 것이 메모리의 연속적이지 않은 영역들을 한 연속적 버퍼로 패킹하여 전송한 다음 그것들을 다시 언패킹 하는 경우에도 그렇습니다.

마지막으로, 쿠다 C 프로그래밍 가이드와 이 문서의 핀으로 고정된 메모리([Pinned Memory](#)) 절에서 논의되었듯, 호스트와 디바이스 사이 더 높은 대역폭은 page-locked (또는 pinned) 메모리가 사용될 때 성취됩니다.

9.1.1. 핀으로 고정된 (Pinned) 메모리

Page-locked 또는 핀으로 고정된 메모리 전송들은 호스트와 디바이스 사이에서 가장 높은 대역폭을 얻습니다. 예를 들어, PCIe x16 Gen 2 카드들에서, 핀으로 고정된 메모리는 대략 6 GB/s 의 전송률을 얻을 수 있습니다.

핀으로 고정된 메모리는 런타임 API 에 있는 `cudaHostAlloc()` 함수들을 사용하여 할당됩니다. `bandwidthTest` 쿠다 샘플은 어떻게 이 함수들을 사용하는지 그리고 어떻게 메모리 전송 성능을 측정하는지를 보여줍니다.

미리 할당된 시스템 메모리의 영역들을 위해, 사용 중인 메모리를 핀으로 고정하는 데 `cudaHostRegister()`가 사용될 수 있습니다. 이 함수에서, 별도의 버퍼를 할당하여 데이터를 그곳으로 복사할 필요는 없습니다.

핀으로 고정된 메모리는 남용되지 않아야 합니다. 과도한 사용은 전체 시스템 성능을 떨어뜨릴 수 있습니다. 핀으로 고정된 메모리는 부족한 자원이기 때문입니다. 그러나 얼마나 많은 것이 너무 많은 것인지는 미리 알기 어렵습니다. 게다가, 시스템 메모리를

핀으로 고정하는 연산은 대부분의 보통 시스템 메모리 할당보다 무겁습니다. 모든 최적화들과 마찬가지로, 최적 성능 파라미터들을 위해, 응용 프로그램과 시스템의 시험이 계속됩니다.

9.1.2. 계산과 함께 비동기 그리고 중첩된 전송 (Asynchronous and Overlapping Transfers with Computation)

`cudaMemcpy()`를 사용한 호스트와 디바이스 사이 데이터 전송들은 블로킹 전송들입니다. 즉, 데이터 전송이 끝난 뒤에만 제어가 호스트 스레드로 리턴됩니다. `cudaMemcpyAsync()` 함수는 `cudaMemcpy()`의 논블로킹 변형입니다. 여기에서 제어는 즉시 호스트 스레드로 리턴됩니다. `cudaMemcpy()`과 대조적으로, 비동기 전송 버전은 핀으로 고정된 호스트 메모리를 요구합니다 ([Pinned Memory](#) 를 보십시오). 그리고 그 함수는 추가 인자, 스트림 ID, 하나를 포함합니다. 스트림은 디바이스에서 차례대로 실행되는 연산들의 시퀀스입니다. 다른 스트림들에서 연산들은 서로 맞물려지고 몇몇 경우들에서는 겹쳐질 수 있습니다. 이는 호스트와 디바이스 사이 데이터 전송을 숨기기 위해 사용되는 속성입니다.

비동기 전송들은 데이터 전송들과 계산의 겹침을 두 가지 다른 방식으로 가능하게 합니다. 모든 쿠다 가능 디바이스들에서, 호스트 계산을 비동기 데이터 전송들 및 디바이스 계산들과 겹치는 것이 가능합니다. 예를 들어, [계산과 데이터 전송들의 겹침](#)은 데이터가 디바이스로 전송되고 디바이스를 사용하는 한 커널이 실행되는 동안 루틴 `cpuFunction()`에서 어떻게 호스트 계산이 수행되는지를 시연합니다.

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

`cudaMemcpyAsync()` 함수의 마지막 인자는 스트림 ID 입니다. 이 경우, 그 ID 는 기본 스트림인 스트림 0 을 사용합니다. 커널 또한 기본 스트림을 사용합니다. 그리고 그 기본 스트림은 메모리 복사가 완료될 때까지 실행을 시작하지 않을 것입니다. 메모리 복사와 커널 모두 제어를 호스트로 즉시 리턴하므로, 호스트 함수 `cpuFunction()`은 그것들의 실행을 겹칩니다.

[계산과 데이터 전송들의 겹침](#)에서, 메모리 복사와 커널 실행은 순차적으로 발생합니다. 동시 복사와 실행이 가능한 디바이스들에서, 디바이스에서의 커널 실행과 호스트와 디바이스 사이 데이터 전송을 겹치는 것이 가능합니다. 디바이스가 이 능력을 가지고 있는지는 `cudaDeviceProp` 구조체의 `asyncEngineCount` 필드를 봄으로써 알 수 있습니다

(또는 `deviceQuery` 쿠다 샘플의 출력된 목록을 봄으로써 알 수 있습니다). 이 능력을 가진 디바이스들에서, 그 겹침은 다시 한 번 핀으로 고정된 호스트 메모리를 요구합니다. 그리고 추가로, 그 데이터 전송과 커널은 반드시 다른 디폴트가 아닌 스트림들을 사용해야만 합니다 (0 이 아닌 스트림 ID 들을 가진 스트림들). 즉, 디폴트가 아닌 스트림들이 이 겹침을 위해 요구됩니다. 왜냐하면 메모리 복사, 메모리 설정 함수들, 그리고 디폴트 스트림을 사용하는 커널 호출들은 디바이스 상의 모든 이전 호출들이 (어떤 스트림에서든) 완료된 다음에만 시작하기 때문입니다. 그리고 디바이스 상의 연산도 그것들이 끝날 때까지 (어떤 스트림에서도) 시작하지 않습니다.

[동시 복사와 실행](#)은 그 기본 기법을 묘사합니다.

Concurrent copy and execute

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(otherData_d);
```

이 코드에서, 데이터 전송과 커널 실행에서, 두 스트림이 `cudaMemcpyAsync` 호출과 그 커널의 실행 구성의 마지막 인자들에서 특정된 대로 만들어져 사용됩니다.

[동시 복사와 실행](#)은 어떻게 커널 실행을 비동기 데이터 전송과 겹칠지를 시연합니다. 이 기법은 데이터가 작은 덩어리들로 나뉘어 여러 단계로 전송될 수 있는 데이터 의존성이 있을 때 사용될 수 있습니다. 그리고 각 작은 덩어리가 도착할 때, 그 여러 개 커널들을 실행합니다. [순차적 복사와 실행](#) 그리고 [단계적 동시 복사와 실행](#)은 이것을 시연합니다. 두 예제는 같은 결과를 만듭니다. 첫 부분은 참조 순차적 구현을 보입니다. 그 구현은 float 형 N 요소로 구성된 배열을 전송하고 연산합니다 (여기서 N 은 `nThreads`로 고르게 나눌 수 있다고 가정됩니다).

Sequential copy and execute

```
cudaMemcpy(a_d, a_h, N*sizeof(float), dir);
```

```
kernel<<<N/nThreads, nThreads>>>(a_d);
```

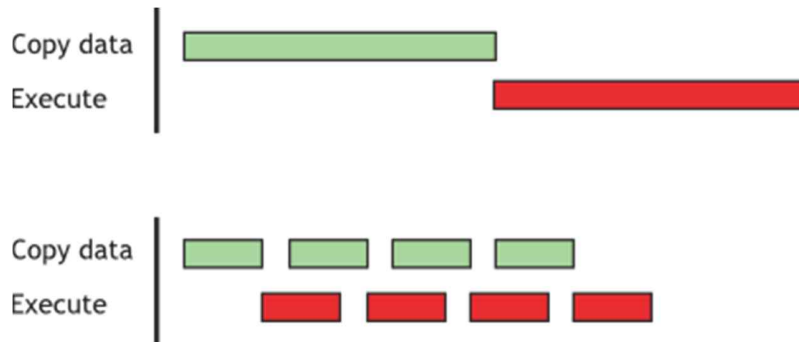
[단계적 동시 복사와 실행](#)은 어떻게 전송과 커널 실행이 nStream 단계들로 나뉠 수 있는지를 보여줍니다. 이 접근법은 약간의 데이터 전송과 실행의 겹침을 허용합니다.

Staged concurrent copy and execute

```
size=N*sizeof(float)/nStreams;
for (i=0; i<nStreams; i++) {
    offset = i*N/nStreams;
    cudaMemcpyAsync(a_d+offset, a_h+offset, size, dir, stream[i]);
    kernel<<<N/(nThreads*nStreams), nThreads, 0, stream[i]>>>(a_d+offset);
}
```

(단계적 동시 복사와 실행에서, N 은 `nThreads*nStreams` 로 고르게 나뉠 수 있다고 가정됩니다.) 한 스트림 안에서는 실행이 순차적으로 수행되므로, 각 스트림들 안에서 데이터 전송들이 완료될때까지는 어떤 커널도 시작하지 않을 것입니다. 현재 GPU 들은 비동기 데이터 전송과 커널 실행을 동시에 처리할 수 있습니다. 복사 엔진 하나를 가진 GPU 들은 비동기 데이터 전송 하나와 커널들의 실행을 수행할 수 있습니다. 반면, 두 개의 복사 엔진을 가진 GPU 들은 호스트에서 디바이스로 비동기 데이터 전송 하나, 디바이스에서 호스트로의 비동기 데이터 전송 하나, 그리고 커널들의 실행을 동시에 수행할 수 있습니다. 한 GPU 에 있는 복사 엔진들의 개수는 `cudaDeviceProp` 구조체의 `asyncEngineCount` 필드에 주어집니다. 또한, `deviceQuery` 쿠다 샘플의 출력 목록에도 있습니다. (비동기 전송과 블로킹 전송을 겹치는 것은 불가능하다는 점이 꼭 언급되어야 합니다. 왜냐하면, 블로킹 전송은 디폴트 스트림에서 발생하기 때문입니다. 그래서 블로킹 전송은 모든 쿠다 호출들이 완료될 때까지 시작될 수 없습니다. 블로킹 전송은 어떤 다른 쿠다 호출도 그 전송이 완료될 때까지 시작하지 못하도록 할 것입니다.) [그림 1](#) 은 두 짧은 코드 실행의 타임라인을 묘사합니다. 그리고 그 그림의 절반 아래에 있는 [단계적 동시 복사와 실행](#)을 위한 `nStreams` 는 4 입니다.

그림 1. 복사와 커널 실행을 위한 타임라인 비교



위 : 순차적 (Sequential), 아래 : 동시 (Concurrent)

이 예제에서, 데이터 전송과 커널 실행 시간들이 비교될 수 있다고 가정되었습니다. 그런 경우, 그리고 실행 시간(t_E)이 전송 시간(t_T)을 초과할 때, 전체 시간의 대략적 추정은 단계적 버전을 위해 $t_E + t_T / nStreams$ 이고 순차적 버전을 위해서는 $t_E + t_T$ 입니다. 만약 전송 시간이 실행 시간을 초과하면, 전체 시간을 위한 대략적 추정은 $t_T + t_E / nStreams$ 입니다.

9.1.3. 제로 복사 (Zero Copy)

제로 복사는 쿠다 도구모음 버전 2.2 에 추가된 한 특징입니다. 제로 복사는 GPU 스레드들이 바로 호스트 메모리에 접근하는 것을 가능케 합니다. 이 목적을 위해, 제로 카피는 사상된 핀으로 고정된 (pinned 또는 non-pageable) 메모리를 요구합니다. 통합된 GPU 들에서 (다시 말해, 쿠다 디바이스 속성들 구조체의 integrated 필드가 1 로 설정된 GPU 들에서), 사상된 핀으로 고정된 메모리는 항상 성능 이득이 있습니다. 왜냐하면, 통합된 GPU 와 CPU 메모리가 물리적으로 같으므로, 제로 카피는 불필요한 복사들을 피하기 때문입니다. 따로 떨어진 GPU 들에서, 사상된 핀으로 고정된 메모리는 오직 특정 경우들에서만 도움이 됩니다. 그 데이터가 GPU 에 캐시되지 않으므로, 사상된 핀으로 고정된 메모리는 오직 한 번 읽히거나 쓰여야 합니다. 그리고 그 메모리를 읽고 쓰는 그 글로벌 로드들과 스토어들은 응집되어야 합니다. 제로 복사는 스트림들을 대신해 사용될 수 있습니다. 왜냐하면, 커널에서 생긴 데이터 전송들이 (설정하고 스트림들의 최적 개수를 결정하는 간접비용 없이) 자동으로 커널 실행과 겹치기 때문입니다.

노트: 우선순위 낮음: 쿠다 도구모음 2.2 이상을 위한 통합된 GPU 들에서는 제로 복사(zero-copy) 연산을 사용하십시오.

[제로 복사 호스트 코드](#)에 있는 호스트 코드는 전형적으로 어떻게 제로 복사가 준비되는지 보여줍니다.

Zero-copy host code

```
float *a_h, *a_map;
...
cudaGetDeviceProperties(&prop, 0);
if (!prop.canMapHostMemory)
    exit(0);
cudaSetDeviceFlags(cudaDeviceMapHost);
cudaHostAlloc(&a_h, nBytes, cudaHostAllocMapped);
cudaHostGetDevicePointer(&a_map, a_h, 0);
kernel<<<gridSize, blockSize>>>(a_map);
```

이 코드에서, `cudaGetDeviceProperties()` 함수에 의해 리턴되는 구조체의 `canMapHostMemory` 필드는 그 디바이스가 호스트 메모리를 디바이스의 주소 공간으로 사상하는 것을 지원하는지 확인하기 위해 사용됩니다. Page-locked 메모리 사상은 `cudaDeviceMapHost`를 인자로 가진 `cudaSetDeviceFlags()`을 호출함으로써 활성화됩니다. 유념하십시오. `cudaSetDeviceFlags()`는 반드시 디바이스를 설정하기 전에 또는 상태(state)를 요구하는 쿠다 호출을 만들기 전에 (즉, 근본적으로, 한 문맥이 만들어지기 전에) 호출되어야 합니다. 사상된 page-locked 호스트 메모리는 `cudaHostAlloc()`를 사용하여 할당됩니다. 그리고 함수 `cudaHostGetDevicePointer()`을 통해 사상된 디바이스 주소 공간을 가리키는 포인터가 얻어집니다. 제로 복사 호스트 코드에 있는 코드에서, `kernel()`은 그 사상된 핀으로 고정된 호스트 메모리를 포인터 `a_map`을 사용하여 참조할 수 있습니다. 만약 `a_map`이 디바이스 메모리의 한 위치를 가리켰더라도, 정확히 같은 동작이 수행되었을 것입니다.

노트: 사상된 핀으로 고정된 호스트 메모리는, 쿠다 스트림들의 사용을 피하면서, 계산과 CPU-GPU 메모리 전송들을 동시에 할 수 있게 합니다. 그러나 그런 메모리 영역들로의 반복된 접근은 반복된 PCIe 전송들을 유발하므로, 이전에 읽은 호스트 메모리 데이터를 수동으로 캐시하기 위해, 디바이스 메모리에 두 번째 영역을 만드는 것을 고려하십시오.

[9.1.4. 통합된 가상 번지 지정 \(Unified Virtual Addressing\)](#)

TCC 드라이버 모드가 사용될 때, 계산 능력 2.0 이상의 디바이스들은 64 비트 리눅스, 맥 OS, 그리고 윈도우즈 XP 와 윈도우즈 비스타/7 에서 통합된 가상 어드레싱(Unified Virtual Addressing, UVA)이라 불리는 특별한 어드레싱 모드를 지원합니다. UVA 를 가지고, 호스트 메모리와 모든 설치된 (UVA 를) 지원하는 디바이스들의 디바이스 메모리들은 하나의 단일 가상 주소 공간을 공유합니다.

UVA 전에, 응용 프로그램은 어느 포인터들이 디바이스 메모리를 가리키는지 (그리고 어느 디바이스를 위한 것인지) 그리고 어느 것이 분리된 비트의 메타데이터로 [또는 그 프로그램의 직접 코딩된(hard-coded) 정보로] 각 포인터를 위한 호스트 메모리를 가리키는지 계속 추적해야 했습니다. 반면, UVA 를 사용하면, 포인터가 가리키고 있는 물리 메모리 공간은 `cudaPointerGetAttributes()`을 사용해 단순히 그 포인터의 그 값을 조사함으로써 결정될 수 있습니다.

UVA 에서, `cudaHostAlloc()`으로 할당된 핀으로 고정된 호스트 메모리는 동일한 호스트와 디바이스 포인터들을 가질 것입니다. 그래서 그런 할당들을 위해 `cudaHostGetDevicePointer()`를 호출할 필요가 없습니다. 그러나, `cudaHostRegister()`를 통해 사후에 핀으로 고정된 호스트 메모리 할당들은 호스트 포인터들과 계속 다른 디바이스 포인터들을 가질 것입니다. 그래서 그 경우, `cudaHostGetDevicePointer()`는 여전히 필요합니다.

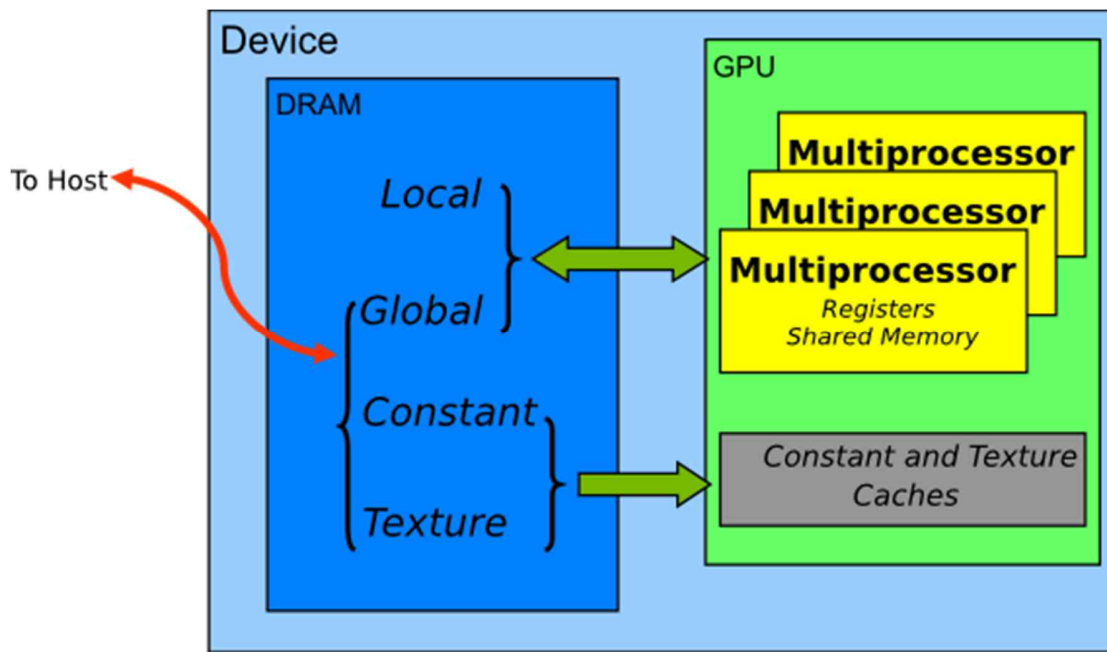
UVA 는 또한 (지원된 구성들에서) 지원되는 GPU 들을 위해 PCIe 버스를 통해 (호스트 메모리를 우회하여) 데이터의 peer-to-peer (P2P) 전송을 가능케 하는 데 필요한 전제 조건입니다.

UVA 와 P2P 를 위한 더 자세한 설명과 소프트웨어 요구들은 CUDA C 프로그래밍 가이드를 참고하십시오.

9.2. 디바이스 메모리 공간 (Device Memory Spaces)

쿠다 디바이스들은 여러 메모리 공간을 사용합니다. 그 공간은 서로 다른 특성을 가집니다. 그 특성은 CUDA 응용 프로그램에서 다른 용도로 사용됩니다. [그림 2](#) 에서 보여지듯, 그 메모리 공간들에는 글로벌(global), 로컬(local), 공유(shared), 텍스처(texture), 그리고 레지스터(registers)가 포함됩니다.

그림 2. 한 쿠다 디바이스의 메모리 공간들



이 메모리 공간들 중, 글로벌 메모리는 가장 풍부합니다. 각 [계산 능력](#) 별 각 메모리 공간에서 사용할 수 있는 메모리 양을 보시려면, 쿠다 C 프로그래밍 가이드의 특징과 기술 사양들(Features and Technical Specifications)을 보십시오. 글로벌, 로컬, 그리고 텍스처 메모리는 가장 큰 접근 지연 시간을 가집니다. 상수(constant) 메모리, 공유 메모리, 그리고 레지스터 파일은 그 뒤를 따릅니다.

메모리 타입들의 다양한 주요 특성들은 [표 1](#)에서 보여집니다.

메모리	위치 on/off chip	캐시됨	접근	범위	수명 (Lifetime)
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes † †	R/W	1 thread	Thread

Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† 계산 능력 2.x 의 디바이스들에서 기본으로 L1 과 L2 로 캐시됨; 비록 몇몇은 컴파일링 프래그들을 통해 L1 으로 캐시되도록 선택할 수 있으나, 더 높은 계산 능력들에서는 기본으로 L2 로만 캐시됨.					
†† 계산 능력 2.x 와 3.x 디바이스들에서 기본으로 L1 과 L2 로 캐시됨. 계산 능력 5.x 의 디바이스들은 로컬들을 오직 L2 로만 캐시함.					

텍스처 접근의 경우, 만약 텍스처 참조가 글로벌 메모리에 있는 한 선형 배열로 한정되면, 그 디바이스 코드는 그 기저의(underlying) 배열에 쓸 수 있습니다. 쿠다 배열들로 한정되는 텍스처 참조들은 서피스(surface)를 같은 기저의 쿠다 배열 스토리지(storage)에 묶음(binding)으로써 surface-write 연산들을 통해 쓰여질 수 있습니다. 같은 커널 실행에서 그 기저 글로벌 메모리 배열로 쓰는 동안, 텍스처에서 읽는 것은 꼭 피해져야 합니다. 왜냐하면, 텍스처 캐시들은 읽기 전용이고 관련된 글로벌 메모리가 수정될 때도 무효화되지(invalidated) 않기 때문입니다.

9.2.1. 글로벌 메모리로 응집된 접근 (Coalesced Access to Global Memory)

아마 쿠다 가능 GPU 구조들을 위한 프로그래밍에서, 성능에 영향을 미치는 가장 중요한 요소는 글로벌 메모리 접근의 응집화일 것입니다. 특정 접근 요구들을 맞출 때, 한 워프

안에 있는 스레들에 의한 글로벌 메모리 로드와 스토어는 디바이스에 의해 한 트랜잭션(transaction)만큼으로 작게 응집됩니다.

노트: 우선순위 높음: 언제든지 가능하면 글로벌 메모리 접근들을 응집하십시오.

그 응집된 접근을 위한 요구들은 쿠다 C 프로그래밍 가이드에 문서화되어 있습니다. 그 요구들은 디바이스의 계산 능력마다 다를 수도 있습니다.

계산 능력 2.x 의 디바이스에서, 그 요구들은 꽤 쉽게 요약될 수 있습니다. 한 워프 안에 있는 스레드들의 동시 접근들은 많은 수의 트랜잭션들을 그 워프 안에 있는 모든 스레드들에 제공하는 데 필요한 수 만큼의 캐시 라인들로 응집할 것입니다. 기본적으로, 모든 접근들은 L1 을 통해 128 바이트 라인들로 캐시됩니다. 흩어진(scattered) 접근 패턴들에서, 너무 많이 가져오는 것(overfetch)을 줄이기 위해, 때때로 L2 에만 캐시하는 것이 유용합니다. L2 는 더 짧은 32 바이트 세그먼트들로 캐시합니다 (쿠다 C 프로그래밍 가이드를 참고하십시오).

계산 능력 3.x 의 디바이스들에서, 글로벌 메모리로 향하는 접근들은 오직 L2 에 캐시됩니다. L1 은 로컬 메모리 접근들을 위해 예약됩니다. 계산 능력 3.5, 3.7, 또는 5.2 의 디바이스들은 글로벌들을 L1 으로도 캐싱하도록 선택할 수 있습니다.

응집된 방식으로 메모리에 접근하는 것은 ECC 가 켜졌을 때 훨씬 더 중요합니다. 흩어진 접근들은 ECC 메모리 전송 간접비용을 증가시킵니다. 특히 글로벌 메모리로 데이터를 쓸 때 그렇습니다.

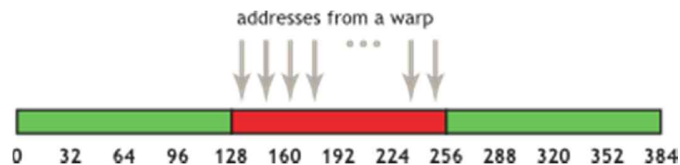
응집의 개념들은 다음의 단순한 예제들로 설명됩니다. 이 예제들에서는 계산 능력 2.x 의 디바이스가 사용된다고 가정됩니다. 이 예제들은 접근들이 L1 을 통해 캐시된다고 가정합니다. (이는 그 디바이스들의 기본 동작입니다.) 그리고 그렇지 않다고 언급되지 않는 한, 접근들은 4 바이트 워드라고 가정됩니다.

9.2.1.1. 단순 접근 패턴 (A Simple Access Pattern)

응집의 가장 간단한 경우는 어떤 쿠다 가능 디바이스에서든 성취될 것입니다. 즉, 한 캐시 라인 안에서 k 번째 스레드가 k 번째 워드에 접근하는 것입니다. 모든 스레드들이 참여할 필요는 없습니다.

예를 들어, 한 워프 안에 있는 스레드들이 인접한 4 바이트 워드들에 접근하면 (예를 들어, 인접한 float 값들), 한 128B L1 캐시 라인이 제공될 것이고, 따라서 한 응집된 트랜잭션이 그 메모리 접근에 대응할 것입니다. [그림 3](#)은 그런 한 패턴을 보여줍니다.

그림 3. 응집된 접근 - 모든 스레드들이 한 캐시 라인에 접근



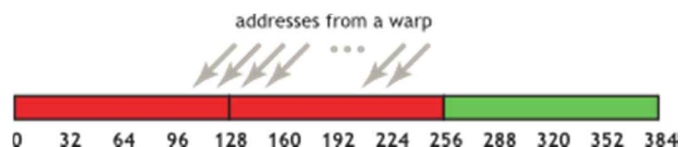
이 접근 패턴은 빨간색 사각형으로 표시된 단일 128B L1 트랜잭션을 만듭니다.

만약 그 라인의 몇몇 워드들이 어떤 스레드에서도 요청되지 않았더라도 (만약 몇몇 스레드들이 같은 워드에 접근했거나 만약 몇몇 스레드들이 그 접근에 참여하지 않았으면), 그 캐시 라인 안의 모든 데이터는 어쨌든 불려옵니다. 게다가, 만약 그 워프 안에 있는 스레드들에 의한 접근들이 이 세그먼트 안에서 순서가 바뀌었더라도, 여전히 한 128B L1 트랜잭션만 처리되었을 것입니다 (계산 능력 2.x 을 가진 한 디바이스의 경우).

[9.2.1.2. 순차적이거나 정렬되지 않은 접근 패턴 \(A Sequential but Misaligned Access Pattern\)](#)

만약 한 워프 안의 순차적 스레드들이 순차적이긴 하지만 캐시 라인 크기로 정렬되지 않은 메모리에 접근하면, 128B L1 캐시가 두 개 요청될 것입니다. [그림 4](#)는 이를 보여줍니다.

그림 4. 두 개의 128 바이트 L1 캐시 라인들로 접근하는 정렬되지 않은 순차적 어드레스들



캐시되지 않는 트래잭션들에서도 (다시 말해, L1 을 우회하고 L2 캐시만 사용하는 것들), L2 세그먼트들의 크기가 32 바이트라는 점을 제외하고, 비슷한 효과가 나타납니다. [그림 5](#) 는 그 한 예를 보여줍니다. 그림 4 와 같은 접근 패턴이 사용됩니다. 그러나 이제 L1 캐싱이 비활성화됩니다. 그래서 이제 다섯 개 32 바이트 L2 세그먼트들이 (그 요구를 만족시키기 위해) 필요해집니다.

그림 5. 다섯 개 32 바이트 L2 캐시 세그먼트들로 접근하는 정렬되지 않은 순차적 어드레스들



`cudaMalloc()`과 같은, 쿠다 런타임 API 를 통해 할당된 메모리는 최소 256B 로 정렬됨이 보장됩니다. 그러므로, 스레드 블록 크기를 워프 크기(다시 말해, 현재 GPU 들에서는 32)의 배수와 같이 합리적으로 선택하는 것은 캐시 라인들에 정렬된 워프들에 의한 메모리 접근들을 용이하게 합니다. (예를 들어, 만약 스레드 블록 크기가 워프 크기의 배수가 아니었으면, 두 번째, 세 번째, 그리고 이후 스레드 블록들에 의해 접근되는 메모리 어드레스들에 무슨 일이 일어났을지 고려해보십시오.)

9.2.1.3. 정렬되지 않은 접근의 효과 (Effects of Misaligned Accesses)

단순한 복사 커널을 사용하여 정렬되지 않은 접근의 영향을 관찰하는 것은 쉽고 유익합니다. [정렬되지 않은 접근들을 설명하는 한 복사 커널](#)은 그 예를 보여줍니다.

A copy kernel that illustrates misaligned accesses

```
__global__ void offsetCopy(float *odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
```

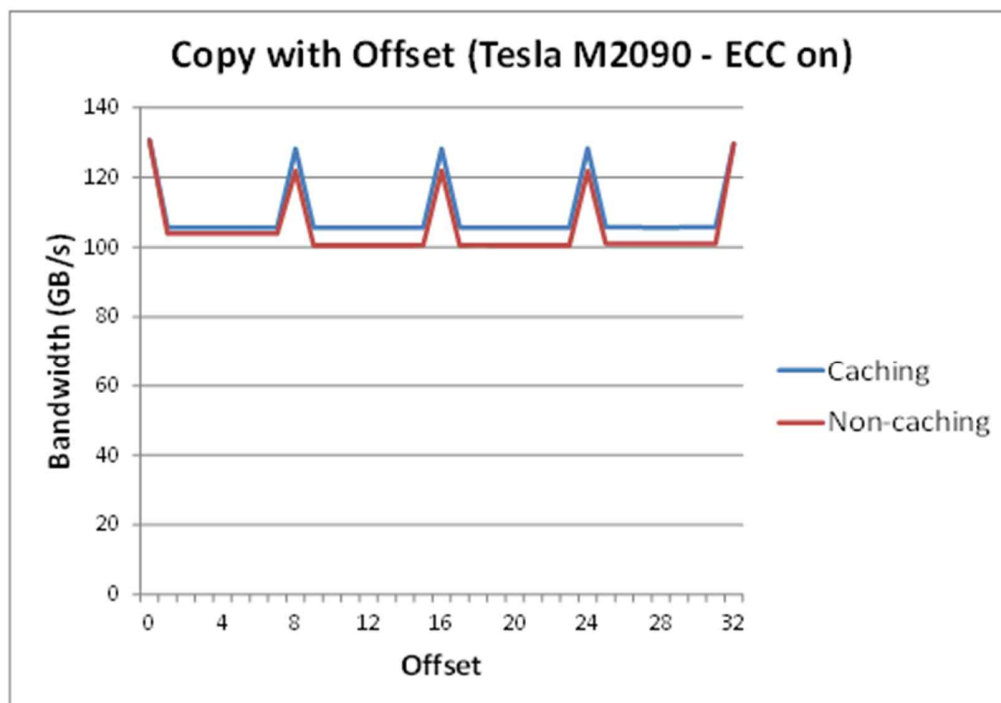
```

    odata[xid] = idata[xid];
}

```

[정렬되지 않은 접근들을 설명하는 한 복사 커널](#)에서, 데이터는 입력 배열 `idata` 에서 출력 배열 `odata` 로 복사됩니다. 두 배열은 모두 글로벌 메모리에 있습니다. 그 커널은 파라미터 `offset` 을 0 에서 32 까지 바꾸며 호스트 코드의 한 루프 안에서 실행됩니다. (그림 4 와 그림 5 는 각각 캐싱과 논캐싱 메모리 접근에 대응됩니다). (기본으로 ECC 가 켜진 계산 능력 2.0) 엔비디아 테슬라 M2090 에서, `offset` 바뀔때 따른 복사를 위한 실패 대역폭이 [그림 6](#)에 보여집니다.

그림 6. *offsetCopy* 커널의 성능



엔비디아 테슬라 M2090 에서, 오프셋 없는 또는 32 워드들의 배수인 오프셋들을 가진 글로벌 메모리 접근들은 한 개의 단일 L1 캐시 라인 트랜잭션 또는 4 개의 L2 캐시 세그먼트 로드들을 발생시킵니다 (비 L1 캐싱 로드들을 위해). 성취된 대역폭은 약

130GB/s 입니다. 그럴지 않으면, 두 개의 L1 캐시 라인들 (캐싱 모드) 또는 4~5 개 L2 캐시 세그먼트들(논캐싱 모드)이 워프 당 로드됩니다. 이는 오프셋 없는 메모리 처리량의 약 4/5 입니다.

흥미로운 점은 이 예제에서 우리는 캐싱하는 경우가 논캐싱 경우보다 더 나쁘게 동작할 거라고 기대했을 수도 있다는 점입니다. 캐싱하는 경우에서, 각 워프는 그것이 요구한 것의 두 배만큼 많은 바이트를 가져오기 때문입니다. 반면 논캐싱 경우에는 요구된 것보다 오직 5/4 만큼 더 많은 바이트를 워프 당 가져옵니다. 그러나, 이 특정한 예제에서, 그 효과는 분명하지 않습니다. 인접한 워프들이 그들의 이웃들이 가져온 그 캐시 라인들을 재사용하기 때문입니다. 비록 캐싱 로드들의 경우에서 그 영향은 여전히 분명하지만, 우리가 기대한만큼 크지는 않습니다. 만약 인접한 워프들이 그 오버패치된 캐시 라인들을 높은 정도로 재사용하지 않았다면, 더 그랬을 것입니다.

9.2.1.4. 건너뛰며 접근 (Strided Accesses)

위에 보여졌듯, 정렬되지 않은 순차적 접근들의 경우에서, [계산 능력](#) 2.x 의 디바이스들의 캐시들은 합리적인 성능을 얻는 데 많은 도움이 됩니다. 그러나, 그것은 1 씩 건너뛰지 않는(non-unit-strided) 접근과 다를 수도 있습니다. 그리고 이는 다차원 또는 행렬들을 다룰 때 자주 발생하는 패턴입니다. 이런 이유로, 가져온 각 캐시 라인 안에 가능한 많은 데이터를 담는 것은 디바이스들에서 메모리 접근들의 성능 최적화를 위해 중요합니다.

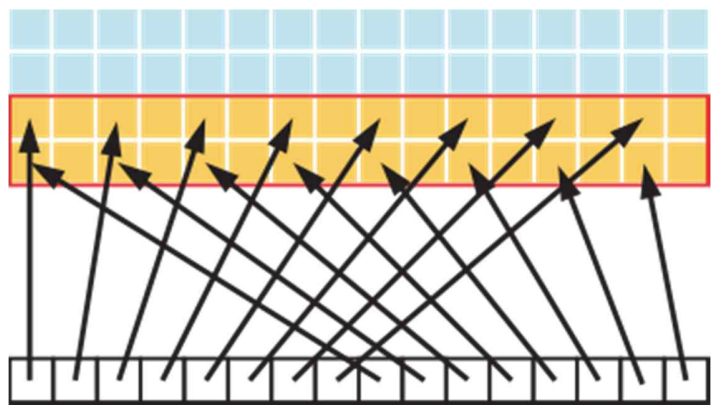
건너뛰는 접근이 실효 대역폭에 미치는 효과를 설명하기 위해, [논유닛 스트라이드 데이터 복사를 설명하기 위한 커널](#) 안의 strideCopy() 커널을 보십시오. 그 커널은 스레드들 사이 스트라이드 요소들의 한 스트라이드를 가지고 데이터를 idata 에서 odata 로 복사합니다.

A kernel to illustrate non-unit stride data copy

```
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

[그림 7](#)은 그런 상황을 설명합니다. 이 경우, 한 워프 안의 스레드들은 스트라이드 2를 가진 메모리의 워드들에 접근합니다. (계산 능력 2.0) 테슬라 M2090에서, 이 동작은 워프당 두 개의 L1 캐시 라인들(또는 논캐싱 모드에서 여덟 개의 L2 캐시 세그먼트들)의 로드한 번을 유발합니다.

그림 7. 2씩 건너 뛰며 메모리에 접근하는 인접한 스레드들



2 스트라이드(건너 뛰기)는 50%의 로드/스토어 효율을 초래합니다. 왜냐하면 트랜잭션 안의 요소들 중 절반이 사용되지 않기 때문입니다. 이는 대역폭이 낭비되었음을 뜻합니다. 스트라이드가 증가할수록, 실효 대역폭은 한 워프 안에 있는 32 개 스레드들을 위해 32 개 캐시 라인이 로드되는 지점까지 감소합니다. [그림 8](#)은 이를 보여줍니다.

그림 8. *strideCopy* 커널의 성능

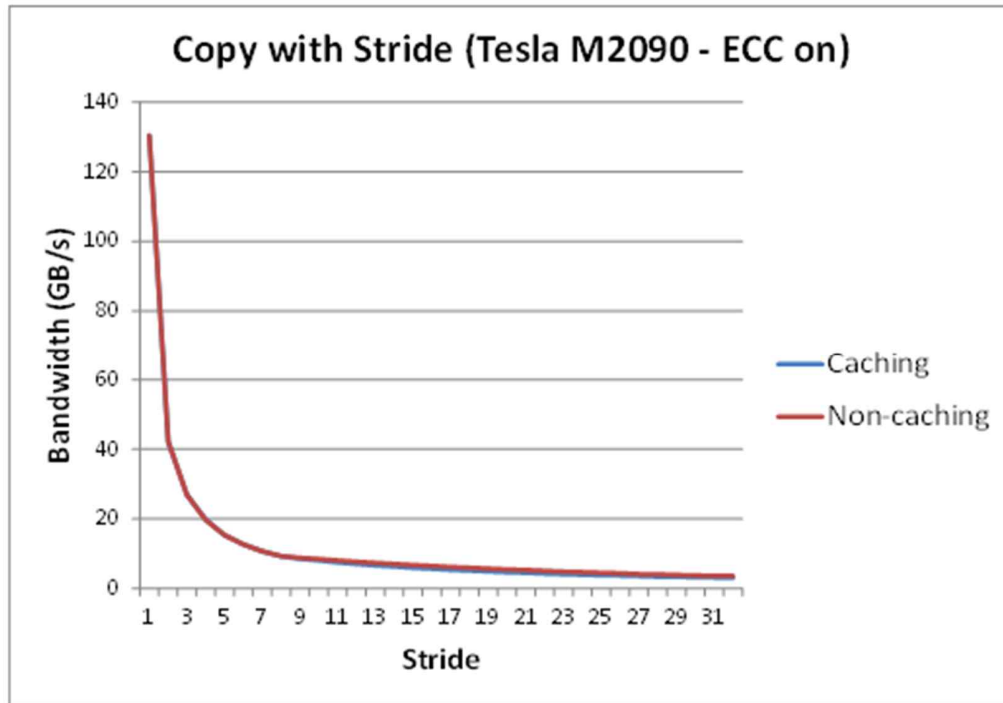


그림 8 에서 설명되었듯, 유닛 스트라이드가 아닌 글로벌 메모리 접근들은 언제나 가능하면 피해야 합니다. 그렇게 하기 위한 한 가지 방법은 공유 메모리를 활용하는 것입니다. 공유 메모리는 다음 절에서 설명됩니다.

9.2.2. 공유 메모리 (Shared Memory)

온칩이므로, 공유 메모리는 훨씬 더 높은 대역폭을 가집니다. 그리고 로컬과 글로벌 메모리보다 더 낮은 호출 시간(latency)을 가집니다. 다음 절에서 자세히 설명되겠지만, 이는 스레드들 사이에 뱅크 충돌들이 없다고 가정된 경우입니다.

9.2.2.1. 공유 메모리와 메모리 뱅크 (Bank)

동시 접근들을 위한 높은 메모리 대역폭을 얻기 위해, 공유 메모리는 같은 크기의 메모리 모듈들인 뱅크들로 나뉩니다. 그 모듈들은 동시에 접근될 수 있습니다. 그러므로, n 개 주소들의 n 개의 개별 메모리 뱅크들에 걸친 메모리 로드 또는 스토어가 동시에 수행될 수 있습니다. 이는 단일 뱅크 대역폭보다 n 배 더 높은 실효 대역폭을 만듭니다.

그러나, 만약 한 메모리 요청에서 여러 주소들이 같은 메모리 뱅크로 사상되면, 그 접근들은 직렬화됩니다. 하드웨어는 뱅크 충돌들을 가진 메모리 요청을 필요에 따라 충돌 없는 별개의 요청들로 나눕니다. 따라서, 실효 대역폭은 메모리 요청 개수 배만큼 떨어집니다. 여기서 한 가지 예외는 한 워프 안의 여러 스레드들이 같은 공유 메모리 위치로 접근할 때입니다. 이 경우는 브로드캐스트를 발생시킵니다. [계산 능력 2.x](#) 이상의 디바이스들은 공유 메모리 접근들을 멀티캐스트할 수 있는 추가 능력을 가지고 있습니다 (다시 말해, 그 워프의 여러 스레드들에 같은 값의 복사본들을 보낼 수 있습니다).

뱅크 충돌을 최소화하기 위해, 어떻게 메모리 주소들이 메모리 뱅크들로 사상되는지 그리고 어떻게 최적으로 메모리 요청들이 스케줄되는지 이해하는 것이 중요합니다.

계산 능력 2.x

계산 능력 2.x 의 디바이스들에서, 각 뱅크는 매 두 클럭 사이클마다 32 비트의 대역폭을 가집니다. 그리고 연속적인 32 비트 워드들이 연속적으로 뱅크들에 할당됩니다. 워프 크기는 32 스레드이고 뱅크 개수 또한 32 입니다. 그래서 뱅크 충돌들은 그 워프 안의 어떤 스레드들 사이에도 일어날 수 있습니다. 더 자세한 사항들은 쿠다 C 프로그래밍 가이드의 계산 능력 2.x 를 보십시오.

계산 능력 3.x

계산 능력 3.x 의 디바이스들에서, 각 뱅크는 매 클럭 사이클(*)마다 64 비트의 대역폭을 가집니다. 두 개의 다른 뱅킹(banking) 모드들이 있습니다. 연속적인 32 비트 워드들(32 비트 모드) 또는 연속적인 64 비트 워드들(64 비트 모드)이 연속적인 뱅크들에 할당됩니다. 워프 크기는 32 스레드이고 뱅크 개수 또한 32 입니다. 그래서 뱅크 충돌들은 그 워프 안의 어떤 스레드들 사이에도 생길 수 있습니다. 더 자세한 사항들은 쿠다 C 프로그래밍 가이드의 계산 능력 3.x 를 보십시오.

노트: (*) 그러나, 향상된 파워 효율을 위해, 계산 능력 3.x 디바이스들은 보통 계산 능력 2.x 의 디바이스들 보다 더 낮은 클럭 주파수들을 가집니다.

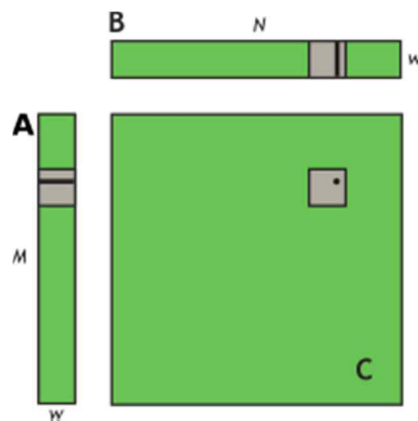
[9.2.2.2. 행렬 곱셈에서 공유 메모리 \(C=AB\)](#)

공유 메모리는 한 블록 안에 있는 스레드들을 협업할 수 있게 합니다. 한 블록 안에 있는 여러 개 스레드가 글로벌 메모리에 있는 같은 데이터를 (반복하여) 사용할 때, 공유 메모리는 그 글로벌 메모리에 있는 데이터 접근을 위해 사용될 수 있습니다. 공유 메모리는 응집되지 않은 메모리 접근을 피하기 위해서도 사용될 수 있습니다. 예를 들어, 응집된 패턴으로 글로벌 메모리에서 데이터를 불러와 저장하고 공유 메모리에서 그것을 재구성하는 식으로 사용될 수도 있습니다. 메모리 बैं크 충돌들이 없다면, 공유 메모리에서는 한 워프에 있는 스레드들이 비순차적 또는 정렬되지 않은 접근을 하여도 페널티가 없습니다.

공유 메모리의 사용이 한 단순한 행렬 곱 ($C=AB$) 예제를 통해 묘사됩니다. 이 예제에서, A 의 차원은 $M \times w$, B 의 차원은 $w \times N$, 그리고 C 의 차원은 $M \times N$ 입니다. 커널들을 단순하게 유지하기 위해, M 과 N 은 32 의 배수이고, w 는 32 로 놓습니다. 그리고 계산 능력 2.0 이상의 디바이스들이 사용된다고 가정됩니다.

이 문제의 자연스런 분해는 블록과 $w \times w$ 스레드 크기의 타일을 사용하는 것입니다. 그러므로, $w \times w$ 타일의 관점에서 볼 때, A 는 열 행렬, B 는 행 행렬, 그리고 C 는 그 둘의 외적입니다. [그림 9](#) 를 보십시오. $N/w \times M/w$ 블록들로 구성된 그리드 하나가 실행됩니다. 여기에서 각 스레드 블록은 A 의 한 단일 타일과 B 의 한 단일 타일에서 C 에 있는 한 다른 타일의 요소들을 계산합니다.

그림 9. 블록-행 행렬로 곱해진 블록-열 행렬. 블록-행 행렬 (B)로 곱해진 블록-열 행렬 (A) 그리고 결과 곱 행렬 (C).



이를 수행하기 위해, `simpleMultiply` 커널([최적화되지 않은 행렬 곱](#))은 행렬 C 의 한 타일의 출력 요소들을 계산합니다.

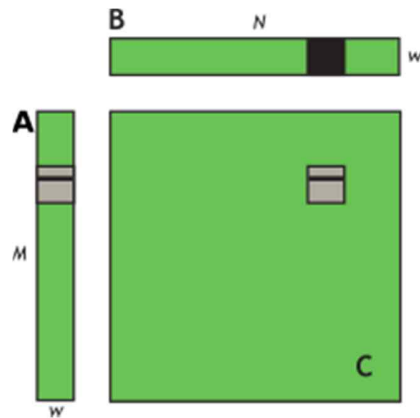
Unoptimized matrix multiplication

```
__global__ void simpleMultiply(float *a, float* b, float *c, int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

[최적화되지 않은 행렬 곱](#)에서, a , b , 그리고 c 는 각각 행렬 A , B , 그리고 C 를 위한 글로벌 메모리를 가리키는 포인터들입니다. `blockDim.x`, `blockDim.y`, 그리고 `TILE_DIM`은 모두 w 로 같습니다. $w \times w$ 스레드 블록 안에 있는 각 스레드는 C 의 한 타일의 한 요소를 계산합니다. `row`와 `col`은 한 특정 스레드에 의해 계산되는 C 에 있는 한 요소의 행과 열입니다. i 에 대한 `for` 루프는 A 의 한 행을 B 의 한 열과 곱합니다. 그 결과는 C 에 써집니다.

(ECC가 꺼진) 엔비디아 테슬라 K20X에서, 이 커널의 실효 대역폭은 단지 6.6GB/s입니다. 성능 분석을 위해, 우리는 `for` 루프에서 워프들이 어떻게 글로벌 메모리에 접근하는지 고려해 볼 필요가 있습니다. 스레드들로 구성된 각 워프는 C 의 한 타일의 한 행을 계산합니다. 그것은 A 의 한 행과 B 의 전체 타일 하나에 의존적입니다. [그림 10](#)은 이를 묘사합니다.

그림 10. 한 타일의 한 행 계산하기. A 의 한 행과 B 의 전체 타일 하나를 사용하여 C 에 있는 타일 하나의 한 행 계산하기.



for 루프의 각 반복(iteration) i 를 위해, 한 워프 안에 있는 스레드들은 B 의 한 행을 읽습니다. 그것은 모든 계산 능력들에서 순차적이고 응집된 접근입니다.

그러나, 각 반복 i 를 위해, 한 워프의 모든 스레드는 행렬 A 를 위한 글로벌 메모리로부터 같은 값을 읽습니다. 한 워프 안에서 그 인덱스 `row*TILE_DIM + i` 는 상수입니다. 비록 그런 접근이 계산 2.0 이상의 디바이스들에서 오직 1 트랜잭션을 요구하지만, 그 트랜잭션에는 낭비되는 대역폭이 있습니다. 왜냐하면, 그 캐시 라인에서 32 워드 중 오직 한 4 바이트 워드만 사용되기 때문입니다. 우리는 이 캐시 라인을 그 루프의 다음 반복들에서 다시 쓸 수도 있고 모든 32 워드들을 활용했을 수도 있습니다. 그러나, 일반적으로, 많은 워프가 같은 멀티프로세서에서 동시에 실행될 때, 캐시 라인은 반복 i 와 $i+1$ 사이에 캐시에서 쉽게 꼬집어 내질 수도 있습니다.

성능은 A 의 타일 하나를 공유 메모리로 읽음으로써 향상될 수 있습니다. [행렬 곱에서 글로벌 메모리 로드 효율 향상을 위해 공유 메모리 사용하기](#)는 그 예를 보여줍니다.

Using shared memory to improve the global memory load efficiency in matrix multiplication

```
__global__ void coalescedMultiply(float *a, float* b, float *c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
```

```

int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
for (int i = 0; i < TILE_DIM; i++) {
    sum += aTile[threadIdx.y][i] * b[i*N+col];
}
c[row*N+col] = sum;
}

```

[행렬 곱에서 글로벌 메모리 로드 효율 향상을 위해 공유 메모리 사용하기](#)에서, A 의 한 타일 안에 있는 각 요소는 글로벌 메모리에서 공유 메모리로 완전히 응집된 방식으로 (낭비되는 대역폭 없이) 오직 한 번만 읽힙니다. for 루프의 각 반복 내부에서, 공유 메모리 안에 있는 한 값은 한 워프 안에 있는 모든 스레드들에 브로드캐스트됩니다. A 의 그 타일을 공유 메모리로 읽은 다음, `__syncthreads()` 동기화 장벽 호출은 필요없습니다. 왜냐하면, 그 데이터를 공유 메모리로 쓰는 워프 내부의 스레드들이 그 데이터를 읽기 때문입니다 (노트: 계산 능력 2.0 이상의 디바이스들에서, `__syncthreads()` 대신, `__shared__` 배열은 정확성을 위해 `volatile` 로 표시(mark)될 필요가 있습니다. 엔비디아 페르미 호환성 가이드를 보십시오). 엔비디아 테슬라 K20X 에서, 이 커널은 7.8GB/s 의 실효 대역폭을 가집니다. 하드웨어 L1 캐시 축출(eviction) 정책이 그 응용 프로그램의 요구(needs)와 잘 맞지 않을 때 또는 L1 캐시가 글로벌 메모리로부터의 읽기들을 위해 쓰이지 않을 때, 이는 사용자에게 의해 관리되는(*user-managed*) 캐시로서 공유 메모리의 사용을 묘사합니다.

[행렬 곱에서 글로벌 메모리 로드 효율 향상을 위해 공유 메모리 사용하기](#)에서, 행렬 B 를 어떻게 다룰지에 따라 더 많은 향상이 만들어질 수 있습니다. 행렬 C 의 한 타일 안의 각 행들의 계산에서, B 의 전체 타일이 읽힙니다. 그 반복되는 B 타일의 읽기는 그것을 공유 메모리로 일단 읽음으로써 제거될 수 있습니다 ([추가 데이터를 공유 메모리로 읽음으로써 향상](#)).

Improvement by reading additional data into shared memory

```

__global__ void sharedABMultiply(float *a, float* b, float *c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM], bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

```

```

float sum = 0.0f;
aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
__syncthreads();
for (int i = 0; i < TILE_DIM; i++) {
    sum += aTile[threadIdx.y][i] * bTile[i][threadIdx.x];
}
c[row*N+col] = sum;
}

```

유념하십시오. [추가 데이터를 공유 메모리로 읽음으로써 향상](#)에서, B 타일 읽기 다음에 `__syncthreads()` 호출이 요구됩니다. 왜냐하면 한 워프가 다른 워프들에 의해 공유 메모리로 쓰여지는 공유 메모리로부터 데이터를 읽기 때문입니다. 엔비디아 테슬라 K20X 에서, 이 루틴의 실효 대역폭은 14.9GB/s 입니다. 유념하십시오. 두 경우에서, 성능 향상의 이유는 향상된 응집 때문이 아니라, 글로벌 메모리에서 데이터를 불러오는 불필요한 전송을 피했기 때문입니다.

[표 2](#)는 다양한 최적화들의 결과를 보여줍니다.

최적화	NVIDIA Tesla K20X
최적화 없음	6.6 GB/s
A 의 한 타일을 저장하기 위해 공유 메모리를 사용하여 응집됨	7.8 GB/s

B 의 한 타일에서 불필요한(redundant) 읽기를 제거하기 위해 공유 메모리 사용	14.9 GB/s
--	-----------

노트: 우선순위 중간: 불필요한 글로벌 메모리 전송을 피하기 위해 공유 메모리를 사용하십시오.

9.2.2.3. 행렬 곱셈에서 공유 메모리 ($C=AA^T$)

어떻게 글로벌 메모리의 건너 뛰는 접근들(strided accesses)과 공유 메모리 बैंक 충돌이 다뤄지는지를 설명하기 위해, 이전 행렬 곱 예제의 변형이 사용됩니다. 이 변형은 단순히 B 대신 A의 전치를 사용합니다. 그래서 $C = AA^T$ 입니다.

최적화되지 않은 글로벌 메모리로의 건너 뛰는 접근들의 처리에 있는 $C = AA^T$ 를 위한 한 가지 단순한 구현을 보겠습니다.

Unoptimized handling of strided accesses to global memory

```
__global__ void simpleMultiply(float *a, float *c, int M)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * a[col*TILE_DIM+i];
    }
    c[row*M+col] = sum;
}
```



```
}
```

[최적화되지 않은 글로벌 메모리로의 건너 뛰는 접근들](#)에서, A 의 행 번째와 열 번째의 내적을 취함으로써 C 의 행 번째, 열 번째 요소가 얻어집니다. 엔비디아 테슬라 M2090 에서, 이 커널을 위한 실효 대역폭은 3.64GB/s 입니다. 이 결과들은 상응하는 C = AB 커널을 위한 측정치들보다 상당히 더 낮습니다.

각 반복 i 를 위한 두 번째 항, `a[col*TILE_DIM+i]`에서, 차이는 어떻게 절반 워프 안의 스레드들이 A 의 요소들에 접근하는지에 있습니다. 한 워프의 스레드들을 위해, `col` 은 A 의 전치의 순차적 열들을 나타냅니다. 그리고 따라서 `col*TILE_DIM` 은 스트라이드가 `w` 인 글로벌 메모리의 한 건너 뛰는 접근입니다. 이는 많은 대역폭 낭비를 초래합니다.

건너 뛰는 접근을 피하기 위한 한 방법은 전처럼 공유 메모리를 사용하는 것입니다. 차이는 이 경우, 한 워프가 A 의 한 행을 공유 메모리 타일의 한 열로 읽는다는 점입니다. [글로벌 메모리로부터의 응집된 읽기들을 사용한 건너 뛰기 접근의 최적화된 처리](#)는 이를 보여줍니다.

An optimized handling of strided accesses using coalesced reads from global memory

```
__global__ void coalescedMultiply(float *a, float *c, int M)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM],
                    transposedTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    transposedTile[threadIdx.x][threadIdx.y] =
        a[(blockIdx.x*blockDim.x + threadIdx.y)*TILE_DIM + threadIdx.x];
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * transposedTile[i][threadIdx.x];
    }
    c[row*M+col] = sum;
```

```
}
```

내적의 두 번째 항에서 응집되지 않은 접근들을 피하고자, [글로벌 메모리로부터의 응집된 읽기들을 사용한 건너뛰기 접근의 최적화된 처리](#)는 공유된 transposedTile 을 사용합니다. 그리고 첫 번째 항에서 응집되지 않은 접근들을 피하고자, 이전 예제의 공유된 aTile 기법을 사용합니다. 엔비디아 테슬라 M2090 에서, 이 커널의 실효 대역폭은 27.5GB/s 입니다. 이 결과들은 $C = AB$ 의 마지막 커널에 의해 얻어진 성능들보다 살짝 더 낮습니다. 그 원인은 공유 메모리 뱅크 충돌들입니다.

for 루프 안의 transposedTile 안에 있는 요소들의 읽기는 충돌들에서 자유롭습니다. 왜냐하면, 각 절반 워프의 스레드들이 그 타일 안의 행들을 읽기 때문입니다. 이는 여러 뱅크들에 유닛 스트라이드로 접근되도록 유도합니다. 그러나, 그 타일을 글로벌 메모리에서 공유 메모리로 복사할 때, 뱅크 충돌이 발생합니다. 응집된 글로벌 메모리로부터의 로드들을 활성화하기 위해, 데이터는 글로벌 메모리에서 차례대로 읽힙니다. 그러나, 이는 열들에서 공유 메모리로의 쓰기를 요구합니다. 그리고 공유 메모리의 $w \times w$ 타일들의 사용 때문에, 이는 w 개 뱅크들의 스레드들 사이 한 스트라이드를 유발합니다. 그 스레드의 매 스레드는 같은 뱅크로 접근합니다. ([계산 능력](#) 2.0 이상에서, w 는 32 로 선택되었음을 기억하십시오.) 이 다 갈래(many-way) 뱅크 충돌들은 메모리 접근을 매우 느리게 만듭니다. 한 가지 간단한 해결책은 다음 줄에 있는 코드 같이, 추가 컬럼 하나를 가지도록 공유 메모리 배열을 채워 넣는 것입니다.

```
__shared__ float transposedTile[TILE_DIM][TILE_DIM+1];
```

이 채워 넣기(padding)은 충돌들을 완전히 제거합니다. 왜냐하면 스레드들 사이 스트라이드가 $w+1$ 뱅크들이기 때문입니다 (다시 말해, 현재 디바이스들에서 33). 나머지(modulo) 연산이 뱅크 인덱스들을 계산하기 위해 사용되므로, 이는 유닛 스트라이드와 같습니다. 엔비디아 테슬라 M2090 에서, 이 기법이 적용된 다음의 실효 대역폭은 39.2GB/s 입니다. 이는 마지막 $C = AB$ 커널의 결과들과 비슷합니다.

이 최적화들의 결과들은 [표 3](#)에 요약됩니다.

최적화	NVIDIA Tesla M2090
최적화 없음	3.6 GB/s
글로벌 읽기들을 응집하기 위해 공유 메모리 사용하기	27.5 GB/s
뱅크 충돌들 제거하기	39.2 GB/s

이 결과들은 [표 2](#)의 것들과 비교되어야 합니다. 이 표들에서 보여지듯, 공유 메모리의 사용은 성능을 크게 향상시킬 수 있습니다.

이 절에 있는 예제들은 공유 메모리 사용을 위한 세 가지 이유들을 설명하였습니다.

- 글로벌 메모리로 응집된 접근을 하기 위해, 특히 큰 스트라이드들을 피하기 위해 (일반적인 행렬들에서, 스트라이드는 32 보다 훨씬 큼)
- 불필요한 글로벌 메모리 로드들을 제거하거나 줄이기 위해
- 버려지는 대역폭을 줄이기 위해

[9.2.3. 로컬 메모리 \(Local Memory\)](#)

로컬 메모리라는 이름은 물리적 위치 때문이 아니라 그 메모리의 범위가 그 스레드에 지역적이기 때문에 붙여졌습니다. 사실, 로컬 메모리는 오프-칩입니다. 그러므로, 로컬

메모리로의 접근은 글로벌 메모리로의 접근 만큼 느립니다. 다른 말로, 용어 로컬이 더 빠른 접근을 뜻하지는 않습니다.

로컬 메모리는 오직 자동 변수들(automatic variables)을 붙들고 있기 위해 사용됩니다. nvcc 컴파일러가 그 변수를 들고 있기에 불충분한 레지스터 공간이 있다고 결정할 때, 이는 nvcc 컴파일러에 의해 수행됩니다. 로컬 메모리에 위치될 것 같은 자동 변수들은 큰 구조체 또는 배열들입니다. 그 구조체와 배열들은 너무 많은 레지스터 공간을 소모했을 것들 그리고 컴파일러가 동적으로 색인하기로 결정했을 것들입니다.

(nvcc 에 -ptx 또는 -keep 커맨드라인 옵션으로 컴파일링 함으로써 얻어진) PTX 어셈블리 코드를 조사함으로써, 우리는 변수가 첫 컴파일 단계들 동안 로컬 메모리에 위치했는지를 알 수 있습니다. 만약 그렇다면, 그 변수는 .local 니모닉을 사용하여 선언됩니다. 그리고 ld.local 과 st.local 니모닉을 사용하여 접근됩니다. 만약 그렇지 않더라도, 목표 아키텍처를 위해 너무 많은 레지스터 공간을 소모하는 변수들이 찾아지면, 다음 컴파일 단계들에서는 로컬 변수로 선언될 수도 있습니다. 한 특정 변수가 로컬 변수인지 확인할 수 있는 방법은 없습니다. 그러나 --ptxas-options=-v 옵션으로 실행될 때, 컴파일러는 커널 당 전체 로컬 메모리 사용(lmem)을 보고합니다.

[9.2.4. 텍스처 메모리 \(Texture Memory\)](#)

읽기 전용 텍스처 메모리 공간은 캐시됩니다. 그러므로, 텍스처 패치는 오직 캐시 미스 한 번에서 생긴 글로벌 메모리 한 번 읽기만큼의 비용이 듭니다. 그렇지 않으면 (캐시 히트이면), 그것은 단지 텍스처 캐시에서 한 번 읽기 만큼의 비용이 듭니다. 텍스처 캐시는 2 차원 공간적 지역성을 위해 최적화됩니다. 그래서 가까운 텍스처 주소들을 읽는 같은 워프의 스레드들은 가장 좋은 성능을 얻습니다. 텍스처 메모리는 또한 상수 지연 시간을 가진 스트리밍 패치들을 위해 설계되었습니다. 즉, 한 번의 캐시 히트는 DRAM 대역폭 요구를 줄입니다, 그러나 패치 지연 시간을 줄이지는 않습니다.

어떤 어드레싱 상황들에서, 텍스처 패칭(fetching)을 통해 디바이스 메모리를 읽는 것은 글로벌 또는 상수 메모리에서 디바이스 메모리를 읽기 것의 한 가지 이로운 대안이 될 수 있습니다..

[9.2.4.1. 추가적 텍스처 능력 \(Additional Texture Capabilities\)](#)

만약 텍스처들이 `tex1Dfetch()`가 아닌 `tex1D()`, `tex2D()`, 또는 `tex3D()`을 사용하여 패치되면, 그 하드웨어는 영상 처리와 같은 몇몇 응용 프로그램들을 위해 유용할 수도 있는 다른 능력들을 제공합니다. [표 4](#)는 그 능력들을 보여줍니다.

특징	사용	경고
필터링	텍셀들(texels) 사이 빠른, 저정밀 보간	그 텍스처 참조가 부동 소수점 데이터를 리턴할 때만 유효
정규화된 텍스처 좌표	해상도 독립적 코딩	없음
어드레싱 모드들	경계 경우들을 자동으로 처리 ¹	오직 정규화된 텍스처 좌표들과만 사용될 수 있음.

¹ 표 4의 바닥 행에 있는 경계 경우들의 자동 처리는, 그것이 유효한 주소 범위 밖으로 떨어졌을 때, 어떻게 한 텍스처 좌표가 처리되는지를 나타냅니다. clamp와 wrap의 두 가지 옵션이 있습니다. 만약 x 가 좌표이고 N 이 일차원 텍스처를 위한 텍셀(texel) 개수라고 가정합니다. clamp에서, 만약 $x < 0$ 이면 x 는 0으로 대체됩니다. 그리고 만약 $1 \leq x$ 이면 $1 - 1/N$ 으로 대체됩니다. wrap에서, x 는 $frac(x)$ 로 대체됩니다. 여기서 $frac(x) = x - floor(x)$ 입니다. Floor는 x 보다 작거나 같은 가장 큰 정수를 리턴합니다. 그래서, $N \neq 1$ 인 clamp 모드에서 1.3의 값을 가진 x 는 1.0으로 조여집니다(clamped). 대신 wrap 모드에서, 그 값은 0.3으로 바뀝니다.

한 커널 호출에서, 텍스처 캐시는 글로벌 메모리 쓰기들과 일관성 있게 유지되지 않습니다. 그래서 같은 커널 호출에서 글로벌 스토어들을 통해 써진 어드레스들에서 불러온 텍스처 패치는 정의되지 않은 데이터를 리턴합니다. 즉, 만약 그 위치가 이전 커널 호출 또는 메모리 복사에 의해 갱신되었으면, 한 스레드는 텍스처를 통해 한 메모리 위치를 안전하게 읽을 수 있습니다. 그러나 만약 같은 스레드 또는 같은 커널 호출 안에 있는 또다른 스레드에 의해 이전에 갱신되었으면, 그렇지 않습니다.

[9.2.5. 상수 메모리 \(Constant Memory\)](#)

한 디바이스에는 총 64 KB 의 상수 메모리가 있습니다. 그 상수 메모리 공간은 캐시됩니다. 결과적으로 상수 메모리에서의 읽기는 오직 한 번의 캐시 미스를 통해 글로벌 메모리에서 한 번 메모리 읽기를 한 만큼의 비용이 듭니다. 그렇지 않으면, 그것은 단지 상수 캐시에서 한 번 읽기만큼 비용이 듭니다. 한 워프에 있는 스레드들에 의한 다른 주소들로의 접근들은 직렬화됩니다. 따라서, 그 비용은 한 워프 안에 있는 모든 스레드들에 의한 고유한 주소들의 읽기 개수에 따라 선형적으로 증가합니다.

보통, 상수 캐시는 같은 워프 안의 스레드들이 오직 적은 수의 구분된 위치들로 접근할 때 가장 좋습니다. 만약 한 워프의 모든 스레드들이 같은 위치에 접근하면, 상수 메모리는 레지스터 접근만큼 빠르게 사용될 수 있습니다.

9.2.6. 레지스터 (Registers)

일반적으로, 레지스터에 접근하는 데는 명령어 당 클럭 사이클이 추가로 소모되지 않습니다. 그러나 지연이 생길 수도 있습니다. 레지스터 쓰고 읽기 의존성과 레지스터 메모리 뱅크 충돌 때문입니다.

쓰고 읽기 의존성의 호출 시간(latency)은 약 24 사이클입니다. 그러나 호출 시간은 충분한 스레드로 구성된 워프들을 가진 멀티프로세서들에 의해 완전히 숨겨집니다. 멀티프로세서 당 32 개 쿠다 코어를 가진 계산 능력 2.0 의 디바이스들에서, 호출 시간을 완전히 숨기기 위해서는 768 개 스레드들(24 개 워프)이 요구될 수도 있습니다.

레지스터 메모리 뱅크 충돌들을 피하기 위해, 컴파일러와 하드웨어 스레드 스케줄러는 명령어들을 가능한 최적으로 스케줄할 것입니다. 블록 당 스레드 수가 64 의 배수일 때, 그것들은 가장 좋은 결과들을 성취합니다. 이 규칙을 따르는 것 외에, 응용 프로그램은 이 뱅크 충돌들을 직접 제어할 수 없습니다. 특히, 데이터를 float4 또는 int4 타입들로 압축(pack)하기 위한 레지스터 관련 이유들이 없으면요.

9.2.6.1. 레지스터 압력 (Register Pressure)

주어진 과제(task)를 위해 사용할 수 있는 레지스터들이 충분히 없을 때, 레지스터 압력이 발생합니다. 비록 각 멀티프로세서가 수천 개의 32 비트 레지스터들을 가지고 있지만 (CUDA C 프로그래밍 가이드의 특징과 기술적 사양들을 보십시오), 이 레지스터들은 동시 스레드들 사이에는 막혀있습니다(partitioned). 컴파일러가 너무 많은 레지스터들을 할당하는 것을 막기 위해, `-maxrregcount=N` 컴파일러 커맨드라인 옵션이 사용될 수 있습니다 (`nvcc` 를 보십시오). 또는 스레드 당 할당되는 레지스터들의 최대 개수를

제어하기 위해 `launch bounds kernel definition qualifier` 가 사용될 수 있습니다 (CUDA C 프로그래밍 가이드의 실행 구성(Execution Configuration)을 보십시오).

9.3. 할당 (Allocation)

`cudaMalloc()`과 `cudaFree()`을 통한 디바이스 메모리 할당과 할당 해제는 비용이 많이 드는 연산들입니다. 그래서 할당이 전체 성능에 미치는 영향을 최소화하기 위해 어디에서든 가능하면 디바이스 메모리는 재사용 되어야 합니다. 그리고/또는 응용 프로그램에 의해 부분적으로 할당되어야 합니다.

10. 실행 구성 최적화 (Execution Configuration Optimizations)

좋은 성능을 얻기 위한 방법 중 하나는 디바이스의 멀티프로세서들을 가능한 바쁘게 유지하는 것입니다. 멀티프로세서들에 작업이 고르지 않게 할당된 디바이스는 최적의 아닌 성능을 낼 것입니다. 그러므로, 응용 프로그램의 스레드와 블록들은 하드웨어를 최대한 활용하도록 만들어져야 합니다. 그리고 작업이 자유롭게 분배되는 것을 막는 일은 제한되어야 합니다. 여기서 핵심 개념은 점유(occupancy)입니다. 점유는 다음 절들에서 설명됩니다.

몇몇 경우들에서, 우리의 응용 프로그램이 여러 개의 독립적 커널들이 동시에 실행할 수 있도록 함으로써, 하드웨어가 더 많이 활용될 수 있습니다. 여러 개 커널들을 한 번에 실행시키는 것은 동시 커널 실행이라 불립니다. 동시 커널 실행은 아래에서 설명됩니다.

또다른 중요한 개념은 한 특정 작업을 위해 할당된 시스템 자원들의 관리입니다. 이 자원 활용을 어떻게 관리하는지는 이 장의 마지막 절들에서 논의됩니다.

10.1. 점유 (Occupancy)

쿠다에서 스레드 명령어들은 차례대로 실행됩니다. 그리고, 그 결과로, 한 워프가 멈출 때 다른 워프들을 실행하는 것은 호출 시간(latency)들을 숨기기 위한 그리고 하드웨어를 바쁘게 하기 위한 유일한 방법입니다. 그러므로, 멀티프로세서들의 활성 워프 개수에 관련된 몇몇 측정 기준이 어떻게 효과적으로 그 하드웨어를 바쁘게 유지할지를 결정하기 위해 중요합니다. 그 측정 기준은 점유(occupancy)입니다.

점유는 멀티프로세서당 활성 워프 수 대 가능한 최대 활성 워프 수의 비율입니다. (가능한 최대 활성 워프 수는 쿠다 샘플의 deviceQuery 또는 쿠다 C 프로그래밍 가이드의 계산 능력을 보십시오.) 점유를 바라보는 또 다른 방법은 하드웨어 능력 대 활성화되어 사용 중인 처리 워프들의 백분율입니다.

더 높은 점유가 항상 더 높은 성능으로 연결되지는 않습니다. 즉, 점유가 증가한다고 성능도 꼭 높아지는 것은 아닙니다. 그러나, 낮은 점유는 항상 메모리 호출 시간을 감추는 것을 방해하고 성능 저하를 유발합니다.

10.1.1. 점유 계산하기 (Calculating Occupancy)

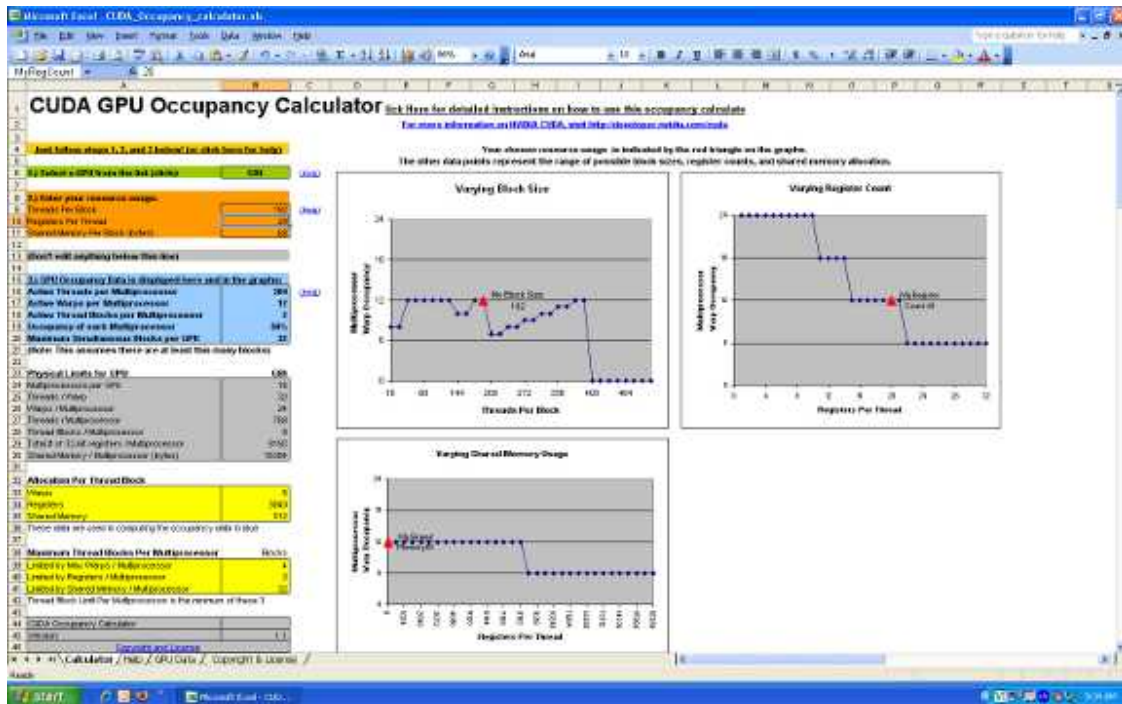
점유를 결정하는 몇 가지 요소 중 하나는 레지스터 사용 가능성입니다. 레지스터 저장용 스레드들이 로컬 변수들을 낮은 호출 시간(latency) 접근으로 유지할 수 있게 합니다. 그러나 (레지스터 파일로 알려진) 레지스터들은 한정된 자원입니다. 그래서, 한 멀티프로세서는 레지스터들을 (그 멀티프로세서 안의) 모든 스레드들과 반드시 공유해야 합니다. 레지스터들은 한 전체 블록에 모두 한꺼번에 할당됩니다. 그래서, 각 스레드 블록이 많은 레지스터를 사용하면, 멀티프로세서 하나에 머무를 수 있는 스레드 블록들의 수는 줄어듭니다. 따라서 그 멀티프로세서의 점유도 낮아집니다. 스레드당 레지스터의 최대 개수는 컴파일할 때, 파일당 (-maxrregcount 옵션을 사용하여) 또는 커널당 (__launch_bounds__ 한정자를 사용하여) 수동으로 설정될 수 있습니다. ([레지스터 압력](#)을 보십시오).

점유 계산에서, 각 스레드가 사용한 레지스터 개수는 핵심 요소 중 하나입니다. 예를 들어, [계산 능력](#) 1.1의 디바이스들은 멀티프로세서당 8,192개의 32비트 레지스터들을 가집니다. 그리고 최대 768개 동시에 머무르는 스레드들을 가질 수 있습니다 (24워프 x 워프당 32스레드들). 이 디바이스에서, 한 멀티프로세서의 100% 점유를 위해, 각 스레드는 최대 10개의 레지스터들을 쓸 수 있음을 뜻합니다. 그러나, 어떻게 레지스터 카운트가 점유에 영향을 끼치는지의 접근법은 레지스터 할당 입도(granularity)를 고려하지 않습니다. 예를 들어, 계산 능력 1.1의 한 디바이스에서, 스레드당 12개 레지스터를 사용하는 128스레드 블록을 가진 한 커널은 멀티프로세서당 5개의 활성 128스레드 블록으로 83%의 점유를 보입니다. 반면, 똑같이 스레드당 12개 레지스터들을 사용하는 256개 스레드 블록들을 가진 한 커널은 66%의 점유를 보입니다. 왜냐하면, 한 멀티프로세서에 오직 256스레드 블록들만 머무를 수 있기 때문입니다. 게다가, 계산 능력 1.1을 가진 디바이스들에서, 레지스터 할당들은 블록당 256레지스터들 근처로 반올림됩니다.

사용할 수 있는 레지스터 개수, 각 멀티프로세서에 동시에 머무르는 최대 스레드 개수, 그리고 레지스터 할당 입도는 계산 능력들에 따라 바뀝니다. 레지스터 할당에서 이 미묘한 차이들 때문에, 그리고 멀티프로세서의 공유 메모리가 스레드 블록들 사이에 나뉘어 있다는 사실 때문에, 레지스터 사용과 점유 사이 정확한 관계는 정해지기 어렵습니다. nvcc의 --

ptxas options=v 옵션은 각 커널을 위해 스레드당 사용된 레지스터 개수를 상세히 알립니다. 다양한 계산 능력의 디바이스를 위한 레지스터 할당 공식은 쿠다 C 프로그래밍 가이드의 하드웨어 멀티스레딩 절에 설명되어 있습니다. 그리고 그런 디바이스들에서 사용할 수 있는 전체 레지스터 개수는 쿠다 C 프로그래밍 가이드의 특징과 기술적 사양들 절에 설명되어 있습니다. 다른 방법으로, 엔비디아는 엑셀 스프레드시트의 형태의 점유 계산기를 제공합니다. 그 계산기는 개발자들이 최적의 균형(balance)을 다듬는 것과 다른 가능한 시나리오들을 더 쉽게 시험하는 것을 가능하게 합니다. 그림 11 에 보이는 이 스프레드시트는 [CUDA_Occupancy_Calculator.xls](#) 라 불립니다. 이 스프레드시트는 CUDA 도구모음 설치의 tools 하위디렉토리에 있습니다.

그림 11. GPU 멀티프로세서 점유를 보여주기 위해 쿠다 점유 계산기 사용하기



계산기 스프레드시트에 덧붙여, 점유는 엔비디아 비주얼 프로파일러의 성취된 점유(Achieved Occupancy) 측정 기준을 사용하여 결정될 수도 있습니다. 비주얼 프로파일러는 응용 프로그램 분석의 멀티프로세서 단계의 일부로 점유를 계산합니다.

10.2. 동시 커널 실행 (Concurrent Kernel Execution)

[비동기적으로 전송들과 계산 겹치기](#)에서 설명되었듯, 쿠다 스트림들은 커널 실행과 데이터 전송들을 겹치기 위해 쓰일 수 있습니다. 동시 커널 실행이 가능한 디바이스들에서, 스트림들은 (디바이스의 멀티프로세서들을 더 완전히 이용하려고) 여러 개 커널들을 동시에 실행하기 위해 쓰일 수 있습니다. `cudaDeviceProp` 구조체의 `concurrentKernels` 필드는 디바이스가 이 능력을 가지고 있는지를 나타냅니다 (또는 `deviceQuery` 쿠다 샘플의 출력에도 열거됩니다). 디폴트가 아닌 스트림들(스트림 0 를 제외한 스트림들)은 동시 실행이 요구됩니다. 왜냐하면 기본 스트림을 사용하는 커널 호출들은 오직 (어느 스트림의) 그 디바이스의 모든 이전 호출들이 완료된 뒤에만 시작하기 때문입니다. 그리고 (어느 스트림의) 그 디바이스 상의 어떤 연산도 그것들이 끝날 때까지 시작하지 않습니다.

다음 예제는 기본적 기법을 설명합니다. `kernel1` 과 `kernel2` 가 다른 비(non-)디폴트 스트림들에서 실행되므로, 동시 실행 가능 디바이스 하나는 그 커널들을 동시에 실행할 수 있습니다.

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
kernel1<<<grid, block, 0, stream1>>>(data_1);
kernel2<<<grid, block, 0, stream2>>>(data_2);
```

[10.3. 여러 개의 문맥 \(Multiple contexts\)](#)

쿠다 작업(CUDA work)은 문맥(context)이라 알려진 GPU 를 위한 처리 공간 안에서 발생합니다. 문맥은 그 GPU 를 위한 커널 시작들(launches)과 메모리 할당들을 캡슐화합니다. 그리고 문맥은 페이지 테이블 같은 구조물들을 지원합니다. 쿠다 드라이버 API 에서, 문맥은 명시적(explicit)입니다. 그러나, 쿠다 런타임 응용 프로그램 인터페이스(API)에서, 문맥은 완전히 암시적(implicit)입니다. 쿠다 런타임 API 는 문맥들을 자동으로 만들고 관리합니다.

쿠다 드라이버 API 를 가진, 한 쿠다 응용 프로그램 프로세스는 잠재적으로 주어진 GPU 하나를 위해 하나 이상의 문맥을 만들 수 있습니다. 만약 여러 개 쿠다 응용 프로그램 프로세스들이 같은 GPU 에 동시에 접근하면, 이는 거의 항상 여러 개의 문맥을 암시합니다. 왜냐하면, 쿠다 멀티-프로세스 서비스가 사용 중이지 않은 이상, 문맥 하나는 특정 호스트 프로세스 하나와 묶여있기 때문입니다.

여러 개 문맥들(그리고 글로벌 메모리 할당과 같은 그와 관련된 자원들)이 주어진 GPU 하나에 동시에 할당될 수 있습니다. 오직 이 문맥 중 하나만 그 GPU 에서 어떤 주어진 순간에 작업을 실행할 수 있습니다. 같은 GPU 를 공유하는 문맥들은 시분할됩니다. 추가 문맥을 만드는 일은 문맥당 데이터를 위한 메모리 간접비용과 문맥 교환을 위한 시간 간접비용을 초래합니다. 게다가, 여러 문맥의 작업을 동시에 실행했을 때, 문맥 교환은 사용성(utilization)을 떨어뜨릴 수 있습니다 ([동시 커널 실행](#)을 보십시오).

그러므로, 같은 쿠다 응용 프로그램 안에서는, GPU 당 여러 개 문맥 사용은 피하는 것이 가장 좋습니다. 이를 지원하기 위해, 쿠다 드라이버 API 는 각 GPU 의 한 특별한 문맥에 접근하고 관리하기 위한 주요 문맥(primary context)이라 불리는 방법들을 제공합니다. 한 스레드를 위한 현재 문맥이 없을 때, 이것들은 쿠다 런타임에 의해 암묵적으로 사용되는 것들과 같은 문맥들입니다.

```
// 프로그램/라이브러리를 초기화 할 때
CUcontext ctx;
cuDevicePrimaryCtxRetain(&ctx, dev);

// 프로그램/라이브러리가 작업을 시작(launch)할 때
cuCtxPushCurrent(ctx);
kernel<<<...>>>(...);
cuCtxPopCurrent(ctx);

// 그 문맥을 가진 프로그램/라이브러리가 끝났을 때
cuDevicePrimaryCtxRelease(dev);
```

노트: NVIDIA-SMI 는 GPU 를 배타적 계산 모드([exclusive compute mode](#))로 구성하기 위해 쓰일 수 있습니다. 배타적 계산 모드는 특정 GPU 하나에서 동시 문맥들을 가질 수 있는 스레드 그리고/또는 프로세스 수를 하나로 제한합니다.

10.4. 레지스터 의존성 감추기 (Hiding Register Dependencies)

노트: 우선순위 **중간:** 레지스터 의존성에서 생기는 시간 지연(latency)을 감추기 위해, 멀티프로세서당 충분한 수의 활성 스레드를 (다시 말해, 충분한 점유를) 유지하십시오.

레지스터 의존성은 한 명령어가 이전 명령이 쓴 레지스터에 저장된 결과를 사용할 때 발생합니다. 현재 쿠다 가능 GPU 들의 시간 지연은 대략 24 사이클입니다. 그래서 스레드들은 한 연산 결과를 사용하기 위해 반드시 24 사이클을 기다려야 합니다. 그러나 이 시간 지연은 다른 워프들의 스레드 실행을 통해 완전히 감춰질 수 있습니다. 자세한 사항들은 [레지스터들](#)을 보십시오.

10.5. 스레드와 블록 휴리스틱스 (Thread and Block Heuristics)

노트: 우선순위 중간: 블록 당 스레드 수는 32 스레드의 배수여야 합니다. 그렇게 해야만 최적의 계산 효율성과 응집을 이용할 수 있기 때문입니다.

그리드 당 블록들의 차원과 크기 그리고 블록 당 스레드들의 차원과 크기는 모두 중요한 요소들입니다. 이 파라미터들의 다차원적 양상(aspects)은 다차원 문제들이 쿠다로 쉽게 사상(mapping)되도록 합니다. 그리고 성능에는 아무런 역할을 하지 않습니다. 결과적으로, 이 절은 차원이 아닌 크기를 논의합니다.

시간 지연 감추기(latency hiding)와 점유는 멀티프로세서당 활성 워프 개수에 의존적입니다. 그것은 실행 파라미터들 때문에 생기는 자원(레지스터와 공유 메모리) 제한과 마찬가지로 암묵적으로 결정됩니다. 실행 파라미터 선택은 시간 지연 숨기기(점유)와 자원 활용 사이에 균형을 유지하는 문제입니다.

실행 구성 파라미터들의 선택은 동시에(in tandem) 되어야 합니다. 그러나 각 파라미터에 개별적으로 적용하는 어떤 휴리스틱스가 있습니다. 첫 번째 실행 구성 파라미터를 선택할 때 (그리드당 블록 수, 또는 그리드 크기), 주요 고려 사항은 전체 GPU 를 바쁘게 유지하는 것입니다. 한 그리드 안에 있는 블록 수는 멀티프로세서 수보다 더 커야 합니다. 모든 멀티프로세서들이 최소한 실행할 수 있는 블록 하나는 가져야 하기 때문입니다. 그리고 멀티프로세서당 여러 개의 활성 블록들이 있어야 합니다. 그렇게 해야 __syncthreads()를 기다리고 있지 않은 블록들이 하드웨어를 바쁘게 유지할 수 있기 때문입니다. 이 추천은 자원 사용 가능성에 영향을 받습니다. 그러므로, 이것은 두 번째 실행 파라미터(블록 스레드들의 수, 또는 블록 크기)는 꼭 공유 메모리 사용의 맥락에서 결정되어야 합니다. 미래 디바이스들에서도 동작할 수 있도록 크기를 조절하기 위해, 커널 실행당 블록 개수는 수천 개여야 합니다.

블록 크기를 선택할 때, 한 멀티프로세서에 여러 개 동시 블록들이 머무를 수 있음을 기억하는 것은 중요합니다. 그래서 점유는 블록 크기만으로 결정되지 않습니다. 특히, 더 큰 블록 크기가 더 높은 점유를 암시하는 것은 아닙니다. 예를 들어, 계산 능력 1.1 이하의 한 디바이스에서, 최대 블록 크기가 512 스레드인 한 커널은 66%의 점유를 가집니다.

왜냐하면 그 디바이스에서 멀티프로세서당 최대 스레드 수가 768 개이기 때문입니다. 그러므로, 멀티프로세서당 오직 한 블록만 활성 상태로 될 수 있습니다. 그러나, 그런 디바이스에서, 블록당 256 개 스레드를 가진 커널은 활성 상태로 머무르는 커널 세 개를 가지고 100% 점유를 가질 수 있습니다.

점유에서 언급되었듯, 더 높은 점유가 항상 더 좋은 성능으로 연결되는 것은 아닙니다. 예를 들어, 점유가 66%에서 100%로 향상되어도 일반적으로 성능은 그만큼 증가하지 않습니다. 더 낮은 점유를 가진 커널은 더 높은 점유를 가진 커널보다 스레드당 사용할 수 있는 더 많은 레지스터를 가질 것입니다. 이는 레지스터가 로컬 메모리로 더 적게 유출될 것임을 뜻합니다. 보통, 일단 50%의 점유에 이르면, 점유의 추가 상승은 더는 향상된 성능으로 연결되지 않습니다. 몇몇 경우들에서는, 훨씬 적은 수의 워프들로 시간 지연을 감추는 것이 가능합니다. 여기에는 명령어 수준 병렬성(ILP)이 이용됩니다. 자세한 사항은 http://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf 을 참고하십시오.

블록 크기 선택하기에 관련된 많은 요소들이 있습니다. 그리고 여기에는 어쩔수 없이 약간의 실험들이 요구됩니다. 그러나, 약간의 경험적 법칙들(rules of thumb)은 지켜져야 합니다.

- 블록당 스레드들은 꼭 워프 크기의 배수여야 합니다. 채워지지 않은 워프들로 계산되어 버려지는 계산을 피하고 응집을 가능하게 하기 위함입니다.
- 블록당 최소 64 개 스레드들이 사용되어야 합니다. 그리고 멀티프로세서당 여러 개의 동시 블록들이 있어야 합니다.
- 블록당 128 과 256 스레드가 좋은 선택입니다. 그리고 다른 블록 크기를 가진 실험들을 위한 좋은 시작 범위입니다.
- 만약 시간 지연이 성능에 영향을 끼치면, 멀티프로세서당 한 개의 큰 스레드 블록 보다는 몇 개(3~4)의 더 작은 스레드 블록들을 사용하십시오. 이는 특히 자주 `__syncthreads()`를 호출하는 커널들에 도움이 됩니다.

유념하십시오. 한 스레드 블록이 한 멀티프로세서에서 사용할 수 있는 것보다 더 많은 레지스터를 할당할 때, 커널 실행은 실패합니다. 너무 많은 공유 메모리 또는 스레드가 요청될 때도 마찬가지입니다.

10.6. 공유 메모리의 효과 (Effects of Shared Memory)

공유 메모리는 응집이나 불필요한 글로벌 메모리 접근 제거와 같은 몇몇 상황에서 도움이 될 수 있습니다.

그러나, 공유 메모리는 또한 점유에 제약 사항으로 동작할 수 있습니다. 많은 경우, 한 커널이 요구하는 공유 메모리의 양은 선택된 블록 크기와 관련됩니다. 그러나 스레드들을 공유 메모리 요소들로 사상(mapping)하는 것이 꼭 1:1 일 필요는 없습니다. 예를 들어, 한 커널에서 32 x 32 요소로 구성된 공유 메모리 배열을 사용하는 것이 바람직할 수도 있습니다. 그러나 블록당 최대 스레드 수는 512 이므로, 블록당 32 x 32 스레드를 가진 커널은 실행될 수 없습니다. 그런 경우, 각 스레드가 그 공유 메모리 배열의 두 개 또는 네 개 요소를 처리하는, 32 x 16 또는 32 x 8 스레드를 가진 커널들이 실행될 수 있습니다. 비록 블록당 스레드 수 제한이 논점은 아니지만, 한 스레드로 공유 메모리 배열의 여러 요소들을 처리하는 접근법은 이로울 수 있습니다. 각 요소에 공통인 몇몇 연산들이 그 스레드에 의해 한 번에 수행될 수 있기 때문입니다. 그 스레드에 의해 처리된 공유 메모리 요소 개수에 대한 비용을 나눠 갇으면서요.

점유에 대한 성능의 민감도를 결정하는 한 가지 유용한 기법은, 실행 구성의 세 번째 파라미터에 특정되었듯, 동적으로 할당된 공유 메모리를 통한 실험입니다. 이 파라미터를 단순히 증가시킴으로써 (커널 수정 없이), 그 커널의 점유를 효과적으로 줄이고 성능에 대한 효과를 측정할 수 있습니다.

이전 절에서 언급되었듯, 일단 점유가 50% 이상에 이르면, 일반적으로 더 높은 점유율을 얻고자 파라미터들을 최적화하는 것은 가치가 없습니다. 그 이전 기법은 그런 최고치에 도달했는지 아닌지를 결정하는 데 쓰일 수 있습니다.

11. 명령어 최적화 (Instruction Optimization)

명령어들이 어떻게 실행되는지 앞으로써, (소위 프로그램에서 핫스팟이라 불리는) 자주 수행되는 코드에 대해, 우리는 종종 유용할 수도 있는 저수준 최적화를 할 수 있습니다. 모범 관례에서는 이 최적화는 모든 고수준 최적화가 완료된 뒤에 할 것을 제안합니다.

11.1. 산술 명령어들 (Arithmetic Instructions)

단 정밀 float 은 가장 좋은 성능을 제공하므로, 그 사용이 크게 장려됩니다. 개별 산술 연산의 처리량은 쿠다 C 프로그래밍 가이드에 자세히 설명됩니다.

11.1.1. 나누기 나머지 연산들 (Division Modulo Operations)

노트: 우선순위 낮음: 비싼 나눗기와 나머지 계산들을 피하기 위해 시프트 연산을 쓰십시오.

정수 나눗기와 나머지 연산은 특히 비쌉니다(처리 시간이 오래 걸립니다). 따라서 이 연산들은 언제나 가능하면 사용을 피하거나 비트별 연산들로 대체되어야 합니다. 만약 n 이 2 의 멍수이면, (i / n) 은 $(i >> \log_2 n)$ 과 같고, $(i \% n)$ 은 $(i \& n-1)$ 과 같습니다.

만약 n 이 리터럴이면, 컴파일러는 이 변환들을 수행할 것입니다. (더 많은 정보는 [쿠다 C 프로그래밍 가이드의 성능 가이드라인](#)을 참고하십시오).

11.1.2. 제곱근의 역수 (Reciprocal Square Root)

제곱근의 역수는 항상 단 정밀에서 `rsqrtf()` 처럼 그리고 배 정밀에서 `rsqrt()` 처럼 명시적으로 호출되어야 합니다. 컴파일러는 `1.0f/sqrtf(x)` 를 `rsqrtf()` 로 최적화합니다. 단, 이는 오직 IEEE-754 의미론(semantics)을 어기지 않을 때만 적용됩니다.

11.1.3. 다른 산술 명령어들 (Other Arithmetic Instructions)

노트: 우선순위 낮음: `double` 에서 `float` 로 자동 형 변환을 피하십시오.

컴파일러는 가끔 변환 명령어를 삽입합니다. 이는 추가적 실행 사이클을 만듭니다. 여기, 그 예들이 있습니다.

- 피연산자가 일반적으로 `int` 로 변환되어야 하는 `char` 또는 `short` 에서 동작하는 함수들
- 단 정밀 부동 소수점 계산들의 입력으로 사용되는 (어떤 타입 접미사도 없이 정의된) 배 정밀 부동 소수점 상수들

우리는 두 번째 경우를 단 정밀 부동 소수점 상수를 사용해 피할 수 있습니다. 그 상수들은 `3.141592653589793f`, `1.0f`, `0.5f` 같이 `f` 접미사를 붙여 정의됩니다. 이 접미사는 정확도에 대한 암시 뿐 아니라 성능에도 영향을 미칩니다. 정확도에 대한 영향은 [double 로 승격 그리고 float 로 강등](#)에서 논의됩니다. 유념하십시오. 이 구별은 [계산 능력](#) 2.x 디바이스들에서 성능을 위해 특히 중요합니다.

단 정밀 코드에서는 `float` 타입과 단 정밀 수학 함수 사용이 강력히 추천됩니다. 계산 능력 1.2 이전의 디바이스들처럼 자체적으로 배 정밀을 지원하지 않는 디바이스들을 위해 컴파일을 할 때, 각 배 정밀 부동 소수점 변수는 단 정밀 부동 소수점 양식으로 바꿉니다

(그러나 그 크기는 64 비트로 유지됩니다). 그리고 배 정밀 연산이 단 정밀 연산으로 강등됩니다.

쿠다 수학 라이브러리의 보완적 오차 함수(complementary error function), `erfcf()`는 특히 완전한 단 정밀 정확도에서 빠릅니다.

11.1.4. 작은 분수 인자들을 가진 누승법 (Exponentiation With Small Fractional Arguments)

몇몇 분수 지수들을 위한, 제곱근, 세제곱근, 그리고 그 역수를 사용한 `pow()`의 사용에 비해 누승법(exponentiation)은 더 크게 가속될 수 있습니다. 또한, 1/3 처럼 지수가 정확히 한 부동 소수점 수로 표현될 수 없는 누승법들에서, 이는 훨씬 더 정확한 결과를 제공합니다. `pow()`의 사용이 초기 표현적 오차들을 증폭하기 때문입니다.

아래 표의 공식들은 `x >= 0`, `x != -0` 에서 유효합니다. 즉, `signbit(x) == 0`.

계산	공식	ulps (double) ¹	ulps (single) ²
$x^{1/9}$	<code>r = rcbrt(rcbrt(x))</code>	1	1/1
$x^{-1/9}$	<code>r = cbrt(rcbrt(x))</code>	1	1/1
$x^{1/6}$	<code>r = rcbrt(rsqrt(x))</code>	1	2/2
$x^{-1/6}$	<code>r = rcbrt(sqrt(x))</code>	1	1/2
$x^{1/4}$	<code>r = rsqrt(rsqrt(x))</code>	1	2/2

$x^{-1/4}$	<code>r = sqrt(rsqrt(x))</code>	1	1/3
$x^{1/3}$	<code>r = cbrt(x)</code>	1	1/1
$x^{-1/3}$	<code>r = rcbrt(x)</code>	1	1/1
$x^{1/2}$	<code>r = sqrt(x)</code>	0	0/3
$x^{-1/2}$	<code>r = rsqrt(x)</code>	1	2/2
$x^{2/3}$	<code>r = cbrt(x); r = r*r</code>	2	3/3
$x^{-2/3}$	<code>r = rcbrt(x); r = r*r</code>	2	3/3
$x^{3/4}$	<code>r = sqrt(x); r = r*sqrt(r)</code>	2	2/6
$x^{-3/4}$	<code>r = rsqrt(x); r = r*sqrt(r)</code>	2	4/5
$x^{7/6}$	<code>r = x*rcbrt(rsqrt(x))</code>	2	2/2
$x^{-7/6}$	<code>r = (1/x) * rcbrt(sqrt(x))</code>	2	3/3

$x^{5/4}$	<code>r = x*rsqrt(rsqrt(x))</code>	2	3/3
$x^{-5/4}$	<code>r = (1/x)*sqrt(rsqrt(x))</code>	2	3/5
$x^{4/3}$	<code>r = x*cbrt(x)</code>	1	2/2
$x^{-4/3}$	<code>r = (1/x)*rcbrt(x)</code>	2	2/3
$x^{3/2}$	<code>r = x*sqrt(x)</code>	1	1/3
$x^{-3/2}$	<code>r = (1/x)*sqrt(x)</code>	2	3/3
¹ 바르게 반올림된 결과와 비교하여			
² 바르게 반올림된 결과와 비교하여 1st: -prec-sqrt=true -prec-div=true 2nd: -prec-sqrt=false -prec-div=false			

11.1.5. 수학 라이브러리들 (Math Libraries)

노트: 우선순위 **중간:** 속도가 정밀도보다 중요할 때는 빠른 수학 라이브러리를 사용하십시오.

두 종류의 런타임 수학 연산들이 지원됩니다. 그것은 이름으로 구분될 수 있습니다. 몇몇은 언더스코어들로 접두어가 붙은 이름들을 가집니다. 반면, 아닌 것들도 있습니다 (예를 들어, `__functionName()` 대 `functionName()`). `__functionName()` 이름 붙이기 규칙을 따르는 함수들은 하드웨어 수준으로 직접 사상합니다. 그 함수들은 더 빠르지만 약간 더 낮은 정확도를 제공합니다 (예를 들어, `__sinf(x)`와 `__expf(x)`). `functionName()` 이름 붙이기 규칙을 따르는 함수들은 더 느리지만 더 높은 정확도를 가집니다 (예를 들어, `sinf(x)`와 `expf(x)`). `__sinf(x)`, `__cosf(x)`, `__expf(x)`의 처리량은 `sinf(x)`, `cosf(x)`, `expf(x)`의 처리량보다 훨씬 더 많습니다. 만약 인자 x 의 크기가 줄어야하면, `sinf(x)`, `cosf(x)`, `expf(x)`는 훨씬 더 비쌉니다 (약 10 배 정도 더 느립니다). 게다가, 그런 경우, 그 인자 줄임(argument-reduction) 코드는 로컬 메모리를 사용합니다. 이는 성능에 훨씬 더 (나쁜) 영향을 줄 수 있습니다. 로컬 메모리의 높은 시간 지연 때문입니다. 더 자세한 사항들은 쿠다 C 프로그래밍 가이드에서 보실 수 있습니다.

또한 유념하십시오. 같은 인자를 가진 사인과 코사인이 계산될 때마다, 그 사인코사인 함수들은 성능 최적화를 위해 사용되어야 합니다.

- 단 정밀 빠른 수학을 위한 `__sincosf()` (다음 문단을 보십시오)
- 정규 단 정밀을 위한 `sincosf()`
- 배 정밀을 위한 `sincos()`

`nvcc`의 `-use_fast_math` 컴파일러 옵션은 매 `functionName()` 호출을 `__functionName()` 호출이 되도록 강제합니다. 이 스위치는 정확도가 성능보다 덜 중요할 때마다 꼭 사용되어야 합니다. 이는 초월 함수의 경우 자주 해당됩니다. 이 스위치는 단 정밀 부동 소수점에 대해서만 효과적입니다.

노트: 우선순위 **중간:** 언제든지 가능하면 더 느리고, 더 일반적인 것보다 더 빠르고 더 전문화된 수학 함수들을 사용하십시오.

작은 정수 거듭제곱들(예를 들어, x^2 or x^3)을 위해, 명시적인 곱셈은 거의 확실하게 `pow()` 같은 일반적 누승법 루틴들의 사용보다 더 빠릅니다. 컴파일러 최적화 향상은 지속적으로 이 차이를 좁혀 나가는 것을 추구하는 반면, 명시적 곱셈(또는 동등한 특정 목적을 위해 만들어진 인라인 함수 또는 매크로의 사용)은 큰 이점을 가질 수 있습니다. 그 이점은 같은 기저(base)를 가진 여러 거듭제곱들이 필요해질 때 증가됩니다 (예를 들어, x^2 과 x^5 가 아주 근접하여 계산되는 곳). 왜냐하면 이는 컴파일러의 공통 부표현 제거(common sub-expression elimination, CSE) 최적화에 도움이 되기 때문입니다.

기저 2 또는 10 을 사용하는 누승법을 위해, `pow()` 또는 `powf()` 대신, `exp2()` 또는 `expf2()` 그리고 `exp10()` 또는 `expf10()` 함수들을 사용하십시오. `pow()`과 `powf()`는 레지스터 압력과 명령어 카운트 측면에서 무거운 함수입니다. 왜냐하면 일반적 누승법에서 발생하는 많은 특별한 경우들과 기저와 지수의 전체 범위들에 걸쳐 좋은 정확도를 얻는 것의 어려움 때문입니다. 반면, `exp2()`, `exp2f()`, `exp10()`, 그리고 `exp10f()` 함수들은 성능 면에서 `exp()` 그리고 `expf()`와 비슷합니다. 그리고 그것들의 `pow()/powf()` 대응물들보다 10 배 더 빠를 수 있습니다.

지수가 1/3 인 누승법을 위해서는, 범용 누승법 함수들 `pow()` 또는 `powf()` 보다는 `cbrt()` 또는 `cbrtf()` 함수를 사용하십시오. `cbrt()` 또는 `cbrtf()` 함수가 `pow()` 또는 `powf()` 보다 훨씬 더 빠릅니다. 비슷하게, 지수가 -1/3 인 누승법을 위해서는 `rcbrt()` 또는 `rcbrtf()`를 사용하십시오.

`sin(π *<expr>)`을 `sinpi(<expr>)`로, `cos(π *<expr>)`을 `cospi(<expr>)`로, 그리고 `sincos(π *<expr>)`을 `sincospi(<expr>)`로 대체하십시오. 이는 정확도와 성능 모두에 이롭습니다. 한 예로서, 사인 함수를 라디안 대신 도(degree)로 평가하기 위해서는 `sinpi(x/180.0)`를 사용하십시오. 비슷하게, 함수 인자가 π *<expr>의 형태일 때 단 정밀 함수들 `sinpi()`, `cospi()`, `sincospi()`는 `sinf()`, `cosf()`, `sincosf()`를 대체해야 합니다. (`sinpi()`가 `sin()`보다 성능 이득이 있는 이유는 인자가 단순히 줄기 때문입니다. `sinpi()`는 pi 를 단 정밀 또는 배 정밀 근사치가 아닌 무한대로 정확한 수학적 pi 를 사용하여 효과적으로 그리고 암묵적으로 곱합니다.)

11.1.6. 정밀도 관련 컴파일러 플래그들 (Precision-related Compiler Flags)

기본적으로, `nvcc` 컴파일러는 계산 능력 2.x 의 디바이스들을 위한 IEEE 준수 코드를 만들어냅니다. 그러나, `nvcc` 는 더 빠르지만 덜 정확한 코드를 만들어내기 위한 옵션도 제공합니다. 그 코드는 이전 디바이스들을 위해 만들어진 코드에 더 가깝습니다.

- `-ftz=true` (비정규화된(denormalized) 숫자들이 0 으로 초기화됨)
- `-prec-div=false` (덜 정확한 나누기)
- `-prec-sqrt=false` (덜 정확한 제곱근)

또 다른, 더 공격적인 옵션은 `-use_fast_math` 입니다. 이 옵션은 모든 `functionName()` 호출을 대응하는 `__functionName()` 호출로 바꿉니다. 이는 코드가 더 빠르게 실행되게 만듭니다. 단, 정밀도와 정확도는 줄어듭니다. [수학 라이브러리들](#)을 보십시오.

11.2. 메모리 명령어들 (Memory Instructions)

노트: 우선순위 높음: 글로벌 메모리의 사용을 최소화하십시오. 어디서든 가능하면 공유 메모리 접근을 선호하십시오.

메모리 명령어들은 공유, 로컬, 또는 글로벌 메모리로부터 읽거나 그 메모리들로 쓰는 명령어들을 포함합니다. 캐시되지 않은 로컬 또는 글로벌 메모리에 접근할 때, 400~600 클럭 사이클의 메모리 지연 시간이 있습니다.

한 예로, 다음 샘플 코드의 할당 연산자는 높은 처리량을 가집니다. 그러나, 결정적으로 데이터를 글로벌 메모리에서 읽는 데 400~600 클럭 사이클의 지연 시간이 있습니다.

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

만약 글로벌 메모리 접근 완료를 위해 기다리는 동안 이슈(실행)될 수 있는 충분한 독립적 연산 명령어들이 있으면, 이 글로벌 메모리 시간 지연의 많은 부분은 스레드 스케줄러에 의해 숨겨질 수 있습니다. 그러나, 글로벌 메모리 접근은 언제든지 가능하면 피하는 것이 가장 좋습니다.

12. 제어 흐름 (Control Flow)

12.1. 분기와 발산 (Branching and Divergence)

노트: 우선순위 높음: 같은 워프 안에서 다른 실행 경로들을 피하십시오.

흐름 제어 명령어(`if`, `switch`, `do`, `for`, `while`)는 같은 워프의 스레드들을 발산시킴으로써 (즉, 다른 실행 경로들을 따르게 함으로써) 명령어 처리량에 크게 영향을 끼칠 수 있습니다. 만약 이런 일이 일어나면, 그 다른 실행 경로들은 반드시 직렬화되어야 합니다. 한 워프의 스레드들은 모두 한 프로그램 카운터를 공유하기 때문입니다. 이는 그 워프를 위한 전체 명령어 개수를 늘립니다. 모든 다른 실행 경로들이 완료되면, 그 스레드들은 같은 실행 경로로 다시 합쳐집니다.

제어 흐름이 그 스레드 ID 에 의존하는 경우들에서, 가장 좋은 성능을 얻기 위해서는 제어 조건이 발산적인 워프 수를 최소화하기 위해 써져야 합니다.

그 블록에 걸친 그 워프들의 분배는 결정적이므로, 이것은 가능합니다. 이는 쿠다 C 프로그래밍 가이드의 SIMT 구조에서 언급되었습니다. 한 작은 예제는 제어 조건이 오직 (`threadIdx / WSIZE`)에 의존할 때 입니다. 여기서 `WSIZE` 는 워프 크기입니다.

이 경우에, 워프 발산은 없습니다. 제어 조건이 완벽히 그 워프에 정렬되었기 때문입니다.

12.2. 분기 예측 (Branch Predication)

노트: 우선순위 낮음: 루프들 또는 제어문들 대신 컴파일러가 분기 예측을 쉽게 하도록 만드십시오.

때때로, 컴파일러는 분기 예측을 사용하여 루프를 풀거나(unroll) if 또는 switch 문을 최적화하여 제거할 수도 있습니다. 이 경우, 어떤 워프도 발산하지 않을 수 있습니다. 프로그래머는 또한 다음과 같이 루프 풀기(loop unrolling)를 할 수 있습니다.

```
#pragma unroll
```

이 pragma 에 대한 더 많은 정보는, 쿠다 C 프로그래밍 가이드를 참고하십시오.

분기 예측을 사용할 때, 실행이 제어 조건에 의존하는 어떤 명령어들도 생략되지 않습니다. 대신, 각 그런 명령어는 제어 조건에 따라 참 또는 거짓으로 설정되는 스레드 당 조건 코드 또는 술부(predicate)에 관련됩니다. 비록 이 명령어들이 실행을 위해 스케줄되었지만, 오직 참 술부를 가진 명령어들만 실제로 실행됩니다. 거짓 술부를 가진 명령어들은 결과들을 쓰지 않습니다. 그리고 그것들은 또한 주소를 평가하거나 피연산자들을 읽지 않습니다.

컴파일러는 오직 그 분기 조건에 의해 제어되는 명령어들의 개수가 한 특정 문턱값보다 작거나 같은 때만, 한 분기 명령어를 단정된(predicated) 명령어들로 바꿉니다. 만약 컴파일러가 그 조건이 많은 발산적 워프들을 만들 것 같다고 결정하면, 이 문턱값은 7 입니다. 그렇지 않으면 4 입니다.

12.3. 루프 카운터 Signed 대 Unsigned

노트: 우선순위 낮음 중간: 루프 카운터로 부호 없는 정수들(unsigned integers) 대신 부호 있는 정수들(signed integers)을 쓰십시오.

C 언어 표준에서, 부호 없는 정수 오버플로우 의미론은 잘 정의되어 있습니다. 대신 부호 있는 정수 오버플로우는 정의되지 않은 결과들을 초래합니다. 그러므로, 컴파일러는 부호 있는 산술을 부호 없는 산술을 가진 것보다 더 공격적으로 최적화할 수 있습니다. 이는 루프 카운터들에서 특히 그렇습니다. 왜냐하면 루프 카운터들은 항상 양인 값들을 가지는 일이 흔하기 때문입니다. 그 카운터들은 unsigned 로 선언되지 않을 수도 있습니다. 그러나, 조금 더 좋은 성능을 위해서는 그것들이 signed 로 선언되어야 합니다.

예를 들어, 다음 코드를 고려해 봅시다:

```
for (i = 0; i < n; i++) {
    out[i] = in[offset + stride*i];
}
```

여기서, 부표현(sub-expression) `stride*i` 는 32 비트 정수를 오버플로우할 수 있습니다. 그래서 만약 `i` 가 unsigned 로 선언되면, 그 오버플로우 의미론은, 그렇지 않았으면 컴파일러가 적용했을 수도 있는, 강도 줄임(strength reduction) 같은 몇몇 최적화의 사용을 막습니다. 대신 `i` 가 (오버플로우 의미론이 정의되지 않은) signed 로 선언되었으면, 컴파일러는 그 최적화를 사용하기 위한 더 많은 자유를 가집니다.

12.4. 루프에서 발산적 스레드 동기화하기 (Synchronizing Divergent Threads in a Loop)

노트: 우선순위 높음: 발산적 코드 안에서 `__syncthreads()` 사용을 피하십시오.

잠재적으로 발산적 코드 (예를 들어, 한 입력 배열에 대한 한 루프) 안에서 여러 스레드를 동기화하는 것은 예상치 못한 에러를 유발할 수 있습니다. 모든 스레드가 `__syncthreads()`가 호출된 지점으로 수렴되는 시점을 확실히 하기 위해 우리는 주의해야 합니다. 다음 예제는 1 차원 블록들을 위해 어떻게 이것을 적절히 할지를 설명합니다.

```
unsigned int imax = blockDim.x * ((nelements + blockDim.x - 1) / blockDim.x);
```

```

for (int i = threadIdx.x; i < imax; i += blockDim.x)
{
    if (i < nelements)
    {
        ...
    }

    __syncthreads();

    if (i < nelements)
    {
        ...
    }
}

```

이 예제에서, 그 루프는 발산을 피하면서 각 스레드를 위해 같은 반복 횟수를 가지도록 주의깊게 쓰여졌습니다 (imax 는 그 블록 크기의 배수로 반올림된 요소들의 개수입니다). 범위를 벗어난 접근들을 막기 위해 그 루프 안쪽에 가드들(guards)이 추가되었습니다. __syncthreads()가 있는 지점에서, 모든 스레드들은 수렴됩니다.

잠재적으로 발산적인 코드에서 호출된 디바이스 함수가 __syncthreads()를 호출할 때, 비슷한 주의가 기울여져야 합니다. 이 논점을 해결하는 가장 쉬운 방법은 비발산적 코드에서 디바이스 함수를 호출하는 것입니다. 그리고 thread_active 플래그를 파라미터로 그 디바이스 함수에 전달하는 것입니다. 이 thread_active 플래그는, 모든 스레드가 __syncthreads()로 참여하는 것을 허용하면서, 그 디바이스 함수 안에서 어떤 스레드가 그 계산에 사용되었어야 했는지를 가리키기 위해 사용됩니다.

13. 쿠다 응용 프로그램 배포 (Deploying CUDA Applications)

응용 프로그램에서 하나 이상의 요소들을 GPU 가속하였으면, 그 결과물을 원래 기대치와 비교하는 것이 가능합니다. 초기 평가 단계에서 개발자는 주어진 핫스팟들을 가속함으로써 얻을 수 있는 잠재적 속도 향상의 상한을 결정할 수 있었습니다.

전체 속도 향상을 더 높이기 위해 다른 핫스팟들을 공략하기 전에, 개발자는 그 부분적으로 병렬화된 구현을 고려해야 합니다. 그리고 그것을 제품에 적용해야 합니다. 이는 여러 가지 이유에서 중요합니다. 예를 들어, 이는 사용자들이 가능한 빨리 성능 이득을 얻게 합니다 (그 속도 향상은 부분적이나 소중합니다). 그리고 이는 개발자와 사용자의 위험(risk)을 최소화합니다. 개발이 혁명적이기 보다는 진화적으로 진행되도록 유도되기 때문입니다.

14. 프로그래밍 환경 이해하기 (Understanding the Programming Environment)

각 세대의 엔비디아 프로세서들에는 쿠다가 이용할 수 있는 GPU 의 새 특징들이 추가됩니다. 결과적으로, 그 아키텍처의 특징들을 이해하는 것이 중요합니다.

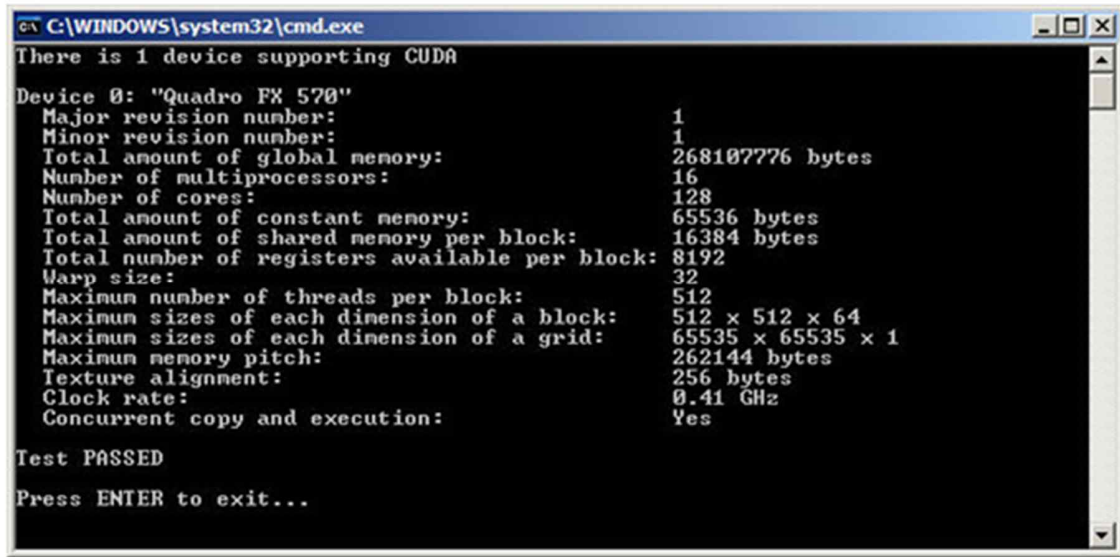
프로그래머들은 두 가지 버전 숫자들을 인지하고 있어야 합니다. 첫 째는 계산 능력(compute capability)입니다. 그리고 둘 째는 쿠다 런타임과 쿠다 드라이버 API 들의 버전 숫자들입니다.

14.1. 쿠다 계산 능력 (CUDA Compute Capability)

계산 능력은 그 하드웨어의 특징들을 설명합니다. 그리고 블록당 최대 스레드 수 그리고 멀티프로세서당 레지스터 수 같은, 그 디바이스와 다른 사양들에 의해 지원되는 명령어들의 집합을 반영합니다. 더 높은 계산 능력 버전은 더 낮은 (즉, 더 이전) 버전들을 포함하는 더 큰 집합입니다. 다시 말해, 계산 능력은 이전 버전에 대해 호환성을 가집니다 (backward compatible).

디바이스에서 GPU 의 계산 능력은 deviceQuery 쿠다 샘플에서 설명되었듯 프로그래밍적으로 문의될 수 있습니다. 그림 12 는 그 프로그램의 출력을 보여줍니다. 이 정보는 cudaGetDeviceProperties()를 호출하고 그것이 리턴하는 구조체 안의 정보에 접근함으로써 얻어집니다.

그림 12. deviceQuery 에 의해 보고된 샘플 쿠다 구성 데이터



```
C:\WINDOWS\system32\cmd.exe
There is 1 device supporting CUDA

Device 0: "Quadro FX 570"
Major revision number:          1
Minor revision number:          1
Total amount of global memory:  268107776 bytes
Number of multiprocessors:      16
Number of cores:                128
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 16384 bytes
Total number of registers available per block: 8192
Warp size:                      32
Maximum number of threads per block: 512
Maximum sizes of each dimension of a block: 512 x 512 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
Maximum memory pitch:          262144 bytes
Texture alignment:             256 bytes
Clock rate:                    0.41 GHz
Concurrent copy and execution:  Yes

Test PASSED
Press ENTER to exit...
```

계산 능력의 주(major)와 부(minor) 버전 숫자들은 그림 12 의 세 번째와 네 번째 줄에 보여집니다. 이 시스템의 디바이스 0 는 계산 능력 1.1 을 가집니다.

다양한 GPU 들의 계산 능력에 대한 더 자세한 사항들은 쿠다 C 프로그래밍 가이드의 CUDA-Enabled GPUs and Compute Capabilities 에 있습니다. 특히, 개발자들은 그 디바이스에 있는 멀티프로세서 수, 레지스터 수, 사용할 수 있는 메모리 양, 그리고 그 디바이스의 특별한 능력들에 주목해야 합니다.

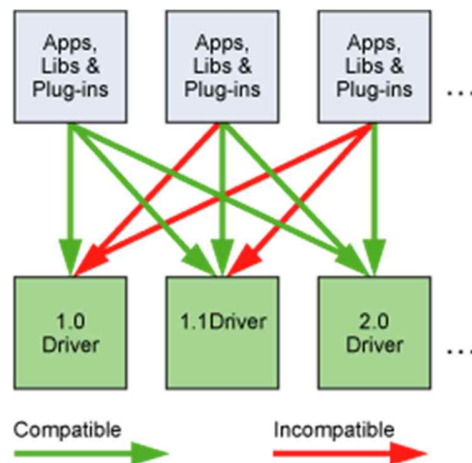
14.2. 추가적 하드웨어 데이터 (Additional Hardware Data)

어떤 하드웨어 특징들은 계산 능력으로 설명되지 않습니다. 예를 들어, 호스트와 디바이스 사이 비동기 데이터 전송들과 커널 실행을 겹쳐서 하는 능력은 계산 능력 1.1 의 대부분의 그러나 모두는 아닌 GPU 들에서 사용할 수 있습니다. 그런 경우, 그 디바이스가 어떤 특징의 능력을 가지는지 보시려면 `cudaGetDeviceProperties()`를 호출하십시오. 그 디바이스 속성 구조체의 `asyncEngineCount` 필드는 커널 실행과 데이터 전송들을 겹치는 것이 가능한지를 나타냅니다 (그리고 만약 그렇다면, 얼마나 많은 동시 전송들이 가능한지를 가리킵니다). 비슷하게, `canMapHostMemory` 필드는 제로 복사 데이터 전송들이 수행될 수 있는지를 가리킵니다.

14.3. 쿠다 런타임 그리고 드라이버 응용 프로그램 인터페이스 버전 (CUDA Runtime and Driver API Version)

쿠다 드라이버 API 와 쿠다 런타임은 쿠다를 사용할 수 있게 하는 두 가지 프로그래밍 인터페이스입니다. 그것의 버전 숫자는 개발자가 API 들과 관련된 특징들을 확인할 수 있게 하고, 한 응용 프로그램이 현재 설치된 것보다 더 새로운 (더 나중) 버전인지 확인할 수 있게 합니다. 쿠다 드라이버 API 는 현재보다 더 오래된 버전들에 대해서만 호환성을 가지므로, 이 확인은 중요합니다. 또한, 이는 드라이버 API 의 한 특정 버전에 대해 컴파일된 플러그인과 라이브러리들(쿠다 런타임을 포함하는)이 이후 드라이버들에서도 계속 동작할 것임을 뜻합니다. 그러나, 한 특정 버전의 드라이버 API 에 대해 컴파일된 응용 프로그램들, 플러그인들, 그리고 라이브러리들(쿠다 런타임을 포함하는)은 이전 버전의 드라이버들에 대해서는 동작하지 않을 수도 있습니다. [그림 13](#) 은 이를 묘사합니다.

그림 13. 쿠다 버전들의 호환성



14.4. 어느 계산 능력을 타겟을 할 것인가 (Which Compute Capability Target)

런타임에 제시될 그 하드웨어의 계산 능력이 의심스러울 때, 계산 능력 2.0 으로 가정되는 것이 가장 좋습니다. 계산 능력 2.0 은 쿠다 C 프로그래밍 가이드의 특징과 기술적 사양들 절에 정의되어 있습니다.

특정 버전의 엔비디아 하드웨어와 쿠다 소프트웨어를 목표로 삼기 위해, `nvcc` 의 `-arch`, `-code`, 그리고 `-gencode` 옵션들을 사용하십시오. 예를 들어, 워프 셔플 연산을 사용하는 코드들은 반드시 `-arch=sm_30`(또는 더 높은 계산 능력)으로 컴파일되어야 합니다.

최대 호환성을 위한 빌딩(building)을 보십시오. 여기서는 여러 세대의 쿠다 가능 디바이스를 위한 코드를 동시에 만드는 방법이 논의됩니다.

14.5. 쿠다 런타임 (CUDA Runtime)

쿠다 소프트웨어 환경의 호스트 런타임 요소는 오직 호스트 함수들에 의해서만 사용될 수 있습니다. 그 요소는 다음을 다루기 위한 함수들을 제공합니다.

- 디바이스 관리
- 문맥 관리
- 메모리 관리
- 코드 모듈 관리
- 실행 제거
- 텍스처 참조 관리
- OpenGL 와 Direct3D 의 상호운용성(Interoperability)

더 낮은 쿠다 드라이버 API 와 비교하여, 쿠다 런타임은 암시적 초기화, 문맥 관리, 그리고 디바이스 코드 모듈 관리를 제공함으로써 디바이스 관리를 훨씬 쉽게 하였습니다. nvcc 가 생성한 C/C++ 호스트 코드는 그 쿠다 런타임을 활용합니다. 그래서 이 코드에 링크하는 응용 프로그램들은 그 쿠다 런타임에 의존할 것입니다. 비슷하게, cuBLAS, cuFFT, 그리고 다른 쿠다 도구모음 라이브러리들을 사용하는 어떤 코드들 또한 그 쿠다 런타임에 의존할 것입니다. 쿠다 런타임은 이 라이브러리들에 내부적으로 사용됩니다.

쿠다 런타임 API 를 구성하는 함수들은 쿠다 도구모음 참조 지침서(CUDA Toolkit Reference Manual)에 설명됩니다.

쿠다 런타임은 그 커널이 시작되기 전에, 커널 로딩과 커널 파라미터 설정, 그리고 시작 구성(launch configuration)을 다룹니다. 쿠다 런타임은 암시적 드라이버 버전 확인, 코드 초기화, 쿠다 문맥 관리, 쿠다 모듈 관리 (함수 사상을 위한 cubin), 커널 구성, 그리고 파라미터 전달을 모두 수행합니다.

이는 두 주요한 부분들로 구성됩니다.

- C 스타일 함수 인터페이스 (`cuda_runtime_api.h`).
- C 스타일 함수들 위에 만들어진 C++ 스타일 편의 래퍼들(convenience wrapper) (`cuda_runtime.h`)

런타임 API 에 대한 더 많은 정보는 쿠다 C 프로그래밍 가이드의 쿠다 C 런타임을 참고하십시오.

15. 배포를 위한 준비 (Preparing for Deployment)

15.1. 쿠다 사용 가능성 시험 (Testing for CUDA Availability)

쿠다 응용 프로그램을 배포하기 전에, 종종 그 응용 프로그램이 (타겟 머신이 쿠다 가능 GPU 그리고/또는 충분한 버전의 설치된 엔비디아 드라이버를 가지지 못한 경우에도) 동작할 수 있는지 확실히 하는 것이 바람직합니다. (알려진 구성의 단일 머신을 타겟으로 하는 개발자들은 이 절을 생략할 수도 있습니다.)

쿠다 가능 GPU 탐지

한 응용 프로그램이 임의의/알려지지 않은 구성의 타겟 머신들에 배포될 때, 그런 디바이스가 없을 경우에도 적절한 조치를 취하기 위해, 응용 프로그램은 꼭 쿠다 가능 GPU 가 존재하는지 명시적으로 시험해야 합니다. `cudaGetDeviceCount()` 함수는 사용할 수 있는 디바이스 수를 조회하기 위해 쓰일 수 있습니다. 모든 쿠다 런타임 API 함수들처럼, 만약 쿠다 가능 GPU 가 없으면, 이 함수는 정중하게(*gracefully*) 실패하고 `cudaErrorNoDevice` 를 그 응용 프로그램에 리턴할 것입니다. 또는 만약 적절한 버전의 설치된 쿠다 가능 GPU 가 없으면, 이 함수는 정중하게 실패하고 `cudaErrorInsufficientDriver` 를 그 응용 프로그램에 리턴할 것입니다. 만약 `cudaGetDeviceCount()`가 오류를 보고하면, 그 응용 프로그램은 다른 코드 경로로 되돌아가야 합니다.

여러 GPU 를 가진 시스템은 다른 하드웨어 버전들과 능력들을 가진 GPU 들을 포함할 수도 있습니다. 같은 응용 프로그램에 여러 GPU 가 사용될 때, 여러 세대의 하드웨어들을 섞는 것보다는 같은 타입의 GPU 를 사용하는 것이 추천됩니다. `cudaChooseDevice()` 함수는 희망하는 특징들과 가장 잘 어울리는 디바이스를 선택하는 데 쓰일 수 있습니다.

하드웨어와 소프트웨어 구성 탐지

어떤 기능을 활성화하기 위해, 한 응용 프로그램이 어떤 하드웨어 또는 소프트웨어 능력에 의존적일 때, 사용할 수 있는 디바이스 구성과 설치된 소프트웨어 버전들에 대한 자세한 사항들은 쿠다 API 를 통해 조회될 수 있습니다.

`cudaGetDeviceProperties()` 함수는 사용할 수 있는 디바이스들의 다양한 특징들(features)을 보고합니다. 여기에는 그 디바이스의 [계산 능력](#)도 포함됩니다 (쿠다 C 프로그래밍 가이드의 계산 능력 절을 보십시오). 어떻게 사용할 수 있는 쿠다 소프트웨어 API 버전들을 조회하는지에 대한 자세한 사항들은 [쿠다 런타임과 드라이버 API 버전](#)을 참고해 주십시오.

15.2. 오류 처리 (Error Handling)

모든 쿠다 런타임 API 호출들은 `cudaError_t` 타입의 오류 코드를 리턴합니다. 만약 오류가 발생하지 않았으면, 그 리턴 값은 `cudaSuccess` 일 것입니다. (예외는 커널 시작들(launches)입니다. 그 시작들은 void 를 리턴합니다. 그리고 `cudaGetErrorString()`은 그 함수로 전달되는 `cudaError_t` 코드를 설명하는 문자열을 리턴합니다). (cuBLAS, cuFFT 와 같은) 쿠다 도구모음 라이브러리들은 그 라이브러리들의 오류 코드들을 리턴합니다.

몇몇 쿠다 API 호출과 모든 커널 시작들은 호스트 코드에 대해 비동기적입니다. 오류들 또한 호스트에 비동기적으로 보고될 수도 있습니다. 종종 이는 `cudaMemcpy()` 또는 `cudaDeviceSynchronize()`를 호출하는 동안과 같이 호스트와 디바이스가 서로 동기화하는 다음번에 발생합니다.

함수들이 실패하지 않을 것 같더라도, 항상 모든 쿠다 API 함수들의 오류 리턴 값들을 확인하십시오. 이는 응용 프로그램이 오류를 가능한 한 빨리 찾고 복구할 수 있게 합니다. 쿠다 API 오류를 확인하지 않는 응용 프로그램은 (GPU 에 의해 계산된 데이터가 불완전하다는, 유효하지 않다는, 또는 초기화되지 않았다는 사실에 대한 알림도 없이) 그대로 종료될 것입니다.

노트: 쿠다 도구모음 샘플들은 다양한 쿠다 API 들로 오류 확인을 위한 여러 헬퍼 함수들을 제공합니다. 이 헬퍼 함수들은 쿠다 도구모음의 `samples/common/inc/helper_cuda.h` 에 위치합니다.

15.3. 최대 호환성을 위한 건설 (Building for Maximum Compatibility)

쿠다 가능 디바이스의 각 세대는 그 디바이스에 의해 지원되는 특징들을 나타내는 계산 능력 버전을 가집니다 ([쿠다 계산 능력](#)을 보십시오). 하나 이상의 계산 능력 버전들이 한 파일을 빌드하는 동안 nvcc 컴파일러로 특정될 수 있습니다. 그 응용 프로그램의 타겟 GPU(들)의 본래 계산 능력을 위한 컴파일은 1) 응용 프로그램 커널들의 가장 좋은 성능을

확실히 얻기 위해 그리고 2) 주어진 세대의 GPU 에서 사용할 수 있는 특징들이 확실히 사용될 수 있도록 하기 위해 중요합니다.

한 응용 프로그램이 여러 계산 능력을 위해 (nvcc 의 -gencode 플래그를 사용하여) 동시에 빌드되면, 그 특정된 계산 능력의 바이너리들은 실행할 수 있는 파일(excutable)로 결합됩니다. 그리고 쿠다 드라이버는 현재 디바이스의 계산 능력에 따라 가장 적절한 바이너리를 런타임에 선택합니다. 만약 적절한 본래 바이너리(cubin)가 사용될 수 없으면, 그러나 중간 PTX 코드(추상 가상 명령어 집합을 타겟으로 하는 그리고 포워드 호환성을 위해 사용되는)가 사용될 수 있으면, 그 커널은 그 PTX 에서 Just In Time(JIT)으로 컴파일됩니다 ([컴파일러 JIT 캐시 관리 도구들](#)을 보십시오). 만약 그 PTX 또한 사용될 수 없으면, 그 커널 시작은 실패합니다.

Windows

```
nvcc.exe -ccbin "C:\vs2008\VC\bin"
-Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MT"
-gencode=arch=compute_20,code=sm_20
-gencode=arch=compute_30,code=sm_30
-gencode=arch=compute_35,code=sm_35
-gencode=arch=compute_50,code=sm_50
-gencode=arch=compute_50,code=compute_50
--compile -o "Release\mykernel.cu.obj" "mykernel.cu"
```

Mac/Linux

```
/usr/local/cuda/bin/nvcc
-gencode=arch=compute_20,code=sm_20
-gencode=arch=compute_30,code=sm_30
-gencode=arch=compute_35,code=sm_35
-gencode=arch=compute_50,code=sm_50
-gencode=arch=compute_50,code=compute_50
-O2 -o mykernel.o -c mykernel.cu
```

다른 방법으로, nvcc 의 커맨드 라인 옵션 `-arch=sm_XX` 가 사용될 수 있습니다. 이 옵션은 위에서 설명된 더 명시적인 `-gencode=` 커맨드 라인 옵션들의 약칭입니다.

```
-gencode=arch=compute_XX,code=sm_XX  
-gencode=arch=compute_XX,code=compute_XX
```

그러나, `-arch=sm_XX` 커맨드 라인 옵션이 PTX 백엔드 타겟을 기본으로 포함하는 반면 (그것이 암시하는 `code=compute_XX` 타겟 때문에), 이 옵션은 한 번에 한 타겟 cubin 구조만 특정할 수 있습니다. 그리고 같은 nvcc 커맨드 라인에 여러 `-arch=` 옵션들을 사용할 수 없습니다. 이것이 위 예제들이 `-gencode=`을 명시적으로 사용한 이유입니다.

15.4. 쿠다 런타임과 라이브러리 분배하기 (Distributing the CUDA Runtime and Libraries)

쿠다 응용 프로그램들은 쿠다 런타임 라이브러리에 의지하여 빌드됩니다. 쿠다 런타임 라이브러리는 디바이스, 메모리, 그리고 커널 관리를 다룹니다. 쿠다 드라이버와 달리, 쿠다 런타임은 여러 버전들에 걸친 포워드(forward)나 백워드(backward) 바이너리 호환성을 보장합니다. 따라서, 동적 링크를 또는 그 쿠다 런타임에 의지하는 그 밖의 정적 링크를 사용할 때, 쿠다 런타임은 응용 프로그램과 함께 [재분배](#)되기에 좋습니다. 이것은 사용자가 (그 응용 프로그램이 의지하여 빌드되었던 것과) 같은 설치된 쿠다 도구모음을 가지지 않더라도, 실행될 수 있는 파일(executable)이 실행될 수 있음을 확실히 할 것입니다.

노트: 쿠다 런타임을 정적으로 링크할 때, 그 런타임의 여러 버전이 같은 응용 프로그램 프로세스에 동시에 평화롭게 공존할 수 있습니다. 예를 들어, 만약 한 응용 프로그램이 쿠다 런타임의 한 버전을 사용하면, 그리고 그 응용 프로그램의 플러그인 하나가 다른 버전과 정적으로 링크되면, 설치된 엔비디아 드라이버가 둘 모두를 위해 충분하기만 하면, 그것은 완벽히 받아들여질 수 있습니다.

정적으로 링크된 쿠다 런타임

가장 쉬운 옵션은 쿠다 런타임에 의지하여 정적으로 링크하는 것입니다. 만약 쿠다 5.5 이후에서 링크를 위해 nvcc 를 사용하면, 이것이 기본 설정입니다. 정적 링킹은 실행할 수 있는 파일을 약간 더 크게 만듭니다. 그러나 정적 링킹은 바른 버전의 런타임 라이브러리 함수들이 그 응용 프로그램 라이브러리에 (그 쿠다 런타임 라이브러리의 별도 재분배 없이) 포함됨을 보장합니다.

동적으로 링크된 쿠다 런타임

만약 쿠다 런타임에 의지하는 정적 링킹이 몇 가지 이유로 실용적이지 않다면, 동적으로 링크된 버전의 쿠다 런타임 라이브러리 또한 사용될 수 있습니다. (이것은 쿠다 5.0 이전에서 기본 옵션이었습니다.)

쿠다 5.5 이후에서 nvcc 를 사용하여 응용 프로그램을 링크할 때, 쿠다 런타임과 함께 동적 링킹을 사용하기 위해, `--cudart=shared` 플래그를 그 링크 커맨드 라인에 추가하십시오. 그렇지 않으면 [정적으로 링크된 쿠다 런타임 라이브러리](#)가 기본으로 사용됩니다.

응용 프로그램이 쿠다 런타임에 의지하여 동적으로 링크된 다음, 이 버전의 런타임 라이브러리는 응용 프로그램과 함께 번들(꾸러미)로 제공되어야 합니다. 그 런타임 라이브러리는 응용 프로그램 실행 가능 파일(executable)과 같은 디렉터리로 또는 설치 경로의 하위디렉터리로 복사될 수 있습니다.

다른 쿠다 라이브러리들

비록 쿠다 런타임이 정적 링킹 옵션을 제공하지만, 쿠다 도구모음에 포함된 다른 라이브러리들(cuBLAS, cuFFT 등)은 오직 동적으로 링크된 형태로만 사용할 수 있습니다. [동적으로 링크된 버전의 쿠다 런타임 라이브러리](#)와 같이, 그 응용 프로그램을 배포할 때, 이 라이브러리들은 응용 프로그램 실행 가능 파일(executable)과 함께 번들로 제공되어야 합니다.

[15.4.1. 쿠다 도구모음 라이브러리 재배포 \(CUDA Toolkit Library Redistribution\)](#)

쿠다 도구모음의 최종 사용자 라이선스 동의(EULA, End-User License-Agreement)는 많은 쿠다 라이브러리들의 재배포를 어떤 조건 하에서 허용합니다. EULA 는 라이브러리들에 의존적인 응용 프로그램들이 그 응용 프로그램이 빌드되고 시험된 [정확한 버전의](#)

[라이브러리들과 함께 재배포되는 것을](#) 허용합니다. 이는 최종 사용자들이 머신에 다른 버전의 쿠다 도구모음을 설치 했거나 아무 것도 설치하지 않은 곤란한 상황을 피할 수 있게 합니다. 자세한 사항들은 EULA 를 참고해 주십시오.

노트: 이는 엔비디아 드라이버에는 적용되지 않습니다. 최종 사용자는 여전히 반드시 엔비디아 드라이버를 최종 사용자의 GPU(들)과 운영 체제에 적절히 내려받고 설치해야 합니다.

[15.4.1.1. 어떤 파일들을 재배포 하나](#)

하나 이상의 쿠다 라이브러리들의 동적으로 링크된 버전들을 재배포할 때, 재배포될 필요가 있는 정확한 파일들을 밝히는 것은 중요합니다. 다음 예제들은 한 예로 쿠다 도구모음 5.5 의 cuBLAS 라이브러리를 사용합니다.

리눅스 (Linux)

리눅스의 공유된 라이브러리에서, SONAME 이라 불리는 문자열 필드가 있습니다. 그 필드는 그 라이브러리의 바이너리 호환성 수준을 나타냅니다. 거기에 의존하여 응용 프로그램이 빌드된 라이브러리의 SONAME 은 반드시 그 응용 프로그램과 함께 재배포되는 라이브러리의 파일명과 일치해야 합니다. 예를 들어, 표준 쿠다 도구모음 설치에서, 파일 `libcublas.so` 와 `libcublas.so.5.5` 는 모두 특정 빌드의 cuBLAS 를 가리키는 심볼릭 링크입니다. 그 이름은 `libcublas.so.5.5.x` 와 같이 명명됩니다. 여기서 `x` 는 빌드 넘버입니다 (이를테면, `libcublas.so.5.5.17`). 그러나, 이 라이브러리의 SONAME 은 “`libcublas.so.5.5`” 로 주어집니다.

```
$ objdump -p /usr/local/cuda/lib64/libcublas.so | grep SONAME
SONAME                  libcublas.so.5.5
```

이 때문에, 그 응용 프로그램을 링킹할 때, `-lcublas` 가 (버전 넘버가 특정되지 않고) 사용되어도, 링크 타임에 찾아진 SONAME 은 “`libcublas.so.5.5`” 이 그 응용 프로그램을 로딩할 때 동적 로더가 찾을 파일의 이름임을 암시합니다. 따라서 (SONAME 은) 그 응용 프로그램과 함께 재배포된 파일의 이름(또는 같은 파일으로의 심볼릭 링크)이어야만 합니다.

ldd 도구는 라이브러리의 정확한 파일 이름들을 밝히는 데 유용합니다. 응용 프로그램은 응용 프로그램을 로딩할 때 선택했을 라이브러리(그리고, 만약 있다면, 라이브러리의 복사 경로)를 런타임에 찾을 것입니다. 그 응용 프로그램은 현재 라이브러리 탐색 경로로 주어집니다.

```
$ ldd a.out | grep libcublas
libcublas.so.5.5 => /usr/local/cuda/lib64/libcublas.so.5.5
```

맥 (Mac)

맥 OS X 의 공유된 라이브러리에서, `install name` 이라 불리는 필드가 있습니다. 그 필드는 기대되는 설치 경로와 그 라이브러리의 파일명을 가리킵니다. 쿠다 라이브러리들은 또한 이 파일명을 바이너리 호환성을 나타내기 위해 사용합니다. 이 필드의 값은 그 바이너리에 의존하여 빌드된 응용 프로그램으로 전파됩니다. 그리고 이 필드의 값은 런타임에서 정확한 버전의 라이브러리 위치를 찾기 위해 사용됩니다.

예를 들어, 만약 cuBLAS 라이브러리의 설치 이름이 `@rpath/libcublas.5.5.dylib` 로 주어지면, 그 라이브러리 버전은 5.5 이고 응용 프로그램과 함께 재배포된 이 라이브러리의 카피(copy)는 반드시 `libcublas.5.5.dylib` 로 명명되어야 합니다. 단, 링크 타임에는 오직 `-lcublas` 만 (특정된 버전 넘버 없이) 사용됩니다. 게다가, 이 파일은 그 응용 프로그램의 `@rpath` 에 설치되어야 합니다. [재배포된 쿠다 라이브러리들은 어디에 설치하는가](#)를 보십시오.

한 라이브러리의 설치 이름을 보기 위해, `otool -L` 명령어를 사용하십시오.

```
$ otool -L a.out
a.out:
```

```
@rpath/libcublas.5.5.dylib (...)
```

윈도우즈 (Windows)

윈도우즈에서 쿠다 라이브러리들의 바이너리 호환성 버전은 그 파일명의 일부로 표시됩니다.

예를 들어, cuBLAS 5.5 에 링크된 64 비트 응용 프로그램은 런타임에 `cublas64_55.dll` 을 찾을 것입니다. 비록 그 응용 프로그램이 의존하여 링크된 파일은 `cublas.lib` 이지만, `cublas64_55.dll` 이 그 응용 프로그램과 함께 재배포되어야 합니다. 32 비트 응용 프로그램들에서는 그 파일이 `cublas32_55.dll` 였을 것입니다.

그 응용 프로그램이 런타임에 찾을 것을 기대되는 정확한 DLL 파일명을 검증하기 위해, 비주얼 스튜디오 커맨드 프롬프트의 `dumpbin` 도구를 사용하십시오.

```
$ dumpbin /IMPORTS a.exe
Microsoft (R) COFF/PE Dumper Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file a.exe

File Type: EXECUTABLE IMAGE

Section contains the following imports:

...
cublas64_55.dll
...
```

15.4.1.2. 재배포된 쿠다 라이브러리들을 어디에 설치하나

일단 재배포를 위한 바른 라이브러리 파일들이 밝혀졌으면, 그것들은 설치를 위해 그 응용 프로그램이 그것들을 찾을 수 있는 한 위치로 구성되어야만(configured) 합니다.

윈도우즈에서, 만약 쿠다 런타임 또는 다른 동적으로 링크된 쿠다 툴킷 라이브러가 같은 디렉토리에 실행할 수 있는 형태(executable)로 위치하면, 윈도우즈는 그것을 자동으로 locate 할 것입니다. 리눅스 또는 맥에서, 그 실행할 수 있는 파일(executable)이 시스템 경로들을 찾기 전에 이 라이브러리들을 위한 그것의 로컬 경로를 찾게하기 위해, `-rpath` 링커 옵션이 꼭 사용되어야 합니다.

Linux/Mac

```
nvcc -I $(CUDA_HOME)/include
-Xlinker "-rpath '$ORIGIN'" --cudart=shared
-o myprogram myprogram.cu
```

Windows

```
nvcc.exe -ccbin "C:\vs2008\VC\bin"
-Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MT" --cudart=shared
-o "Release\myprogram.exe" "myprogram.cu"
```

노트: 당신의 비주얼 스튜디오의 위치를 반영하기 위해, `-ccbin` 의 값 조절이 필요할 수도 있습니다.

라이브러리들이 재배포될 다른 경로를 특정하기 위해, 아래와 같은 링커 옵션들을 사용하십시오.

Linux/Mac

```
nvcc -I $(CUDA_HOME)/include
-Xlinker "-rpath '$ORIGIN/lib'" --cudart=shared
-o myprogram myprogram.cu
```

Windows

```
nvcc.exe -ccbin "C:\vs2008\VC\bin"
-Xcompiler "/EHsc /W3 /nologo /O2 /Zi /MT /DELAY" --cudart=shared
-o "Release\myprogram.exe" "myprogram.cu"
```

리눅스와 맥을 위해, `-rpath` 옵션이 앞서와 같이 사용됩니다. 윈도우즈를 위해, `/DELAY` 옵션이 사용됩니다. CUDA DLL 들을 포함하는 디렉토리를 특정하기 위해, 어떤 쿠다 API 함수로의 첫 번째 호출 전에, 이는 응용 프로그램의 `SetDllDirectory()` 호출을 요구합니다.

노트: 윈도우즈 8 을 위해, `SetDllDirectory()` 대신 `SetDefaultDllDirectories()`와 `AddDllDirectory()`가 꼭 사용되어야 합니다. 더 자세한 정보는 이 루틴들을 위한 MSDN 문서를 참고해 주십시오.

16. 배포 기반시설 도구들 (Deployment Infrastructure Tools)

16.1. Nvidia-SMI

엔비디아 시스템 관리 인터페이스(`nvidia-smi`)는 관리와 엔비디아 GPU 디바이스들의 모니터링을 돕는 커맨드라인 유틸리티입니다. 이 유틸리티는 관리자들이 GPU 디바이스 상태를 조회할 수 있게 하고, 적절한 특전을 가지고, 관리자들이 GPU 디바이스 상태를 고칠 수 있게 합니다. 비록 다른 엔비디아 GPU 들도 제한적으로 지원하긴 하지만, `nvidia-smi` 는

테슬라와 쿼드로 GPU 들을 대상으로 합니다. nvidia-smi 는 리눅스 상의 엔비디아 GPU 디스플레이 드라이버들과 함께 출하됩니다. 그리고 그리고 64 비트 윈도우즈 서버 2008 R2 와 윈도우즈 7 과 함께 출하됩니다. nvidia-smi 는 문의된 정보를 XML 또는 사람이 읽을 수 있는 평범한 글로 (표준 출력 또는 파일로) 출력할 수 있습니다. 자세한 사항은 nvidia-smi 문서를 보십시오. 새로운 버전의 nvidia-smi 는 이전 버전들과 호환되지 않을 수도 있음에 유념하십시오.

16.1.1. 조회할 수 있는 상태 (Queryable state)

ECC 오류 카운트 (error counts)

고칠 수 있는 단일 비트와 찾을 수 있는 더블 비트 오류들이 보고됩니다. 오류 카운들은 현재 부트 사이클과 GPU 의 수명을 위해 제공됩니다.

GPU 사용성 (utilization)

그 GPU 의 계산 자원들과 메모리 인터페이스를 위한, 현재 사용율이 보고됩니다.

활성 계산 프로세서 (Active compute process)

상응하는 프로세스 이름/ID 와 할당된 GPU 메모리와 함께, 그 GPU 에서 동작하고 있는 활성 프로세스들의 목록이 보고됩니다.

클럭들과 성능 상태 (Clocks and performance state)

현재 GPU 성능 상태 (pstate) 뿐 아니라 몇몇 중요한 클럭 도메인들을 위해, 최대와 현재 클럭 율들이 보고됩니다.

온도와 팬 속도 (Temperature and fan speed)

액티브 쿨링을 가진 제품들의 팬 속도들과 함께, 현재 GPU 코어 온도가 보고됩니다.

전력 관리 (Power management)

이 측정들을 보고하는 제품들을 위한, 현재 보드 전력 소모와 전력 제한들이 보고됩니다.

제품 정보 확인 (Identification)

다양한 동적 그리고 정적 정보가 보고됩니다. 여기에는 보드 시리얼 넘버들, PCI 디바이스 ID 들, VBIOS/Inforom 버전 넘버들 그리고 제품 이름들이 포함됩니다.

16.1.2. 고칠 수 있는 상태 (Modifiable state)

ECC 모드 (mode)

ECC 보고하기 활성화와 비활성화.

ECC 리셋 (reset)

단일 비트와 더블 비트 ECC 오류 카운트들을 초기화(clear).

계산 모드 (Compute mode)

계산 프로세스들이 그 GPU 에서 동작할 수 있는지 그리고 그 프로세스들이 다른 계산 프로세스들과 배타적으로 또는 동시에 동작할 수 있는지를 표시.

지속 모드 (Persistence mode)

아무 응용 프로그램들도 그 GPU 에 연결되지 않을 때, 엔비디아 드라이버가 로드된 채로 유지되었는지를 표시. 대부분 상황에서, 이 옵션은 활성화되는 것이 좋음.

GPU 리셋 (reset)

이차적(secondary) 버스 리셋을 통해, GPU 하드웨어와 소프트웨어 상태를 재초기화.

16.2. NVML

엔비디아 관리 라이브러리(NVML)는 C 기반 인터페이스입니다. 이 라이브러리는 `nvidia-smi` 를 통해 노출된 쿼리들과 명령어들의 직접 접근을 제공합니다. 이 인터페이스는 제 3 자 시스템 관리 응용 프로그램들을 위한 플랫폼으로 의도되었습니다. NVML API 는 엔비디아 개발자 웹사이트에서 테슬라 개발 키트의 로서 사용할 수 있습니다 (단일 헤더 파일을 통해). 또한, 여기에는 PDF 문서, 스터브 라이브러리(stub library), 그리고 샘플 응용 프로그램들이 포함됩니다. <http://developer.nvidia.com/tesla-deployment-kit> 를 보십시오. NVML 의 각 새로운 버전은 이전 버전들과 호환됩니다(backward-compatible).

추가적인 일련의 펄(Perl)과 파이썬(Python) 바인딩들도 NVML API 를 위해 제공됩니다. 이 바인딩들은 같은 특징들을 C 기반 인터페이스로 노출시킵니다. 그리고 또한 백워드 호환성을 제공합니다. 펄 바인딩은 CPAN 을 통해 제공됩니다. 파이썬 바인딩은 PyPI 를 통해 제공됩니다.

이 모든 제품들은 (`nvidia-smi`, NVML, 그리고 NVML 언어 바인딩들) 각 새로운 쿠다 릴리스와 함께 갱신되고, 거의 같은 기능성을 제공합니다.

추가 정보는 <http://developer.nvidia.com/nvidia-management-library-nvml> 를 참고하십시오.

16.3. 클러스터 관리 도구들 (Cluster Management Tools)

GPU 클러스터를 관리하는 것은 최대 GPU 사용성을 얻는 것과 최대 성능을 얻는 데 도움이 될 것입니다. 많은 산업계의 가장 인기있는 클러스터 관리 도구들은 이제 NVML 을 통해 쿠다 GPU 들을 지원합니다. 이 몇몇 도구들의 목록은 <http://developer.nvidia.com/cluster-management> 에 제공됩니다.

16.4. 컴파일러 JIT 캐시 관리 도구들 (Compiler JIT Cache Management Tools)

한 응용 프로그램이 런타임에 로드한 PTX 디바이스 코드는 디바이스 드라이버에 의해 바이너리 코드로 더 컴파일됩니다. 이는 just-in-time (JIT) 컴파일이라 불립니다. JIT 컴파일은 응용 프로그램 로드 시간을 늘립니다. 그러나, 응용 프로그램들이 최신 컴파일러 향상으로 인한 이득을 사용할 수 있게 합니다. 이는 또한 응용 프로그램이 컴파일 될 때 없던 디바이스들에서 응용 프로그램들을 동작시키기 위한 유일한 방법입니다.

PTX 디바이스 코드의 JIT 컴파일이 사용될 때, 엔비디아 드라이버는 결과 바이너리 코드를 디스크에 캐시합니다. 캐피 위치(cache location)와 최대 캐시 크기와 같은 이 동작의 몇몇 양상들은 환경 변수들의 사용을 통해 제어될 수 있습니다. 쿠다 C 프로그래밍 가이드의 Just in Time 컴파일을 보십시오.

16.5. CUDA_VISIBLE_DEVICES

응용 프로그램 시작 전에, `CUDA_VISIBLE_DEVICES` 환경 변수를 통해, 한 쿠다 응용 프로그램에 의해 보여질 수 있고 열거될, 설치된 쿠다 디바이스들은 재배열될 수 있습니다.

디바이스들은 그 응용 프로그램에 보여질 수 있어야 합니다. 이는 시스템 전체에 걸친 열거될 수 있는 디바이스들의 콤마로 구분된 목록 포함을 통해 수행됩니다. 예를 들어, 전체 시스템에서 디바이스 0 과 2 만 사용하기 위해, 그 응용 프로그램을 시작하기 전에 `CUDA_VISIBLE_DEVICES=0,2` 으로 설정합니다. 그 응용 프로그램은 이 디바이스들은 디바이스 0 과 디바이스 1 로 각각 열거할 것입니다.

A. 권고 그리고 모범 관례 (Recommendations and Best Practices)

이 부록은 이 문서에서 설명된 최적화를 위한 추천들의 요약을 제공합니다.

A.1. 전체적 성능 최적화 전략들 (Overall Performance Optimization Strategies)

성능 최적화 세 가지 전략들을 중심으로 돕니다.

- 병렬 처리 최대화하기
- 최대 메모리 대역폭을 얻기 위해 메모리 사용 최적화하기
- 최대 명령어 처리량을 얻기 위해 명령어 사용 최적화하기

병렬 실행 최대화하기는 알고리즘의 데이터 병렬성을 최대한 드러내는 방식으로 시작한다. 일단 그 알고리즘의 병렬성이 드러나면, 그 알고리즘은 하드웨어에 가능한 효율적으로 사상될 필요가 있습니다. 이것은 각 커널 시작(launch)의 실행 구성을 신중하게 선택함으로써 됩니다. 또한, 그 응용 프로그램은 스트림들을 통해 그 디바이스의 동시 실행을 명시적으로 드러냄으로써 더 높은 수준에서 병렬 실행을 최대화해야 합니다. 호스트와 디바이스 사이 동시 실행을 최대화하는 것 뿐 아니라고요.

메모리 사용 최적화하기는 호스트와 디바이스 사이 데이터 전송들을 최소화하기에서 시작합니다. 왜냐하면 그 전송들은 내부 디바이스 데이터 전송들보다 훨씬 더 낮은 대역폭을 가지기 때문입니다. 글로벌 메모리로의 커널은 접근 또한 그 디바이스 상의 공유 메모리 사용을 최대화함으로써 최소화되어야 합니다. 때때로, 가장 좋은 최적화는 첫 번째 위치에서 언제든지 그것이 필요해질 때 단순히 그 데이터를 다시 계산함으로써 어떤 데이터 전송도 피하는 것입니다.

실효 대역폭은 각 타입의 메모리를 위한 접근 패턴에 따라 약 10 배까지 바뀔 수 있습니다. 메모리 사용 최적화에서 다음 단계는 따라서 최적 메모리 접근 패턴들에 따라 메모리 접근들을 구성하는 것입니다. 이 최적화는 특히 글로벌 메모리 접근들을 위해 중요합니다. 왜냐하면 접근 지연 시간이 수백 클럭 사이클이기 때문입니다. 대조적으로, 공유 메모리 접근들은 대개 높은 정도의 बैं크 충돌이 있을 때만 최적화할 가치가 있습니다.

명령어 사용 최적화하기에 대해 말하자면, 낮은 처리량을 가진 산술 명령어들의 사용은 꼭 피해야 합니다. 마지막 결과에 영향을 주지 않을 때 속도를 위해 정밀도를 희생하는 것을 암시합니다. 예를 들어, 정규 함수들 대신 intrinsics 을 배정밀 대신 단정밀을 사용할 수 있습니다. 마지막으로, 그 디바이스의 SIMT(single instruction multiple data) 본질 때문에, 명령어들의 흐름을 제어하기 위해 특별한 주의가 기울여져야 합니다.

B. nvcc Compiler Switches

B.1. nvcc

호스트 시스템과 쿠다 어셈블리 또는 그 디바이스의 바이너리 명령어들을 위해, 엔비디아 nvcc 컴파일러 드라이버는 .cu 파일들을 C 로 바꿉니다. nvcc 는 많은 커맨드라인 파라미터들을 지원합니다. 그 중에서 다음은 최적화와 관련된 모범 규례들을 위해 특히 유용합니다.

- `-maxregcount=N` 은 파일 하나 당 수준에서, 커널이 사용할 수 있는 레지스터들의 최대 개수를 특정합니다. 레지스터 압력을 보십시오 (또한 커널 당 단위로 사용된 레지스터 수를 제어하기 위해 쿠다 C 프로그래밍 가이드의 실행 구성에 논의된 `__launch_bounds__` 한정자를 보십시오).
- `--ptxas-options=-v` 또는 `-Xptxas=-v` 는 커널 당 레지스터, 공유, 그리고 상수 메모리 사용을 열거합니다.
- `-ftz=true` (비정규화된(denormalized) 숫자들이 0 으로 초기화됩니다)
- `-prec-div=false` (덜 정확한 나누기)
- `-prec-sqrt=false` (덜 정확한 제곱근)
- nvcc 의 `-use_fast_math` 컴파일러 옵션은 매 `functionName()` 호출을 대등한 `__functionName()` 호출로 강제 변환합니다. 이는 줄어든 정밀도와 정확도로 코드를 더 빠르게 실행합니다. [수학 라이브러리들](#)을 보십시오.

공지

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other

information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

© 2007–2015 NVIDIA Corporation. All rights reserved.

This product includes software developed by the Syncro Soft SRL (<http://www.sync.ro/>).

더 읽을 거리: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#ixzz47ZtjzJzA>

친구 맺기(follow us): [@GPUComputing on Twitter](#) | [NVIDIA on Facebook](#)