Prezentacja sesja kół - shownotes

Slajd 1

Tytułowy

Slajd 2

Podczas poprzedniej Sesji Kół Naukowych miałem zaszczyt prezentować mój oryginalny pomysł - solwer multifrontalny na GPU. Był on stworzony przy użyciu frameworka OpenCL i wykorzystywał masowo równoległe przetwrzanie na GPU.

Slajd 3

Jednym z ciekawszych elementów tego pomysłu była metoda dekompozycji macierzy - z macierzy rzadkiej były wydzielane części, obiektywnie patrząc, gęste, do przetwarzania których był przystosowany solwer. Części mogły być przetwarzane w dużej mierze niezależnie od siebie - tylko pewne ich obszary były od siebie wzajemnie zależne. Przetworzenie tych zależności można było zrealizować już po współbieżnym przetworzeniu części.

Slajd 4 (wady oryginalnego podejścia)

Jak każde, zaproponowane wtedy rozwiązanie miało pewne wady. Przede wszystkim, wykorzystywało tylko jedno z dostępnych urządzeń OpenCL; już wtedy, dzięki uprzejmości Katedry Informatyki Stosowanej i Modelowania, miałem do dyspozycji maszynę z czterema procesorami graficznymi, więc był to pierwszy podjęty kierunek rozwoju. Mimo zastosowania sprytnych optymalizacji, solwer wymagał też sporo pamięci, a w przypadku awarii systemu operacyjnego lub braku zasilania rozwiązanie trzeba było powtórzyć.

Slajd 5 (nowy kierunek)

Dlatego zidentyfikowałem nowy kierunek moich badań. Przede wszystkim, chciałem użyć wszystkich dostępnych urządzeń - ale zacząłem myśleć o tym, jak zbudowane są takie "naukowe" maszyny jak ta z którą pracuję. Owszem, ma wiele GPU - bardzo mocnych i bardzo drogich, na specjalnej płycie głównej, również kosztującej więcej niż konsumencka, umieszczonej w specjalistycznej obudowie. Cała konfiguracja systemu operacyjnego jest specyficzna, co powoduje że maszyna ta ma wąskie zastosowania. A cała potęga GPU - podkreślam to przy każdej okazji - tkwi w zastosowaniu urządzeń klasy konsumenckiej. Moją główną myślą było stworzenie oprogramowania które to wykorzysta i będzie miało praktyczne zastosowania.

Slajd 6 (założenia projektu)

Głównym założeniem projektu było wykorzystanie każdego sprzętu do jakiego uda się uzyskać dostęp, a który wspiera OpenCL. Nieważne gdzie ten sprzęt jest fizycznie umieszczony, co jeszcze takie urządzenie "robi", nieważne jaką ma moc, jakie ma połączenie z internetem i jak długo oraz jak wydajnie może pracować. Nic nie może się zmarnować.

Slajd 7 (założenia projektu 2)

Oprócz tego nie chciałem już sytuacji, w której wrzucam do solwera jeden układ równań i czekam na jego rozwiązanie zanim wrzucę kolejny - chciałem rozwiązywać wiele problemów na raz. Skoro już usuwam uwarunkowania pamięciowe, to chciałem też zająć się naprawdę dużymi układami równań - nie takimi z tysiącem niewiadomych, ale z dziesiątkami lub setkami tysięcy niewiadomych, jeśli nie większymi. Oprogramowanie miało też być naturalnie odporne na błędy: od błędów formatu pliku, do całkowitej katastrofy.

Slajd 8 (inspiracja)

Zainspirowały mnie istniejące już projekty przetwarzania rozproszonego na masową skalę, takie jak SETI@home - na podstawie którego później stworzono Berkeley Open Infrastructure for Network Computing, BOINC, czy Folding@home. Zobaczyłem tutaj też - przez porównanie ze schematem działania tych usług - potencjał wykorzystania tu ponownie mojej metody dekompozycji macierzy na niewielkie części; takie części mogłyby stanowić osobne zadania, a serwer mógłby je tylko podawać pewnej, właściwie dowolnej ilości klientów.

Slajd 9 (schemat)

Schemat rozwiązania jest więc bardzo prosty: mając macierz, podajemy ją serwerowi. Serwer stosuje mój algorytm dekompozycji - dzieli macierz na części, na osobne zadania. Potem rozdziela te zadania według możliwości i zapotrzebowania między podłączonych do niego klientów; nazywam to falą rozwiązania. Kiedy fala zostaje zakończona, serwer sprawdza - według zależności między częściami - czy osiągnięto już rozwiązanie, czy konieczna jest kolejna fala rozwiązania, i zachowuje się odpowiednio.

Slajd 10 (przewaga rozwiązania)

Wspomniane projekty wykorzystują zmarnowany potencjał urządzeń, które mają zasilanie, ale nie wykonują pracy. Uznałem to za dobry kierunek i wzór do

naśladowania. Stworzyłem oprogramowanie podobne do BOINC pracujące z dostępnymi technologiami i popularnymi urządzeniami bez dodatkowych inwestycji, ale jeszcze prostsze - żeby mogli z nim pracować nie tylko programiści. Zrealizowałem też swój - powiem nieskromnie - ambitny cel: stworzenie możliwości rozwiązywania problemów niepraktycznych lub niemożliwych do rozwiązania.

Slajd 11 (architektura)

Podobnie jak w BOINC, wybrałem architekturę klient-serwer. Z początku rozważałem użycie MPI, de facto standardu w programowaniu rozproszonym, ale ostatecznie zrezygnowałem; pomyślałem o przypadkach gdzie mamy maszyny w rozłącznych sieciach i lokalizacjach, bądź na gorszych połączeniach. Wybrałem więc protokół HTTP: jest popularny, co ułatwi tworzenie klienta, i nie jest blokowany, co umożliwia współpracę przez blokowane połączenia.

Slajd 12 (serwer)

Serwer został zbudowany w Ruby on Rails. Zapewniło to jednolitość API, bezstanowość serwera - dzięki czemu spełniło się moje założenie rozwiązywania wielu problemów na raz, i zwiększyło to odporność serwera na błędy. Dzięki temu również ten sam serwer dostarcza i usługę dla maszyny i dla człowieka. Interfejs użytkownika jest zbudowany w nowoczesnych technologiach webowych:

Slajd 13 (serwer UI)

. . . aby zlecić rozwiązanie zadania, trzeba wypełnić trzy pola i nacisnąć przycisk w przeglądarce.

Slajd 14 (serwer UI 2)

Zobaczenie stanu rozwiązań jest jeszcze prostsze. To jest właśnie ta prostota do której dążyłem: nie trzeba tu programisty.

Slajd 15 (klient)

Mój referencyjny klient został stworzony w Ruby. W tym języku łatwo prototypować, dzięki temu mogłem niewielkim kosztem eksplorować różne rozwiązania - lepsze lub gorsze - oraz bardzo dynamicznie zmieniać swoje podejście. Klient pracuje jako usługa w tle; zajmuje urządzneia tylko wtedy, kiedy serwer podaje zadania, i jest stosunkowo lekki w pamięci i procesorze. Jest w pełni asynchroniczny, więc nawet jeśli dane zadanie jest duże (przy pobieraniu, wysyłaniu lub przetwarzaniu), zablokuje tylko jeden wątek przetwarzania.

Slajd 16 (elementy obliczeniowe)

Mówię tu dużo o Ruby, ale jeśli chodzi o szybkość, trudno przebić C lub C++: jest szybki, świetnie zarządza pamięcią, mnogość bibliotek naukowych ułatwia programowanie takich rozwiązań. Dlatego elementy obliczeniowe tyleż klienta, co serwera stworzyłem w języku C++ i połączyłem z Ruby przez rozszerzenia natywne Ruby, co pozwala mi uruchamiać kod natywny - kompilowany - w kontekście interpretowanego Ruby. Jest to eleganckie, hybrydowe rozwiązanie.

Slajd 17 (wstępne wyniki)

na razie wyników nie ma; będą pewnie jakieś wstępne, zależność kopa rozmiaru części dla jakiejś normalnej macierzy i jakiejś popieprzonej

Slajd 18 (zysk z użytych technologii)

To wszystko składa się na potężne, rozszerzalne rozwiazanie: łatwo rozszerzać serwer, bo przeciętny programista nauczy się Ruby w tydzień. Łatwo modyfikować klienta - lub napisać własnego! Interfejs serwera jest udokumentowany, REST to standard. Nieprzypadkowo nazywam swojego klienta "referencyjnym". Korzystam z całej dynamiki i deklaratywności Ruby tam gdzie to możliwe, z całej szybkości C++ tam gdzie ma ona znaczenie, i z całej mocy którą dostarcza OpenCL. Platforma jest więc idealna do zmodyfikowania pod konkretny cel - do użycia jako framework przez specjalistę, oraz do wykorzystania wprost przez specjalistę, ale w innej dziedzinie.

Slajd 19 (Dalszy rozwój)

Rozwiązanie jest oczywiście nadal rozwijane. W tej chwili badam możliwości usprawnienia komunikacji po HTTP, zwiększenia asynchroniczności i w konsekwencji zmniejszenia obciążenia serwera. Badam też wpływ parametrów podziału macierzy na uzyskiwanie wyników, oraz wpływ parametrów uruchomieniowych OpenCL na szybkość przetwarzania po stronie klienta.