Проект

по Системи за Паралелна Обработка

Стабилизиран метод на двуспрегнати градиенти за решаване на асиметрични системи линейни уравнения (BiCGSTAB)

Изготвили:
Олег Каменщик
Александър Димитров

Ръководител:
ас. Христо Христов

Проверил: .....
(ас. Христо Христов)

1. Постановка на задачата

Намирането на решение на линейна система от уравнения от вида **Ax=b** има всевъзможни приложения и поради това е изследвано от алгебрата до голяма степен. Има 2 начина на подхождане към решаването на такава система – директен и итеративен подход, като всеки от 2-та вида има своите предимства и недостатъци. Най-често използваният директен подход е т.нар. *Гаусова елиминация*, която след краен брой стъпки ни предоставя вектора решение **x**, стига аритметичните операции да са извършени точно. Итеративните подходи от своя страна се срещат при решаването на всякакви линейни системи, когато размера на матрицата е голям и има преобладаващи нулеви елементи.

Стабилизираният метод на двуспрегнати градиенти (*BiCGSTAB*) е итеративен подход за решаване на този тип задачи, като той е по-бърз и има по-гладка непрекъснатост на резултатите си от оригиналния метод на двуспрегнати градиенти (*BiCG*), както и други варианти като метод на спрегнати градиенти квадрати (*CGS*).

Тъй като матриците, с които работим са с преобладаващи нулеви елементи (*sparse matrix*) използваме формат, който ни позволява матрица да бъде съхранена в по-малко памет, а именно **CSR** (***C****ompressed* ***S****parse* ***R****ow*) формат. Ако не се използва този формат, една квадратна матрица **N***x***N** ще заема средно $\mathbf{8n^2}$ байта, което за матрица с размер 4000 означава 128MB памет, докато ако се пазят само ненулевите елементи, ще използваме памет пропорционална на процента ненулеви елементи спрямо цялостния размер на матрицата. Както да спестява място на диска, това помага и при зареждането на матриците в рам паметта, като позволява обработката на големи матрици, които в противен случай не могат да се обработват ако се зареждат в пълния си размер. Още един бонус е, че това забързва и операциите с такава матрица, тъй като се знае къде са нулите и няма нужда да се работи с тях.

2. Описание на реализирания алгоритъм

За паралелното решаване на задачата за решаване на големи системи линейни уравнения е избран подходът на многонишковото програмиране. Той предоставя възможност за оптимално намаляване на използвания ресурс време, тъй като позволява отделните процеси извършващи изчисленията да използват едно и също споделено копие на матриците и векторите и да нанасят

резултатите върху друг споделен участък памет представляващ вектор, който да се ползва без синхронизация, тъй като отделните подпроцеси се нуждаят от различни негови участъци.

Прочитането на входните данни става от текстов файл в специален формат, който съдържа информация за вектора **b** и за матрицата **A** в гореспоменатия **CSR** формат. След зареждането ѝ в паметта в зависимост от версията на алгоритъма, ако е процесорната, се започва с изчисленията, а ако е CUDA, се копират всички данни в паметта на видеокартата (за да се намали латентността на паметта) и тогава се започва с процеса, след което резултата се копира обратно в оперативната памет.

Реализацията на итеративният алгоритъм е постъпково описана в *Wikipedia* - http://en.wikipedia.org/wiki/Biconjugate_gradient_stabilized_method като ние се водим по нея. Там където има операции матрица с вектор или вектор с вектор сме реализирали функции или кернели, които да си разпределят задачата и да работят с отделни редове (или участъци от редове) от матрицата, ако става въпрос за операция с матрица или отделни елементи от вектоите (или участъци от елементи) ако става въпрос за операция между вектори. При процесорната версия задачата се разпределя поравно сред нишките, а при CUDA версията задачата се разделя на блокове с определен брой нишки, като всяка нишка обработва само 1 ред или елемент, т.е. броя на блоковете се изчислява от това колко голяма е матрицата и колко нишки се падат на блок.

3. Тестови замервания

Информация относно интерфейса на програмата:

1) Проекта има съответно 3 изпълними файла – за генериране на матрица, CPU версия и GPU версия, като има създаден *Makefile* и се компилират с командата **make all** в основната директория и могат да бъдат намерени след компилация в ***bin*** директорията:

2) Получените изпълними файлове приемат като позиционни параметри:

./bin/genMatrix <size> <percent_non_zero_elems> <output_file>

Тъй като матриците, с които работим за квадратни size указва размера на едната им страна, 2-рия параметър показва на какъв процент да са запълнени (0 до 100), а 3-тия параметър указва файла, в който ще бъде записана матрицата, за да се използва за изчисления от другите 2 файла.

./bin/BiCGSTABCPU <input_matrix_file> <output_file> <thread_count>

./bin/BiCGSTABGPU <input_matrix_file> <output_file> <threads_per_block>

И 2-та файла приемат като първи аргумент името на матрицата, от която да четат данни, като 2-ри параметър файл, в който да запишат данни за изпълнението си и като 3-ти параметър съответно брой нишки за CPU версията или брой нишки за блок за GPU версията, като ако 3-тия аргумент е пропуснат по подразбиране програмите се пускат с 1 нишка или 1 нишка за блок.

3) Примери за използване:

- ./bin/genMatrix 2000 75 data/2000_75_matrix

Генерира се матрица 2000x2000 пълна на 75% с ненулеви елементи и се записва във файла *data/2000_75_matrix.*

- ./bin/BiCGSTABCPU data/2000_75_matrix data/results.txt 2

Изчислява се решението на матрицата записана в *data/2000_75_matrix*, като данни за изпълнението се записват в *data/results.txt* и се използват 2 нишки.

- ./bin/BiCGSTABGPU data/2000_75_matrix data/results.txt 32

Изчислява се решението на матрицата записана в *data/2000_75_matrix*, като данни за изпълнението се записват в *data/results.txt* и се използват 32 нишки за block.

Решението беше разработено на C++ и CudaC, за тестване се използва четириядрен процесор – Intel Core i7-4700HQ @ 2.4GHZ и видеокарта – NVidia GeForce GTX 880M с 8 мултипроцесора и 1536 CUDA ядра. Както беше очаквано се наблюдава ускорение спрямо по-малко нишки само до 4 нишки, заради ограниченията на системата до 4 ядра. При наличието на повече ядра би трябвало да се наблюдава ускорение правопропорционално на броя ядра, на които е пуснато приложението. За CUDA версията, тъй като бройката блокове е пропорционална на размера на матрицата (а той е голям), ускорение се наблюдава при повишаване на броя нишки за блок, тъй като винаги се използват всички налични мултипроцесори.

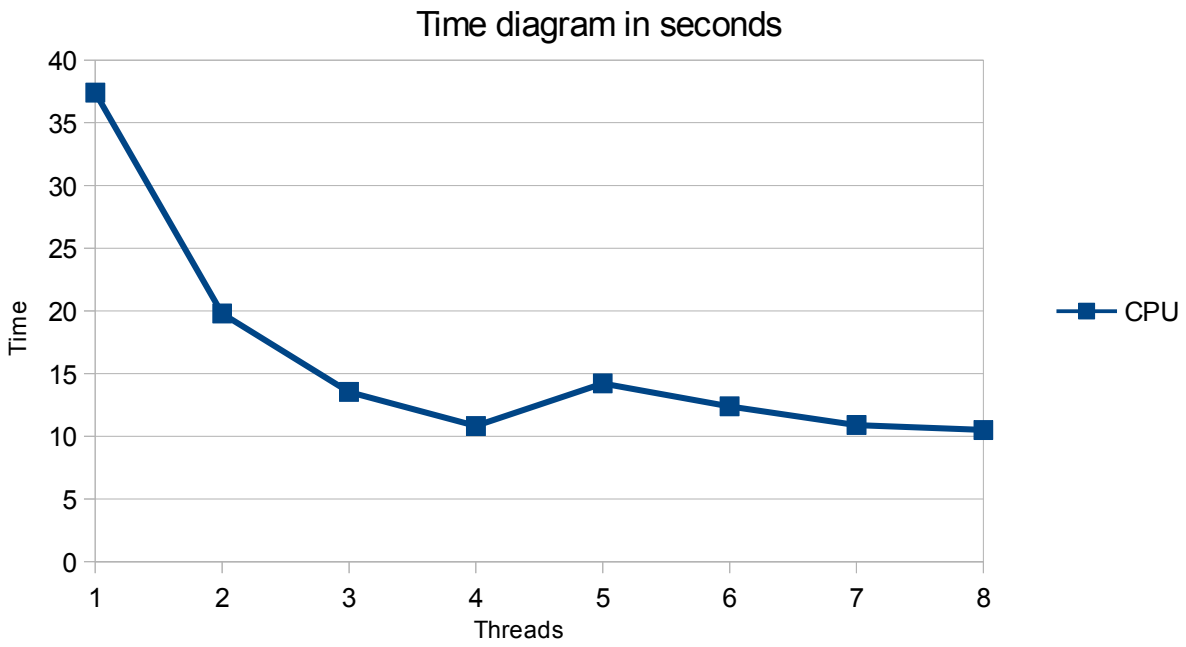
С цел да се измери ускорението **S** (*забързване, boost*) и ефективността **E** (*efficiency*) на описания алгоритъм, където ако **T(p)** е времето необходимо за завършване на работата на алгоритъм с **p** на брой нишки то:

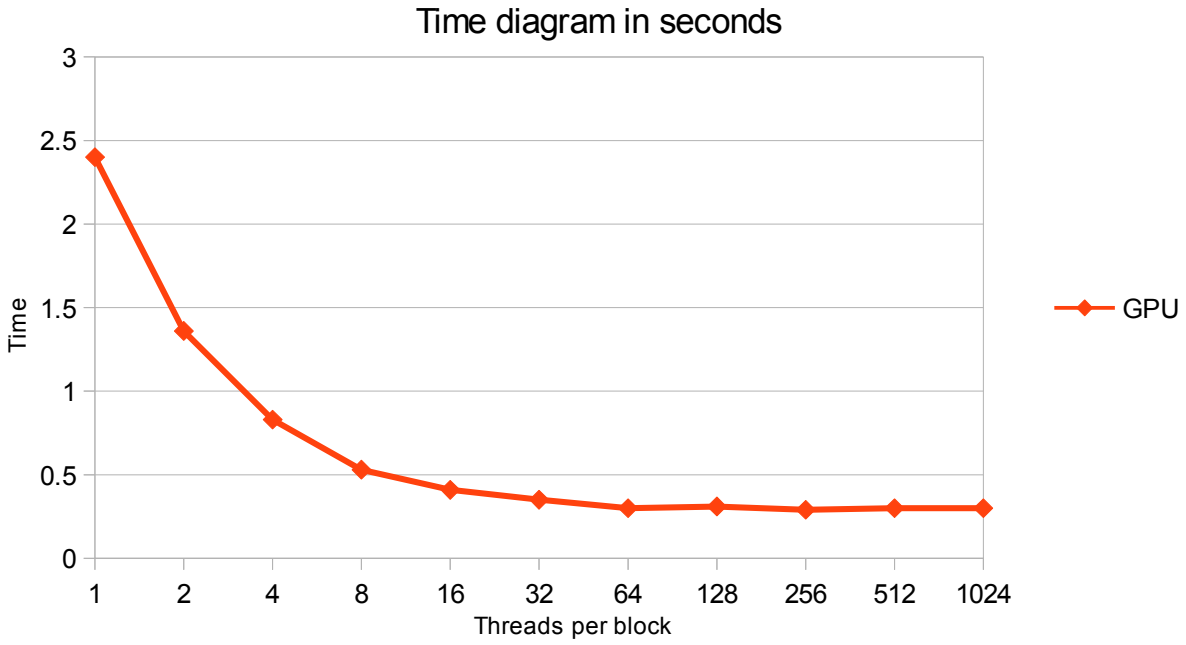
$$S(p) = T(1) / T(p)$$

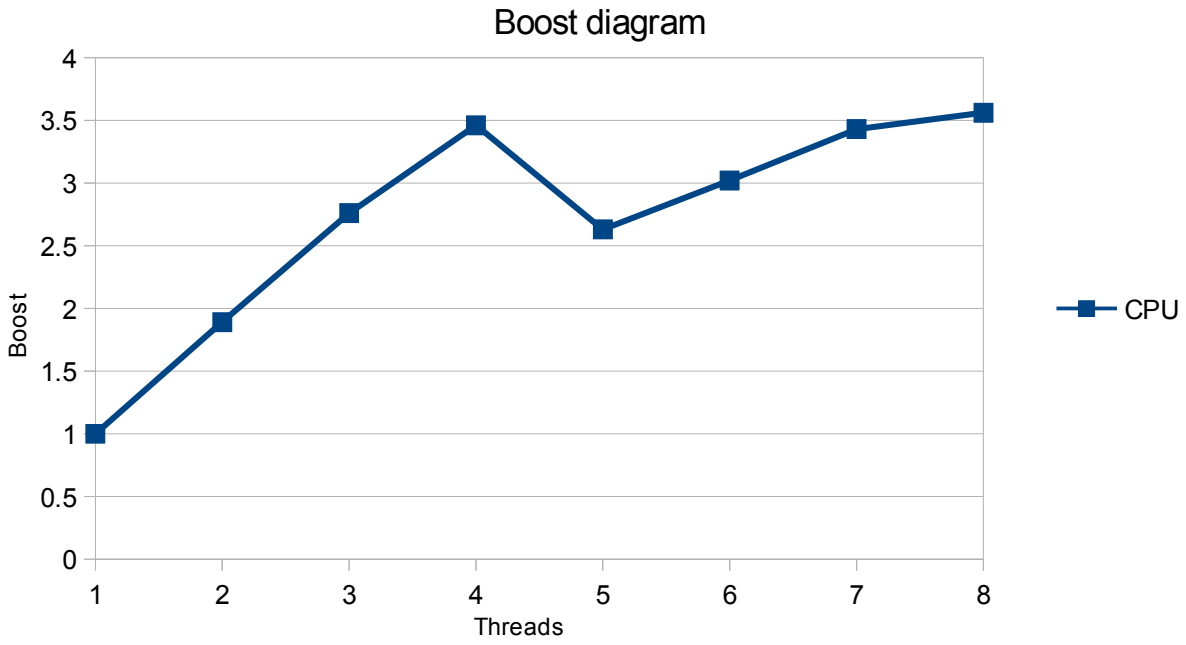
$$E(p) = S(p) / p$$

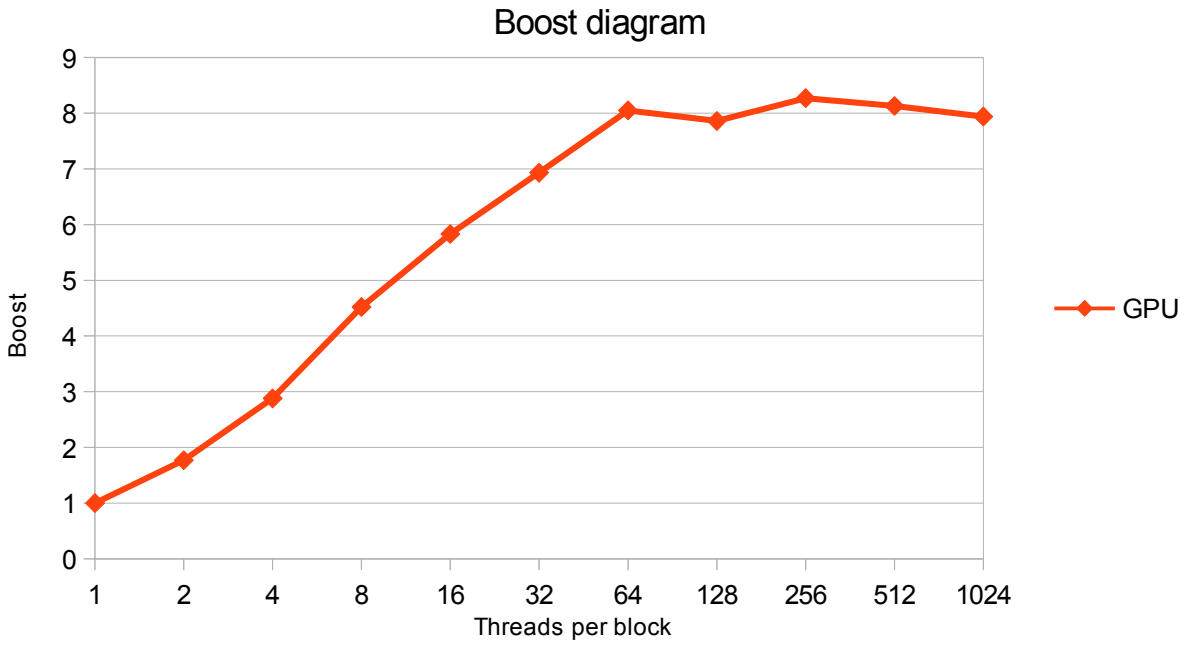
За процесорната версия бяха направени тестове с матрица ***5000x5000*** и 75% плътност, което прави 18,748,900 ненулеви елемента. За CUDA версията бяха направени тестове с матрица ***50000x50000*** и 5% плътност, което прави 124,992,400 ненулеви елемента.

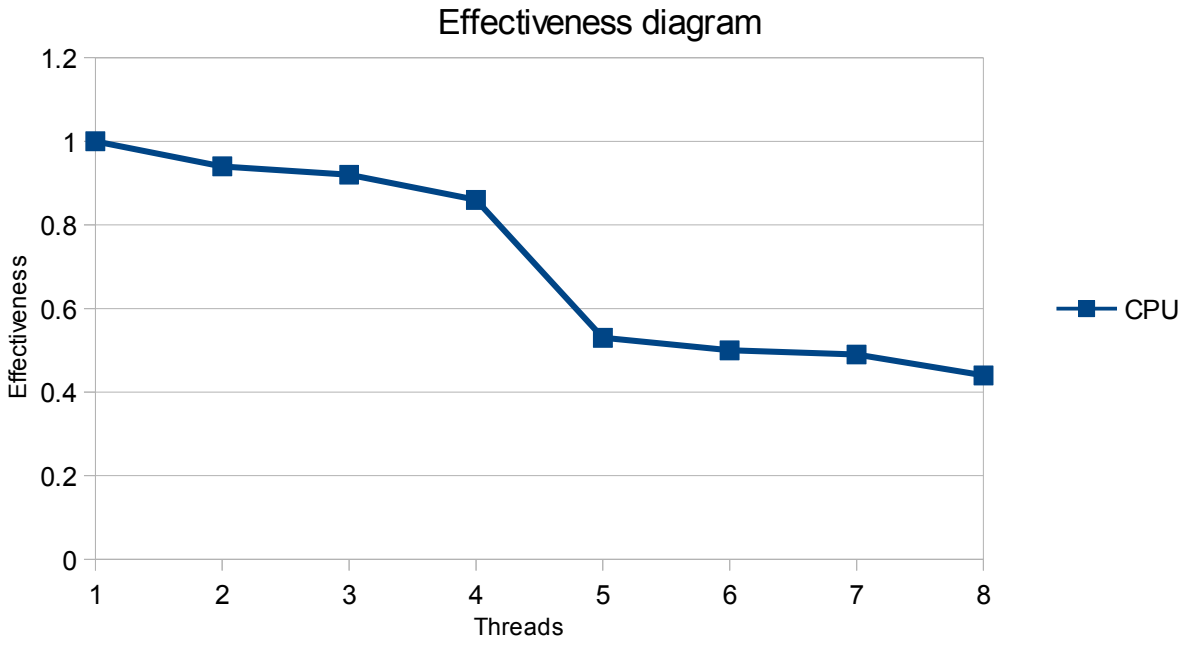
Диаграми

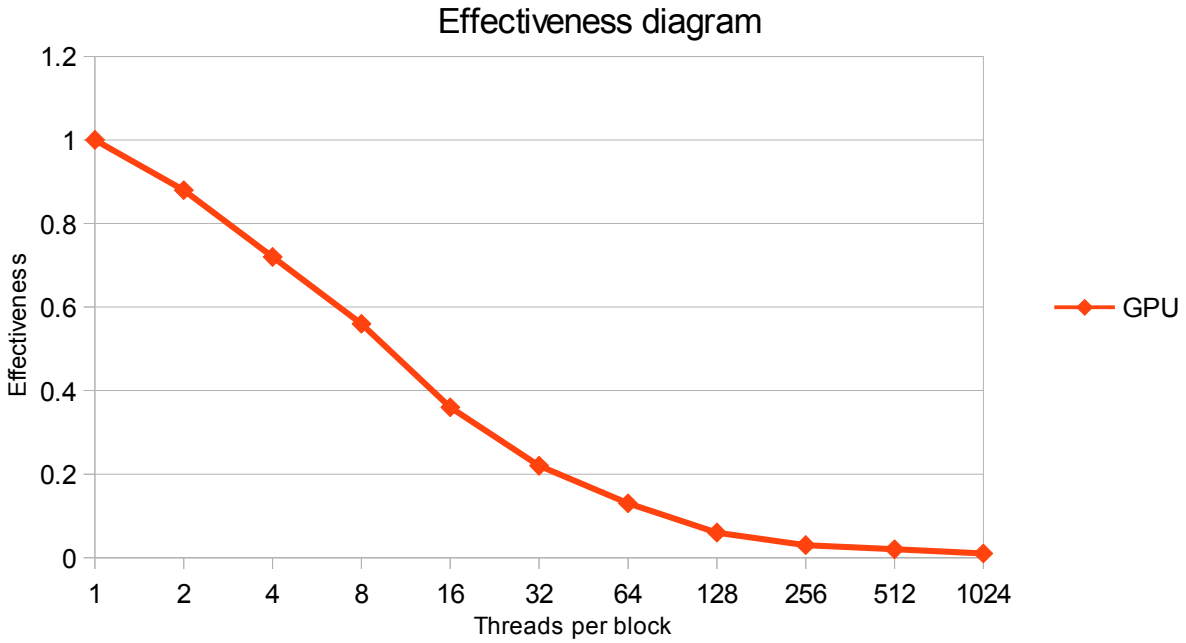

Time diagram in seconds
Time
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
Threads
CPU


Time diagram in seconds
Time
3
2.5
2
1.5
1
0.5
0
1
2
4
8
16
32
64
128
256
512
1024
Threads per block
GPU


Boost diagram
Boost
Threads
CPU
0
0.5
1
1.5
2
2.5
3
3.5
4
1
2
3
4
5
6
7
8


Boost diagram
Boost
Threads per block
GPU
0
1
2
3
4
5
6
7
8
9
1
2
4
8
16
32
64
128
256
512
1024


Effectiveness diagram
1.2
1
0.8
0.6
0.4
0.2
0
Effectiveness
1
2
3
4
5
6
7
8
Threads
CPU


Effectiveness diagram
1.2
1
0.8
0.6
0.4
0.2
0
Effectiveness
1
2
4
8
16
32
64
128
256
512
1024
Threads per block
GPU